Get Started: Chat application

In this guide we'll create a basic chat application. It requires almost no basic prior knowledge of Node.JS or Socket.IO, so it's ideal for users of all knowledge levels.

Introduction

Writing a chat application with popular web applications stacks like LAMP (PHP) has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it's a lot slower than it should be.

Sockets have traditionally been the solution around which most realtime chat systems are architected, providing a bi-directional communication channel between a client and a server.

This means that the server can *push* messages to clients. Whenever you write a chat message, the idea is that the server will get it and push it to all other connected clients.

The web framework

The first goal is to setup a simple HTML webpage that serves out a form and a list of messages. We're going to use the Node.JS web framework `e e` to this end. Make sure Node.JS is installed.

First let's create a `ackage j` manifest file that describes our project. I recommend you place it in a dedicated empty directory (I'll call mine `cha e a le`).

```
 a e      cke  cha  e a  le
 e  i
de c i  i        fi      cke  i  a
de e de cie
```

Now, in order to easily populate the `de e de cie` with the things we need, we'll use `i all a e`:

```
    i   all    a e e   e
```

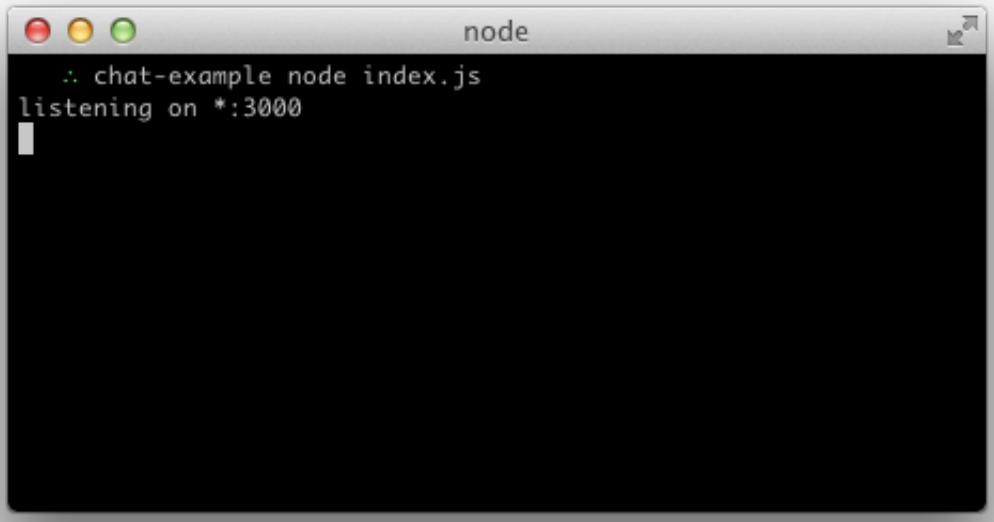
Now that express is installed we can create an `i de j` file that will setup our application.

```
 a  a     e  i e  e  e
 a  h      e  i e  h     Se  e  a

a  ge      f  c i   e   e
   e  e d  h  Hell    ld  h


h   li  e       f  c i
  c   le l g  li  e i g
```

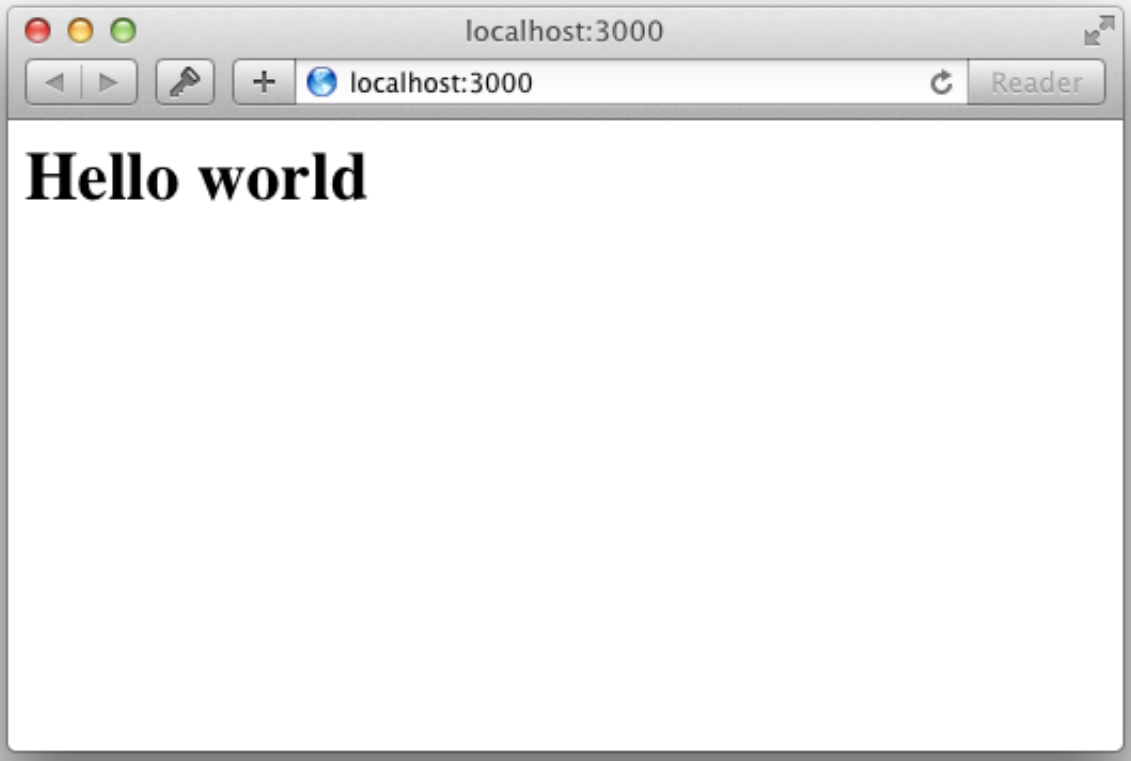
This translates into the following:

1. Express initializes `a` to be a function handler that you can supply to an HTTP server (as seen in line 2).
2. We define a route handler that gets called when we hit our website home.
3. We make the http server listen on port 3000.

If you run `de i de j` you should see the following:



And if you point your browser to `h l calh` :



Serving HTML

So far in `i de j` we're calling `e e d` and pass it a HTML string. Our code would look very confusing if we just placed our entire application's HTML there. Instead, we're going to create a `i de h l` file and serve it.

Let's refactor our route handler to use `e dfile` instead:

```
a   ge      f  c i   e    e
   e   e dfile  i de  h  l
```

And populate `i de h l` with the following:

```
 d c   e h  l
h  l
  head
     i le S cke  IO cha    i le
       le
          a gi      addi g     b    i i g  b  de  b
```

```
      b d    f          Hel e ica  A ial
      f      backg    d         addi g          i i    fi ed  b          id h
      f    i       b  de       addi g         id h        a gi   igh
      f    b         id h      backg    d   gb                b  de      e   addi g
        e  age    li      le    e     e   a gi       addi g
        e  age  li    addi g
        e  age  li   h child  dd    backg    d   eee
         le
    head
   b d
     l id   e  age      l
    f    ac i
      i     id     a   c   le e   ff     b      Se d  b
     f
    b d
  h  l
```

If you restart the process (by hitting Control+C and running `de i de` again) and refresh the page it should look like this:

Integrating Socket.IO

Socket.IO is composed of two parts:

- A server that integrates with (or mounts on) the Node.JS HTTP Server: `cke i`
- A client library that loads on the browser side: `cke i clie`

During development, `cke i` serves the client automatically for us, as we'll see, so for now we only have to install one module:

```
    i  all   a e  cke i
```

That will install the module and add the dependency to `ackage j` . Now let's edit `i de j` to add it:

```
 a a     e i e e  e
 a h      e i e h     Se e a
 a i    e i e   cke i   h

a  ge     f c i   e   e
   e  e dfile i de h l


i     c  ec i    f c i    cke
  c   le l g a   e c  ec ed


h   li e      f c i
  c   le l g li e i g
```

Notice that I initialize a new instance of `cke i` by passing the `h` (the HTTP server) object. Then I listen on the `c ec i` event for incoming sockets, and I log it to the console.

Now in index.html I add the following snippet before the `b d` :

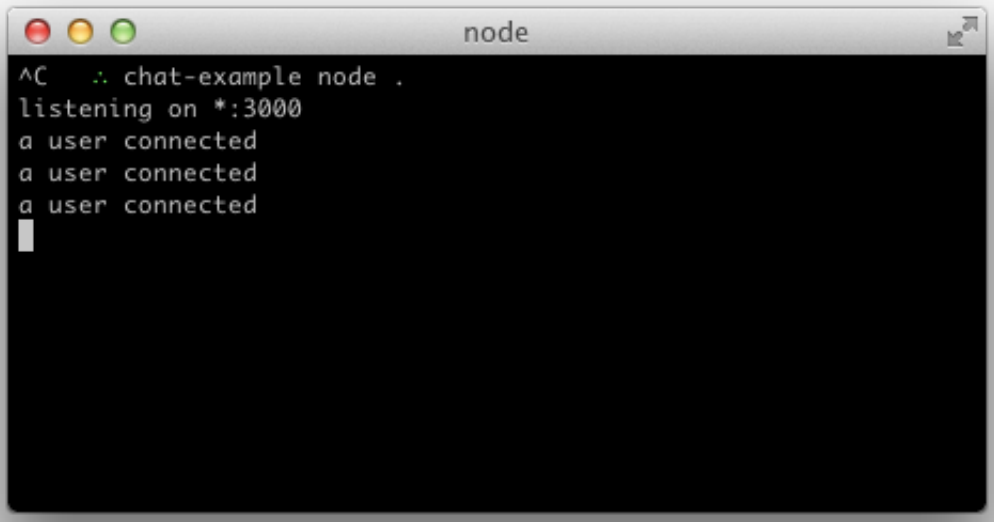
```
  c i    c    cke i   cke i j     c i
  c i
   a   cke   i
   c i
```

That's all it takes to load the `cke i clie` , which exposes a `i` global, and then connect.

Notice that I'm not specifying any URL when I call `i` , since it defaults to trying to connect to the host that serves the page.

If you now reload the server and the website you should see the console print "a user connected".

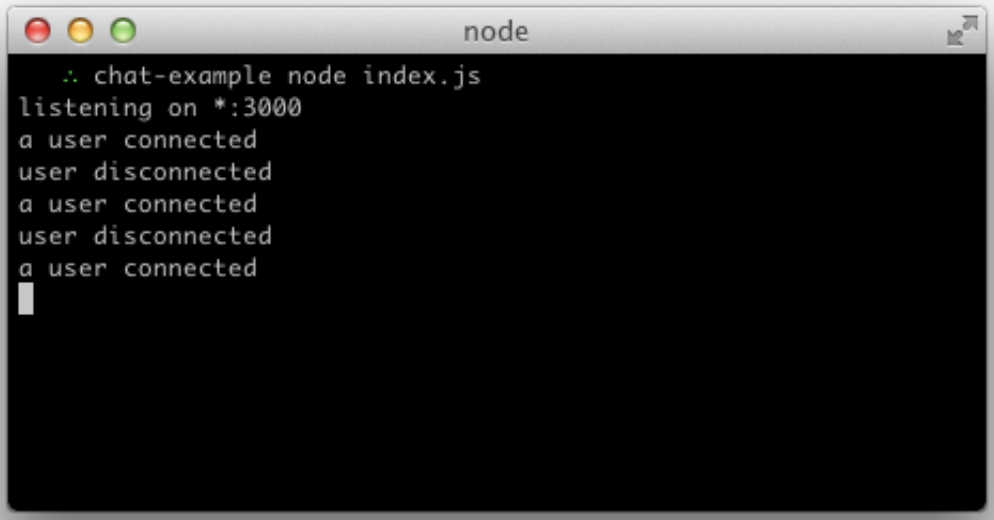
Try opening several tabs, and you'll see several messages:



Each socket also fires a special `di c ec` event:

```
i      c   ec i     f  c i     cke
  c    le l g  a   e  c   ec ed
    cke      di c   ec    f  c i
    c    le l g    e  di c   ec ed
```

Then if you refresh a tab several times you can see it in action:



Emitting events

The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too.

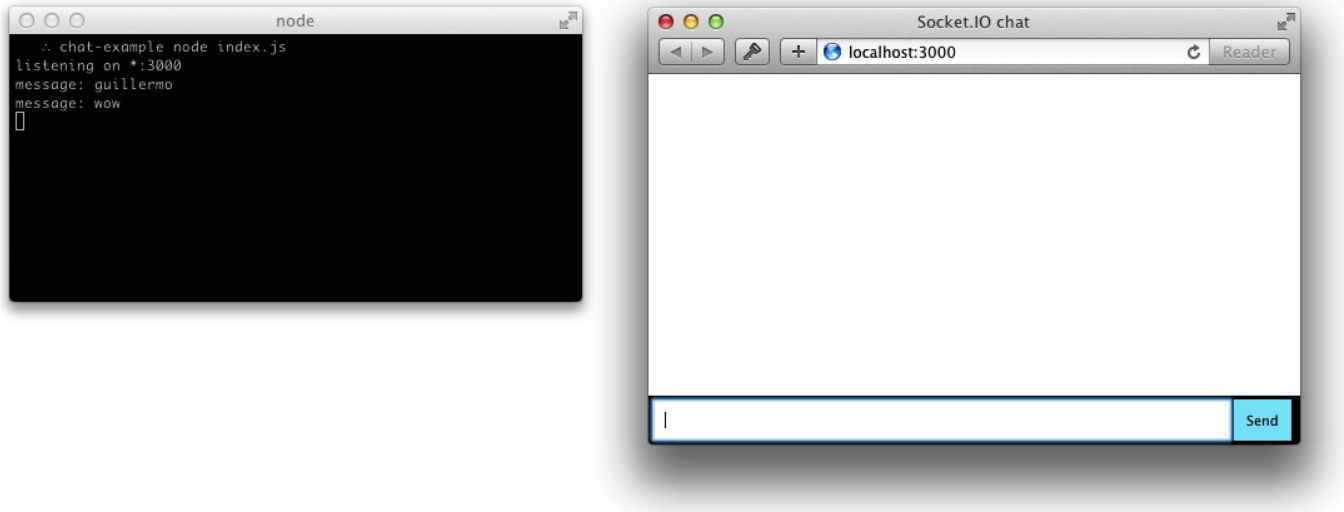
Let's make it so that when the user types in a message, the server gets it as a `cha e age` event. The `c i s` section in `i de h l` should now look as follows:

```
c i     c     cke  i    cke  i  j      c i
c i     c  h      c de j  e   c   j  e          j      c i
c i
 a    cke    i
   f        b i  f  c i
    cke  e i   cha   e  age           al
           al
   e     fal e


 c i
```

And in `i de j` we print out the `cha e age` event:

```
i      c   ec i     f  c i     cke
    cke      cha   e  age   f  c i     g
    c    le l g   e  age        g
```

The result should be like the following video:



Broadcasting

The next goal is for us to emit the event from the server to the rest of the users.

In order to send an event to everyone, Socket.IO gives us the `i e i` :

```
i  e i     e e e      f     e e    e
```

If you want to send a message to everyone except for a certain socket, we have the `b adca` flag:

```
i      c   ec i     f  c i     cke
    cke  b  adca   e i   hi
```

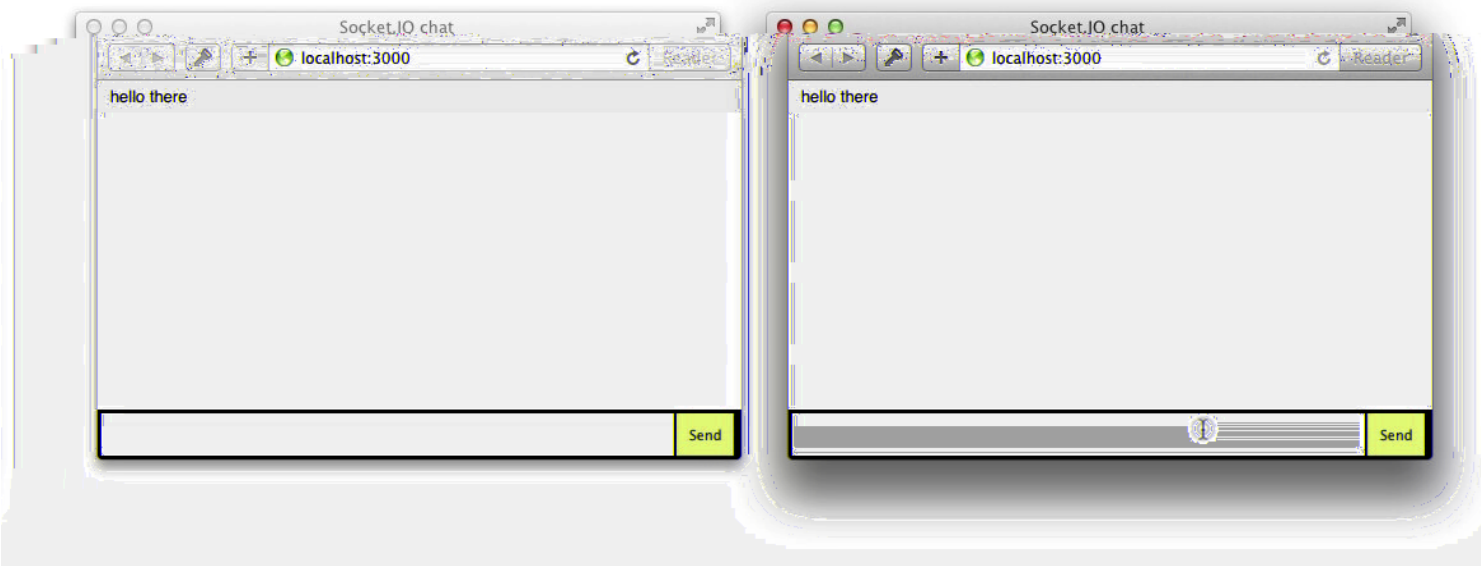
In this case, for the sake of simplicity we'll send the message to everyone, including the sender.

```
i      c   ec i     f  c i     cke
    cke      cha   e  age   f  c i     g
    i  e i   cha   e  age     g
```

And on the client side when we capture a `cha e age` event we'll include it in the page. The total client-side JavaScript code now amounts to:

```
c i
 a    cke    i
   f        b i  f  c i
    cke  e i   cha   e  age           al
           al
    e     fal e

   cke      cha   e  age   f  c i     g
        e  age    a  e d     li     e     g

   c i
```

And that completes our chat application, in about 20 lines of code! This is what it looks like:



Homework

Here are some ideas to improve the application:

- Broadcast a message to connected users when someone connects or disconnects
- Add support for nicknames
- Don't send the same message to the user that sent it himself. Instead, append the message directly as soon as he presses enter.
- Add "{user} is typing" functionality
- Show who's online
- Add private messaging
- Share your improvements!

Getting this example

You can find it on GitHub here.

```
gi  cl  e h      gi h b c   g ille cha  e a  le gi
```