An introduction into Express, Mongoose, and EJS templates.

All the above mentioned are frameworks that run on Node.js. So to get a basic understanding of what there do, let us build a simple application together. It will be all about displaying widgets, creating new widgets, editing and deleting existing widgets. But first things first let us first of all configure our development environment step by step.

A) Configuring our development environment

1) Installing Node.js

It's a good idea to talk a little bit about Node before installing it. Node.js is a server-side technology that's based on the Google's V8 Java-script engine. It's a Highly scalable system that uses asynchronous event-driven I/O(input/output) rather than threads or separate processes .Its ideal for web applications that are frequently accessed but computationally simple .And on like other web-severs like Apache for example that creates a separate thread or invokes a new process each time a request is made to the sever, Node on other hand doesn't create a new thread or processes, instead it listens for specific events and when events happens, responds accordingly. Node doesn't block any other request while waiting for the previous event to complete, and all these events are then processed in a first come first serve basis in an event loop. Node can be installed on unix ,linux ,windows, mac etc. N:B this tutorial is done on windows.

The installation on windows is relatively easy if one is used to download and install anything from the internet. All what one needs to do is to go to this side

http://nodejs.org/download/

and select the version that suites your operating system. then click on it and follow the setup guide. N:B there are also other methods of installations which are not discoursed here, one can as well read the documentation.

2) Mongdb(NOSQL)

This is our data-base ,it is a document oriented database we need it here to persist all our Widgets. This tutorial will be focusing more on the mongoose framework that maps Node.js with mongodb. But for more details on mongodb one can look on http://www.mongodb.org/ to get more insides. N:B Mongodb has a 2 GB memory limitation on a 32 bits system, so for developments that needs more data, one should think about 64 bits system.

To install mongodb, go to go to the link below and get the copy that suites your system.

http://www.mongodb.org/downloads

After downloading, extract the zip file to your c drive, that is it you are done with the installation Do the following to start your sever

i) Create a new folder(this is different from the folder you extracted the downloaded file to) in your c drive, I called mine mongodb.

ii) Open a dos prompt, change the directory to where you extracted the zip file In mine I have it like this In mine I have

Cd C:\mongodb-win32-i386-2.2.0

Now change to the bin folder of this directory, via the cd command like this Cd bin then you should have something like this in your command prompt

C:\mongodb-win32-i386-2.2.0\bin>

iii) While still in the above directory use the mongo.exe like this to start the sever,

mongod - - dpath "c:\mongodb"

remember mongodb is the empty folder you created in step (i), this folder most exist, if it doesn't exist then all the above will not work N:B you can use any folder name of your choice and repeat step (i) to iii every to start mongodb every time you want to use it. To stop mongo press ctrl c or strg c in German

All what is remaining now is for us to install the necessary framework. As we mentioned above, there are Express, Mongoose and the template engine EJS which all run on express

3) Express.Js

This is a light-weight web application framework, that helps to organize web app into MVC (model-view-architectures) on the server side, It enables us to use a variety of templating languages like EJS, JADE, DUSTJS etc It also allows us to use light-weight frameworks for databases like mongoose(mongodb), mysql, redis etc
With node.js installed on the system, one can install Express easily

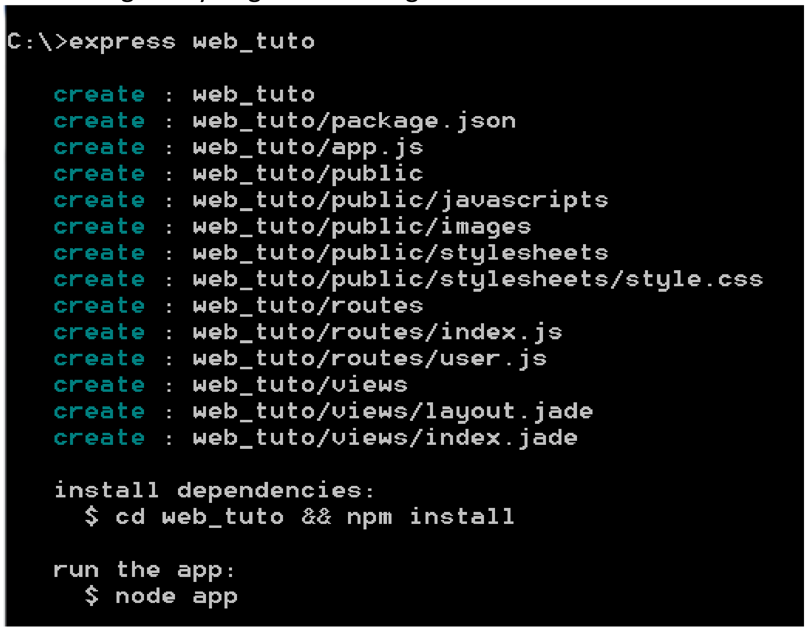
i) Go to the dos prompt, change to c(or any desired directory and type the following)

Npm install –g express

this command installs express globally ,one can then use the express binary module
to locally initialize an app like this

express web_tuto

web_tut being the name of the folder were my application will be created. After
after doing that you get something like this.



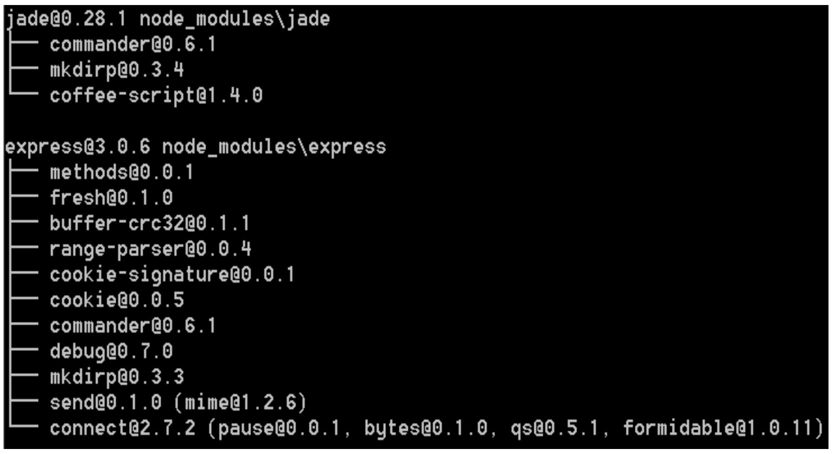
```
C:\>express web_tuto

   create : web_tuto
   create : web_tuto/package.json
   create : web_tuto/app.js
   create : web_tuto/public
   create : web_tuto/public/javascripts
   create : web_tuto/public/images
   create : web_tuto/public/stylesheets
   create : web_tuto/public/stylesheets/style.css
   create : web_tuto/routes
   create : web_tuto/routes/index.js
   create : web_tuto/routes/user.js
   create : web_tuto/views
   create : web_tuto/views/layout.jade
   create : web_tuto/views/index.jade

   install dependencies:
     $ cd web_tuto && npm install

   run the app:
     $ node app
```

Express creates a directory named web_tuto with a bunch of files in which you can base
Your application. You can then open the directory to see what express has created
First there is the package.json manifest file which contains something like this

```
  {
      "name": "application-name",
       "version": "0.0.1",
     "private": true,
     "scripts": {
        "start": "node app"
          },
      "dependencies": {
       "express": "3.0.6",
          "jade": "*"
           }
 }
```

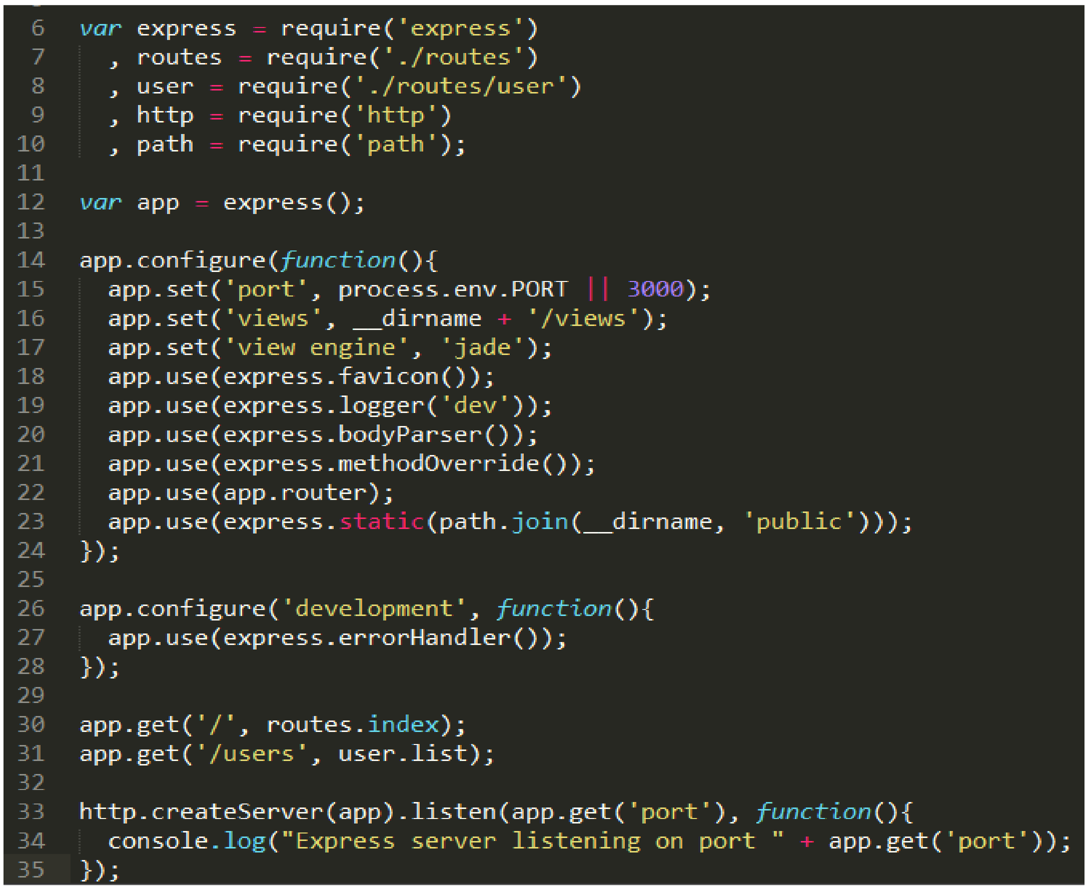
The point of focus here is the “dependencies” this manifest files’s main purpose is to contain the module dependency list npm can use to install dependencies. Lets install dependencies now.

C:\web_tuto> npm intall



```
jade@0.28.1 node_modules\jade
├── commander@0.6.1
├── mkdirp@0.3.4
└── coffee-script@1.4.0

express@3.0.6 node_modules\express
├── methods@0.0.1
├── fresh@0.1.0
├── buffer-crc32@0.1.1
├── range-parser@0.0.4
├── cookie-signature@0.0.1
├── cookie@0.0.5
├── commander@0.6.1
├── debug@0.7.0
├── mkdirp@0.3.3
├── send@0.1.0 (mime@1.2.6)
└── connect@2.7.2 (pause@0.0.1, bytes@0.1.0, qs@0.5.1, formidable@1.0.11)
```

The above code has downloaded and installed the express module and the jade templating language ,modules and their dependencies into the local node_moduelse directory. You can see that express depends on among others, connect module. Next, by exploring the public directory Express created for you, you can see that it contains directories for the static files you may need in your app. N:B This are somewhat standard directories for separting the locations of style sheets, images, and client-side Js files, but you can change them at will. You will see that express has also created a file called app.js with some initial code as shown below.



```
var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');

var app = express();

app.configure(function(){
  app.set('port', process.env.PORT || 3000);
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(path.join(__dirname, 'public')));
});

app.configure('development', function(){
  app.use(express.errorHandler());
});

app.get('/', routes.index);
app.get('/users', user.list);

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

Quite a bunch of things to talk about but let's just consume it like this for the moment and continue with the installation, then while developing our app, we will talk about some of the above.

4) <u>**Mongoose and Ejs**</u>

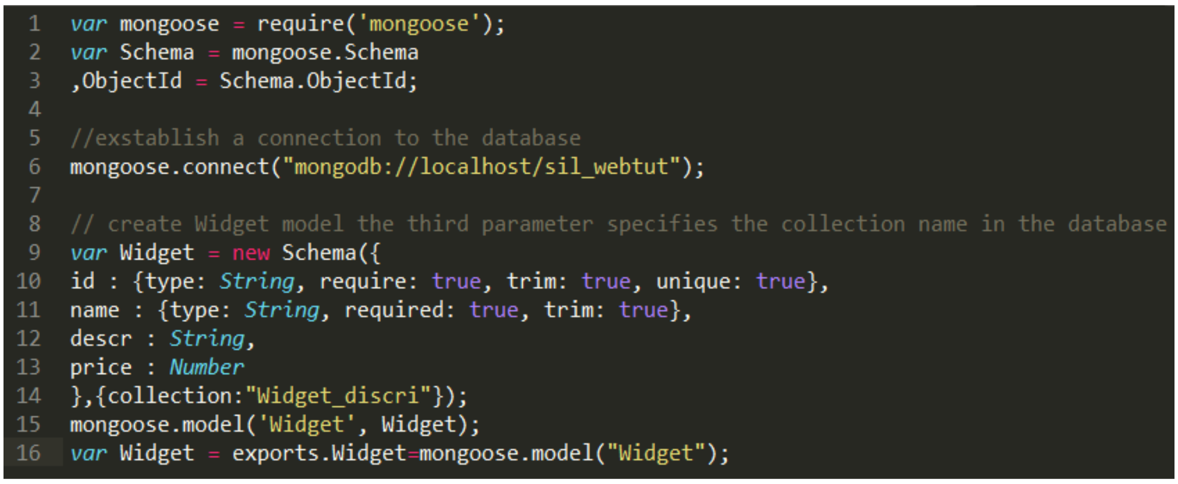
With express running, we can then install our html template dependency(ejs) as well as our mongodb(mongoose) mapper.
This is rather very easy, just change to our web_tuto directory and do the following
<u>**C:\web_tuto> npm install mongoose.**</u> For monogdb
<u>**C:\webt_tuto> npm install ejs.**</u> For html template N:B if you open the node_modeles folder, you will see all these depencies we have installed.

B) Application development

At this point our system is configured and now ready for development, so we can now go ahead and start developing our application, Just to remind you of what we will be developing. This will be an app that is capable of displaying widgets stored on our mongodb database, editing , deleting existing widgets and creating new widgets.

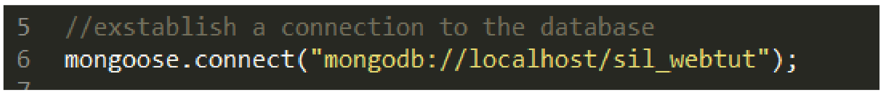
i) Defining our document in mongoose

Let us start by describing our widget in mongoose, so that we have a clearer view on the type of data that we have, before proceeding to manipulate them. Still under the web_tuto directory, I created a file called widget.js which looks like this



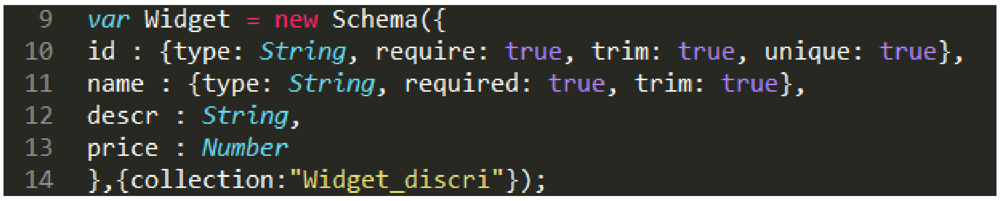
```
var mongoose = require('mongoose');
var Schema = mongoose.Schema
,ObjectId = Schema.ObjectId;

//exstablish a connection to the database
mongoose.connect("mongodb://localhost/sil_webtut");

// create Widget model the third parameter specifies the collection name in the database
var Widget = new Schema({
id : {type: String, require: true, trim: true, unique: true},
name : {type: String, required: true, trim: true},
descr : String,
price : Number
},{collection:"Widget_discri"});
mongoose.model('Widget', Widget);
var Widget = exports.Widget=mongoose.model("Widget");
```

Lets talk a little bit about what the above code is doing, the main goal here is to define a document schema and synchronize it to the database using the mongoose model. We use the key word require to get any file or directory we want to make use of. By default require looks for mongoose in the node_modules folder, so if you change the path, then you need to give a correct directory name. Any document in Mongodb has a unique object id, **shema.ObjectId** defines it for us, then we make a connection to the database, like this



```
//exstablish a connection to the database
mongoose.connect("mongodb://localhost/sil_webtut");
```

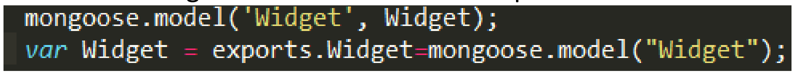
Remember, mongodb is the empty folder we created before, and since we are developing locally, we specify localhost else one needs to specify the correct server when developing remotely, sil_webtut is the database we will be connecting , if it doesn't exist mongoose will create it for us so no call for alarm here. Now we use the mongoose schema to define our documents like this



```
var Widget = new Schema({
id : {type: String, require: true, trim: true, unique: true},
name : {type: String, required: true, trim: true},
descr : String,
price : Number
},{collection:"Widget_discri"});
```

Type: string, define the type of data we are dealing with here string,
Require: true , is just like specifying not null in a relational database. If we create a new Widget with a null on this field, we get an error.
Trim: true, only removes white spaces from a field
Unique: true, indicates that this id can only exist once that means no duplicates, if we Duplicate them, an error will be thrown.
Number: can hold any integers, floats, etc
Collection:"widget_discri", all documents will be stored under this collection name if it is not The collection will still be created and mongoose will look for any default name from within Your document and assign it as the collection name.

We can then register the collection and export it like this



```
mongoose.model('Widget', Widget);
var Widget = exports.Widget=mongoose.model("Widget");
```

Mongoose.model: registers our collection as a mongdb document and
The second line only exports it so that we can then require it in other files and do most of the Things that mongodb can do. That is all we need for now, this as well demonstrates the Power of mongoose, one can even modify the schema and add more complex data types like Arrays, embedded documents etc but this is the very basics of mongose.

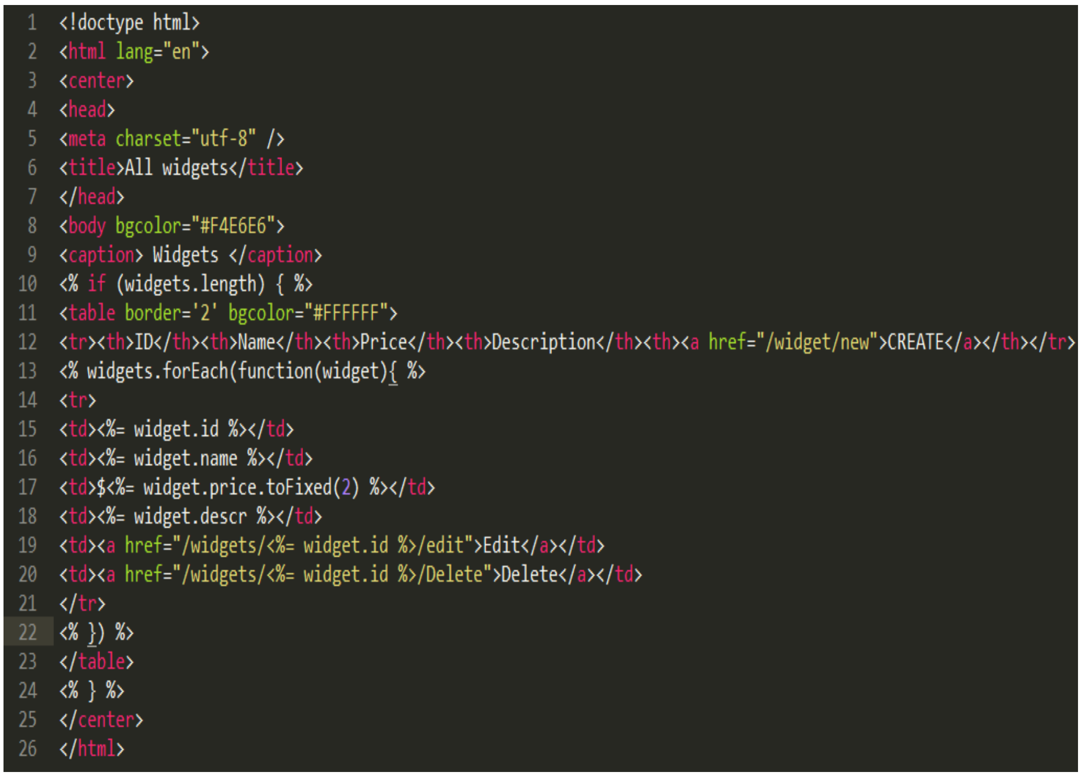
ii) creating the html pages in ejs

Now that we know how our data looks like, lets go ahead and create all the pages we need, We will need the following pages,

1) Allwidgts.html: this is to display all the widgets we have in the database
2) Delete.html: for deleting widgets
3) Edit.html: for editing existing widgets
4) Added.html: for displaying newly added widgets
5) New.html: for creating new widgets

All this pages will be manage by our ejs template engine

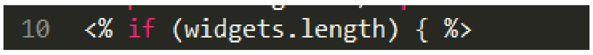
1) Allwidgts.html



```html
<!doctype html>
<html lang="en">
<center>
<head>
<meta charset="utf-8" />
<title>All widgets</title>
</head>
<body bgcolor="#F4E6E6">
<caption> Widgets </caption>
<% if (widgets.length) { %>
<table border='2' bgcolor="#FFFFFF">
<tr><th>ID</th><th>Name</th><th>Price</th><th>Description</th><th><a href="/widget/new">CREATE</a></th></tr>
<% widgets.forEach(function(widget){ %>
<tr>
<td><%= widget.id %></td>
<td><%= widget.name %></td>
<td>$<%= widget.price.toFixed(2) %></td>
<td><%= widget.descr %></td>
<td><a href="/widgets/<%= widget.id %>/edit">Edit</a></td>
<td><a href="/widgets/<%= widget.id %>/Delete">Delete</a></td>
</tr>
<% }) %>
</table>
<% } %>
</center>
</html>
```

Lets do much of the ejs talking on this page, the other pages will contain basically most of this functionalities,
It is good to node that ejs is actually embedded into normal html here, which gives you the power of still using your html know how.
The page above mainly creates an html table but with some strange syntax,

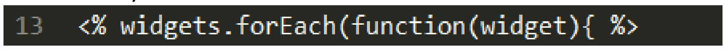
a) If statement



```
<% if (widgets.length) { %>
```

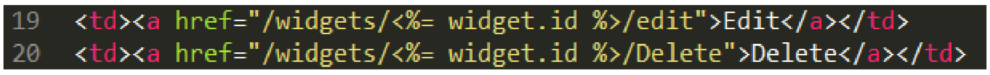
Don't be puzzled with were widgets is coming from, it is actually passed from the backend where were render this page, we can then use it here just like a normal json array to do what so ever we want the <% { %> can be used to de-clear a statement in ejs then condition for this statement can then be written after the %> and when done, one should close the statement this way <% } %> this holds for almost all the statements. In this case we wanted to create a table so we are just checking to make sure that the we have data to display on the table.

b) forEach statement



```
<% widgets.forEach(function(widget){ %>
```

In short to make use of all what is coming from the back end, we must use it in greater
Than and less than signs like this <% %> here we just want to loop across all the
Widgets in the database and display all their respective fields. As seen in our mongoose
Section above, we know how the field names look like so we just do the widgets. To get all
The fields and just at the end of each widget we make a link to the delete and edit page
Like this



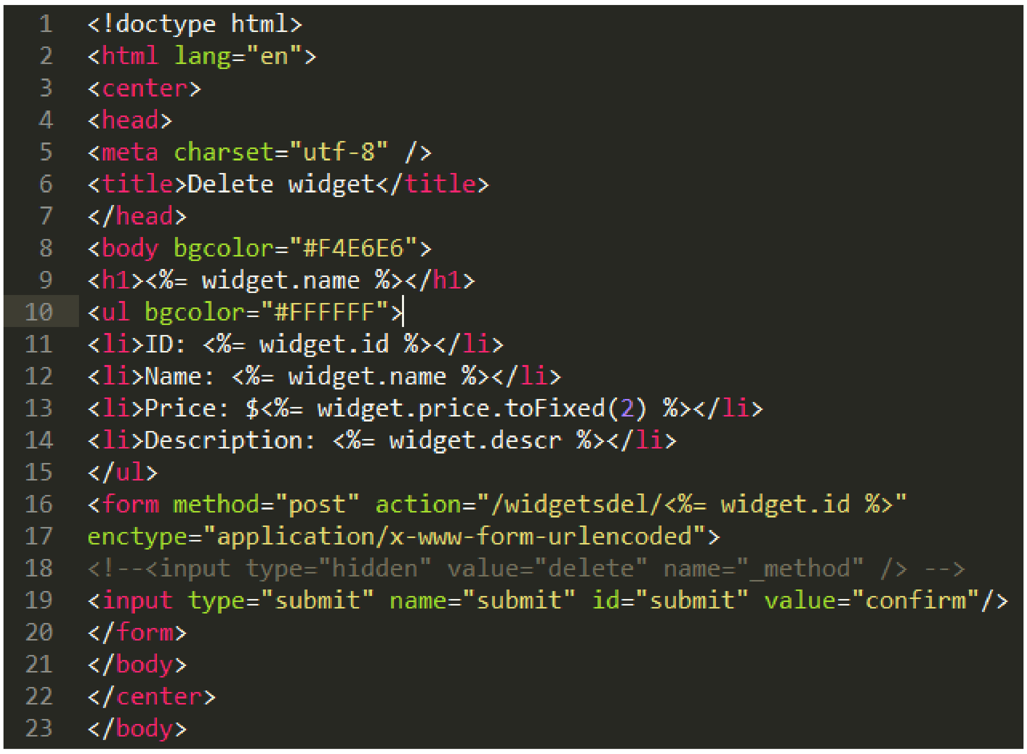
```html
<td><a href="/widgets/<%= widget.id %>/edit">Edit</a></td>
<td><a href="/widgets/<%= widget.id %>/Delete">Delete</a></td>
```

So when we click Edit, this url `/widgets/<%= widget.id %>/edit"` is sent to the server, same holds for Delete with the following url
`"/widgets/<%= widget.id %>/Delete"`

The point to take note of here is the <%=widget.id%> this will read the id of the widget and pass It as a number to the server so in reality we have something like /widgets/1/delete or /widgets/1/edit.

`<a href="/widget/new">CREATE</a>` will only display the new widget page.

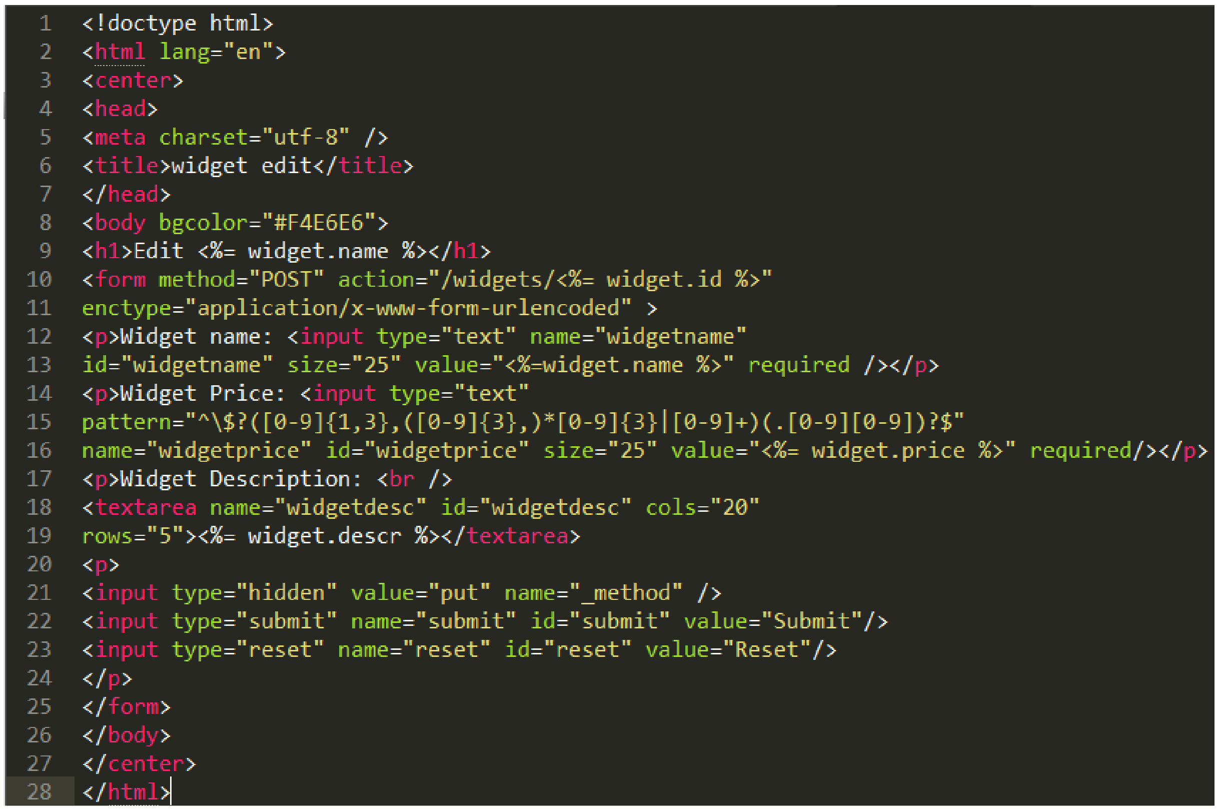
2) Delete.html



```
<!doctype html>
<html lang="en">
<center>
<head>
<meta charset="utf-8" />
<title>Delete widget</title>
</head>
<body bgcolor="#F4E6E6">
<h1><%= widget.name %></h1>
<ul bgcolor="#FFFFFF">
<li>ID: <%= widget.id %></li>
<li>Name: <%= widget.name %></li>
<li>Price: $<%= widget.price.toFixed(2) %></li>
<li>Description: <%= widget.descr %></li>
</ul>
<form method="post" action="/widgetsdel/<%= widget.id %>"
enctype="application/x-www-form-urlencoded">
<!--<input type="hidden" value="delete" name="_method" /> -->
<input type="submit" name="submit" id="submit" value="confirm"/>
</form>
</body>
</center>
</body>
```

This page just displays what we want to delete as an unordered list so that we look at it, confirm before deleting, because it is actually deleted from the database. Notice the <%=%> Syntax that displays the widget fields. After displaying these, we created a form that post a this url `/widgetsdel/<%= widget.id %` to the server, Note again that the url will actually be post like this /widgetsdel/1, 1 being the id of what we want to delete, this is the intelligent bit here, because we will actually use the number to know the document we are deleting from the database. NB I put a name for the title but one can also specify it while rendering the page and use   or famouse <%=title%> synthax to read the title and display it.

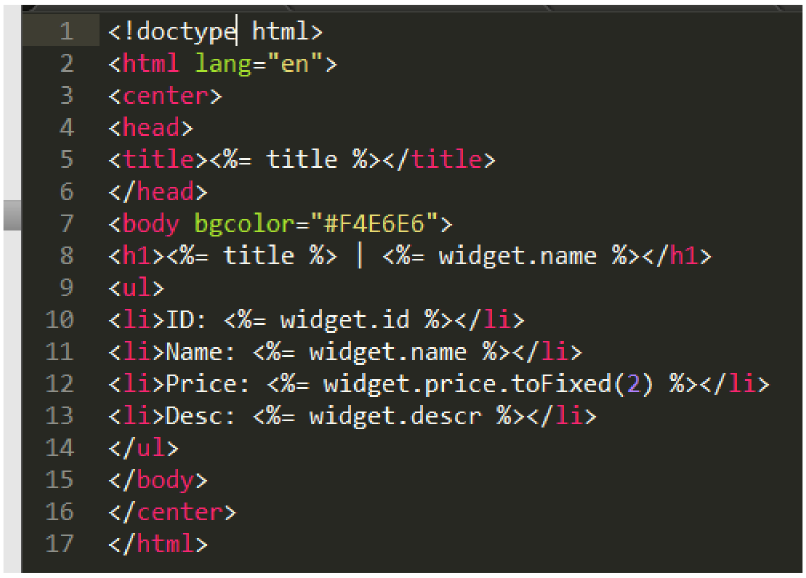
3) Edit.html



```
<!doctype html>
<html lang="en">
<center>
<head>
<meta charset="utf-8" />
<title>widget edit</title>
</head>
<body bgcolor="#F4E6E6">
<h1>Edit <%= widget.name %></h1>
<form method="POST" action="/widgets/<%= widget.id %>"
enctype="application/x-www-form-urlencoded" >
<p>Widget name: <input type="text" name="widgetname"
id="widgetname" size="25" value="<%=widget.name %>" required /></p>
<p>Widget Price: <input type="text"
pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
name="widgetprice" id="widgetprice" size="25" value="<%= widget.price %>" required/></p>
<p>Widget Description: <br />
<textarea name="widgetdesc" id="widgetdesc" cols="20"
rows="5"><%= widget.descr %></textarea>
<p>
<input type="hidden" value="put" name="_method" />
<input type="submit" name="submit" id="submit" value="Submit"/>
<input type="reset" name="reset" id="reset" value="Reset"/>
</p>
</form>
</body>
</center>
</html>
```

If you observe the code very well, you see that the ejs stuff is basically the same as before. There are just two things then left to comment on which are; `pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"` this is the pattern to display price with the dollar symbol infront of it and `<input type="hidden" value="put" name="_method" />` here I will use the app.put to get this url in the server.

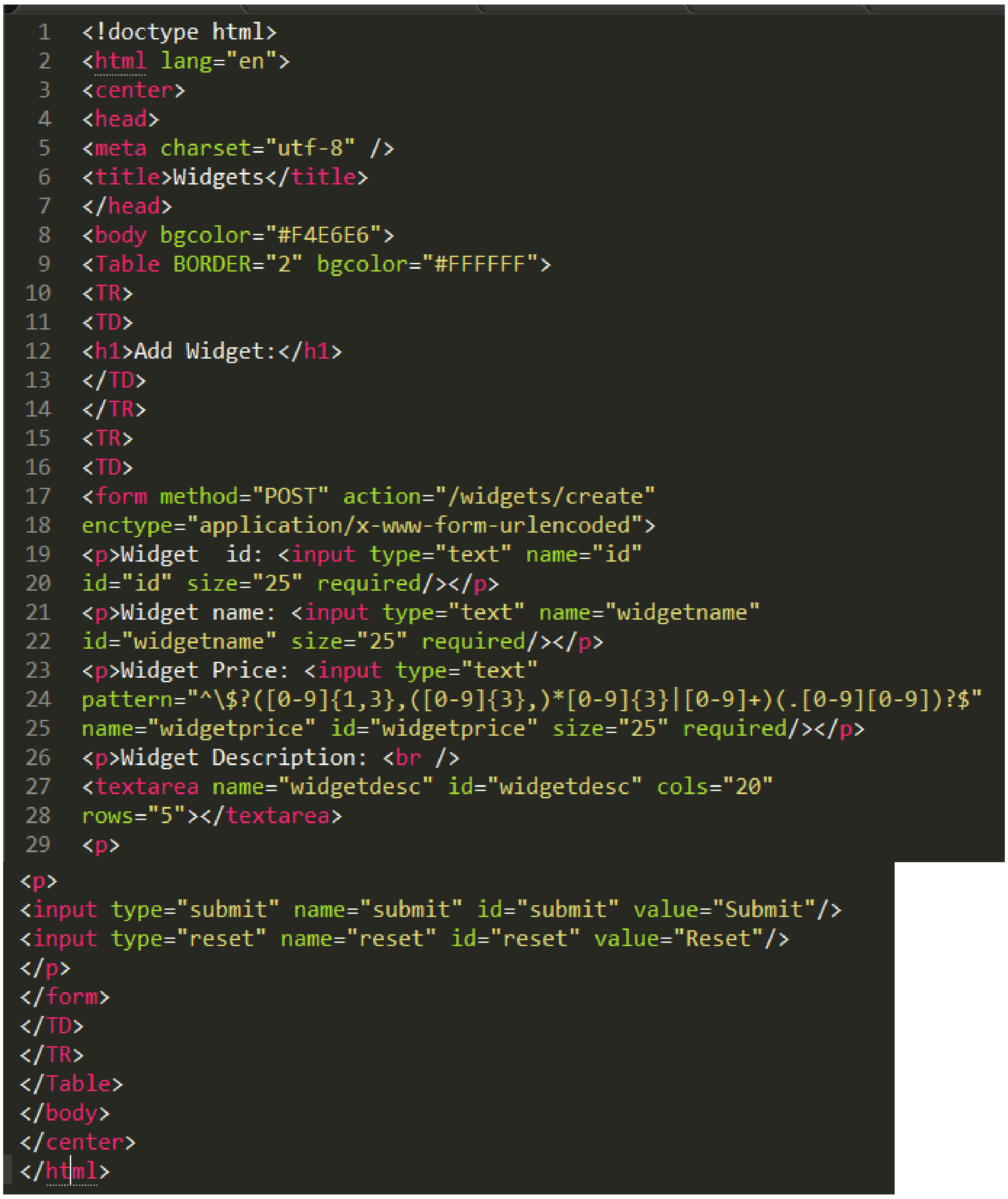
4) Added.html



```
<!doctype html>
<html lang="en">
<center>
<head>
<title><%= title %></title>
</head>
<body bgcolor="#F4E6E6">
<h1><%= title %> | <%= widget.name %></h1>
<ul>
<li>ID: <%= widget.id %></li>
<li>Name: <%= widget.name %></li>
<li>Price: <%= widget.price.toFixed(2) %></li>
<li>Desc: <%= widget.descr %></li>
</ul>
</body>
</center>
</html>
```

What this page does is just to show the newly added widget , but as fare as ejs is concern, we have just one new thing to take node of here, that is any thing outside <%%> is displayed as in html. And finally

5) New.html



```html
<!doctype html>
<html lang="en">
<center>
<head>
<meta charset="utf-8" />
<title>Widgets</title>
</head>
<body bgcolor="#F4E6E6">
<Table BORDER="2" bgcolor="#FFFFFF">
<TR>
<TD>
<h1>Add Widget:</h1>
</TD>
</TR>
<TR>
<TD>
<form method="POST" action="/widgets/create"
enctype="application/x-www-form-urlencoded">
<p>Widget  id: <input type="text" name="id"
id="id" size="25" required/></p>
<p>Widget name: <input type="text" name="widgetname"
id="widgetname" size="25" required/></p>
<p>Widget Price: <input type="text"
pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
name="widgetprice" id="widgetprice" size="25" required/></p>
<p>Widget Description: <br />
<textarea name="widgetdesc" id="widgetdesc" cols="20"
rows="5"></textarea>
<p>
<p>
<input type="submit" name="submit" id="submit" value="Submit"/>
<input type="reset" name="reset" id="reset" value="Reset"/>
</p>
</form>
</TD>
</TR>
</Table>
</body>
</center>
</html>
```

Almost thesame thing as we have seen with other pages just that here we create a new widget.
N:B all thise pages are stored in our views directory in the web_tuto folder, this is very important because if you use another diretory, you will have to specify the full path name as we go on.

III) developing the server backend with express

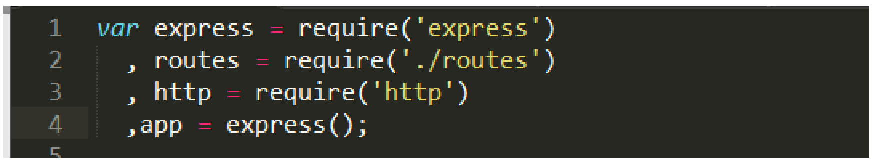
Now that our pages and data are well defined, its time for us to go ahead and develop the backend logic of this application using the Express frame work, The most impornt thing about this frame work is the aspect of routing, This means that one can send urls to the server and then route to the specif code or file that is designed to do something with that particular url. Now rember we promise to leave all what we saw in the app.js file and talk about later, lets look into some of the things that express generated for us.

Exploring the default generated code

Now if we look kindly at the code in the app.js file we will notice that express generated some middleware components for us. Just before we proceed, if you go to a dos prompt, then change to your project directory and type **node app.js** or **node app ,** this will start the application and if you open the browser and type **localhost:3000** in your url, you will see

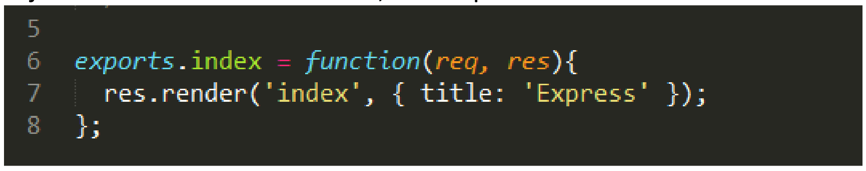
Welcome to Express

Lets get look in to some of the code to understand what is happening.



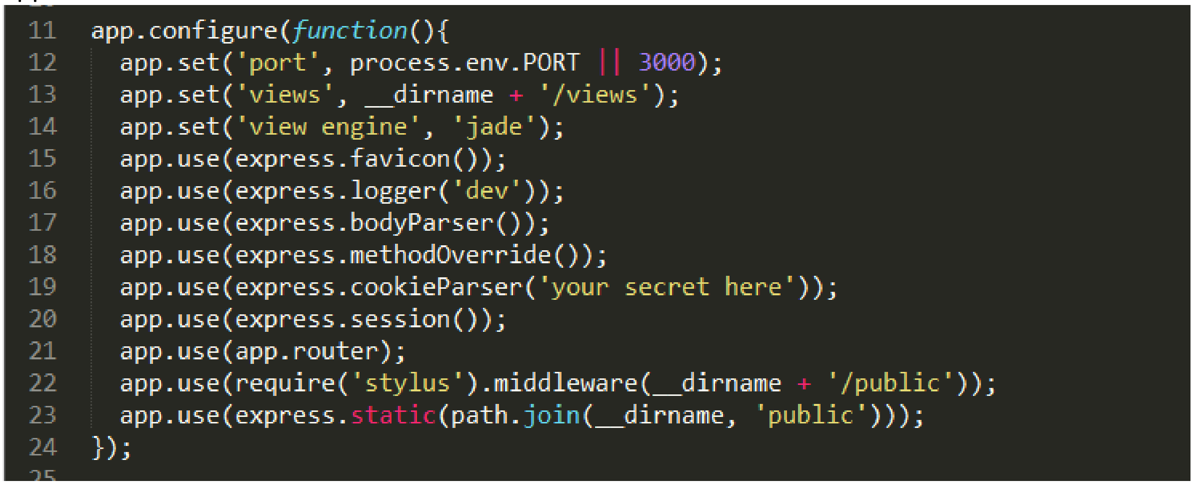
```
var express = require('express')
  , routes = require('./routes')
  , http = require('http')
  ,app = express();
```

First of all, to make use of any thing in Node.js we need to require it , in analogy to object oriented programming, this is just like getting and object and making use of its methods, fields etc , so here we require express from the node_modules folder(this is done by default you don't have to specify the full path name), then the node http also from the same folder and routes just references a file called index.js this file is our default route, if we open it we see this



```

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

So basically its routes to this file and renders a page called index.jade in the views directory passing it a title, by default the index.jade page is a jade template so we can change it. So the **welcome to Express** that is displayed is gets the **title Express** from here and the **welcome to** from the index.jade page. Next we get our express object, this can then be use to build our app. Let us just show what app does and talk about some of the stuff.

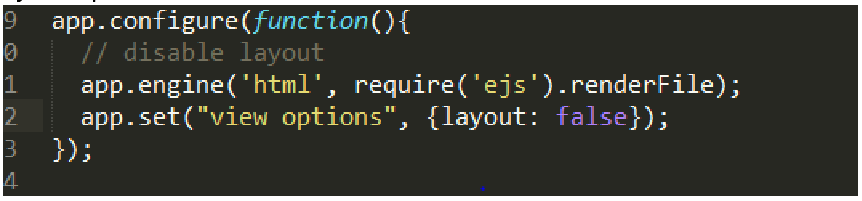


```
11  app.configure(function(){
12    app.set('port', process.env.PORT || 3000);
13    app.set('views', __dirname + '/views');
14    app.set('view engine', 'jade');
15    app.use(express.favicon());
16    app.use(express.logger('dev'));
17    app.use(express.bodyParser());
18    app.use(express.methodOverride());
19    app.use(express.cookieParser('your secret here'));
20    app.use(express.session());
21    app.use(app.router);
22    app.use(require('stylus').middleware(__dirname + '/public'));
23    app.use(express.static(path.join(__dirname, 'public')));
24  });
```

11) We configure our middlware(what is between you and the system) with a call back function
12) we set the port that our server will be listening to, by default it sets 3000, but one can change it and set any desired number.
13) We set the default views, views are were all the html templates be it jade, ejs or normal html pages are saved. N:B if we have a page that is set in another directory, then we need to specify the full pathname.
14) The default views engine i.e jade is set in our case we will be using ejs so we see how to configure that as well.
17) This middleware parses the incoming request body ,converting it into request object Properties.
19) This gets all the parses all the cookies from the browser.

21) contains all the default routes and performs lookup for any given route
This basic info is good to go with as far as the middleware is concerned, what we can do now is just to define out ejs template middleware and create our server with the http we got from node.js

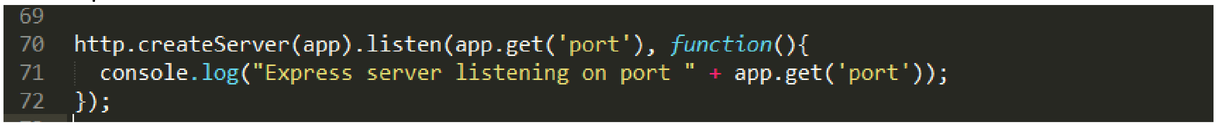
1) Ejs template



```
app.configure(function(){
  // disable layout
  app.engine('html', require('ejs').renderFile);
  app.set("view options", {layout: false});
});
```

App.egine sets a new template engine(ejs) for rendering html files and
App.set overwrites the default egine(jade) that was created by express.

2) Our http server



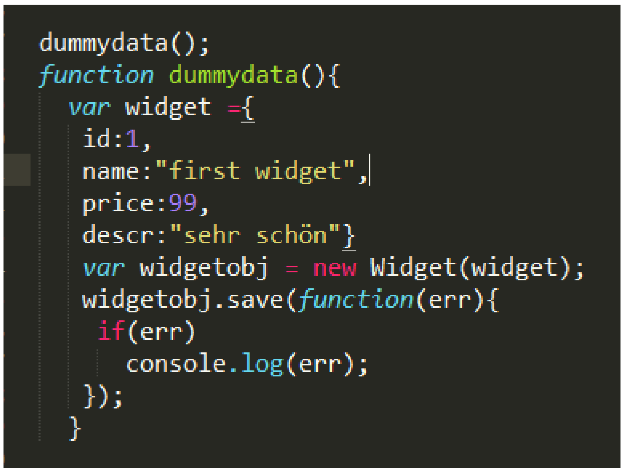
```
http.createServer(app).listen(app.get('port'), function(){
  console.log("Express server listening on port " + app.get('port'));
});
```

This is basically using the createServer method from http to create a server that listens to all the express stuff that we configured with app, then provides a port that the sever listens on and a call back function.

At this point we can now start can start coding our backend and making use of all the framworks(mongoose, ejs etc) to code our widget. For the views, all the pages are store with .html extension in the views directory of our project folder. In the router directory we will be making use of the index.js file this is where most of the work will be done there is also a file in the project directory called widget.js this is were our database stuff is defined as we did before if you forget you can look on mongoose above.

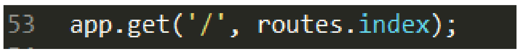
A detailed look in to the routing logic

The very first thing we want to do is to display all the widgets we have in our database, when we start the program, this very page also have create, edit and delete links which can be clicked on to do all do what their names say on widgets. It's a good idea that we have something in the database else we get an error, remember that your database needs to be running if not look into mongodb above to start your database so I created a function called dummydata that puts one widget in our database this function looks like this



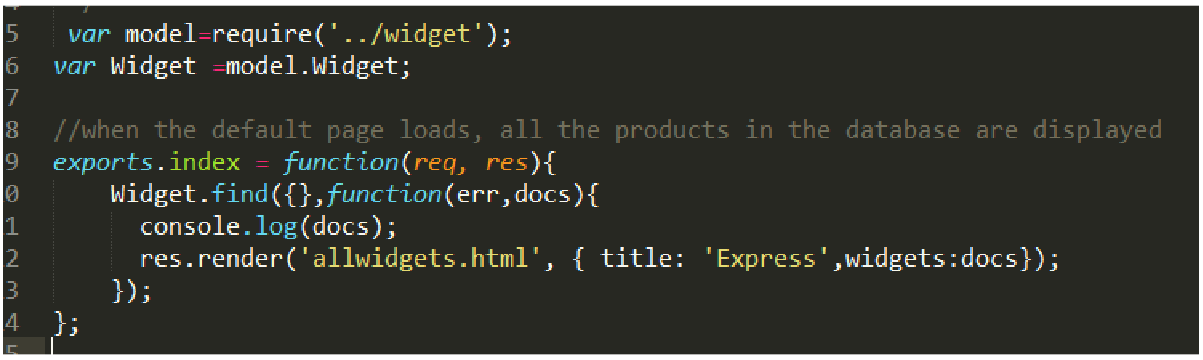
```
dummydata();
function dummydata(){
  var widget ={
   id:1,
   name:"first widget",
   price:99,
   descr:"sehr schön"}
   var widgetobj = new Widget(widget);
   widgetobj.save(function(err){
    if(err)
      console.log(err);
   });
  }
```

Don't worry about the code for the moment, the same logic will be use to in more detail so I will be discussing it. So if we start the application, our **allwidgets** page will load with at least something to display. If we look into our app.js file we see things like app.get, app.post, app.put etc what all these does is to get a request from the url and do a specific task, this is the basis our routing in express.js, lets look at all of them one by one**. N:B please call the dummy function only once when you start the app, if you want to restart the app comment out the function call so not to get errors due to uniqe key properties in the database the function itself is intended just to have some thing to start with. Widgets can be created from the front end when the app starts running.**

1) 

```
app.get('/', routes.index);
```

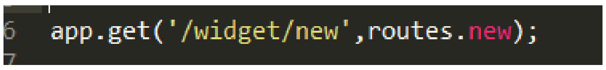
When we type localhost:3000 in the url of our browser, the default request url that is send to the server is '/' we can then use app.get to get that url . The first parameter in app.get is the url and the second is actually a function(call back function), you don't believe me? Wait and see instead of routes.index, one could have written function(req,res){……..}); and it will still work, I have provided that scenario also to demonstrate this. What routes.index does is that it goes into the index.js file located in the routes folder , looks for a function called index and executes it. So we will be looking in to the index file as well for each case to better understand what is happening. That means each time I provide an app.(get, post, put etc) I will write the corresponding route handler in index.js



```
 var model=require('../widget');
var Widget =model.Widget;

//when the default page loads, all the products in the database are displayed
exports.index = function(req, res){
    Widget.find({},function(err,docs){
      console.log(docs);
      res.render('allwidgets.html', { title: 'Express',widgets:docs});
    });
};
```

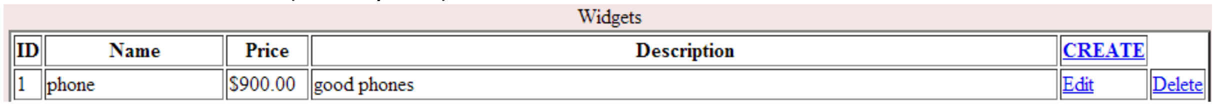
In the first line we require our widget, remember that the file widget.js contains all the logic to connect to the database, create our data(widget) etc, so we import it here via the keyword **require** then get the widget definition via model.widget, with this we can use Widget as a CRUD directly on our database.

The key word **exports.index** actually exports a callback-function called index, this function has parameters **req** and **res.** Req is an instances of **http.ServerRequest** while **res** is an instance of **http.ServerResponse.** we then use the mongodb keyword find, it can take many parameters but in this case we give it no parameter as first argument and a call back function as second, what Widget.find does is that it goes to the mongodb database and returns all the documents, is there is an error, the error massage will be passed to the first parameter of the call back function(err) if not the document return will be passed to docs(remember you can name err and docs the way you want) console.log(docs) only prints the contain of the document in the console, we then use **res.render** to display the **allwidgets.html** page render looks for the page by default in the views directory if you don't have it there, specify the full path name. The most interesting thing here is that we can then pas a title, and what find return to the page it self. Remember in the ejs templates that we talked about? If you look into it, you see that we did some thing like **<%=widgets.id%>** and so many others. This is actually were where we got it from, so now all our widgets have been displayed.

2) 

```
app.get('/widget/new',routes.new);
```

In the first page we displayed above, if we open it via localhost:3000 we see something like this, N:b this depends on the names you gave to your widget in the function(dummydata) above



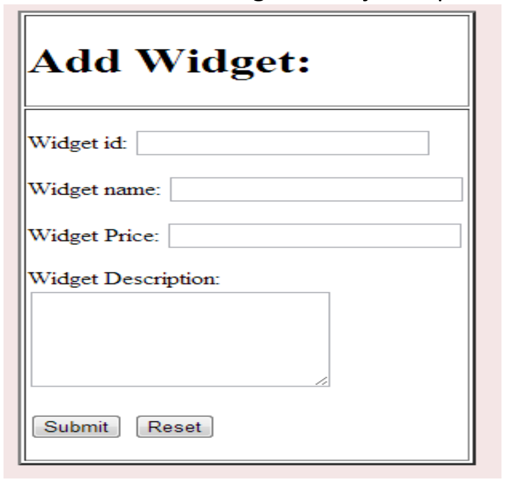
Widgets

ID	Name	Price	Description	CREATE	
1	phone	$900.00	good phones	Edit	Delete

When we click on create, the url **/widgets/new** is sent to our server and we use app.get as in (1). App.get now looks into the index file in the routes directory and execute the new function which looks like this

```
exports.new=function(req,res){

 res.render('new.html');

}
```

There is no big deal here,exports, req,res and render have already be described so what this function deos is just to display the new.html page which is stored in the views directory. We created this page when we were talking about ejs template. The form looks like this in the browser



All we need to do is to enter the id(no two ids should be the same else we get an error), name price and description of our widget, clicking on reset will clear all the fields and all the fields most be enter before the form can be summited. Now when we click on submit, the following url is send to our server /widgets/create we cann then use receive it from the server side like this

```
app.post('/widgets/create',routes.create);
```

This is because in new.html, we define the to submit the url with post like this method=”post” get could have been used as well. Again as before this routes us to the following function located in our index.js file found in the routes folder

```
exports.create = function(req, res) {
var widget = {
id : req.body.id,
name : req.body.widgetname,
price : parseFloat(req.body.widgetprice),
descr: req.body.widgetdesc};
var widgetObj = new Widget(widget);
widgetObj.save(function(err, data) {
if (err) {
res.send(err);
} else {
console.log(data);
res.render('added.html', {title: 'Widget Added', widget: data});
}
});
};
```

Now if you remember very well, to make use of anything in node.js we must require it, so that is why we required our widgets that describes our database functionality in this page its just something basic like this

```
  */
  var model=require('../widget');
 var Widget =model.Widget;
```

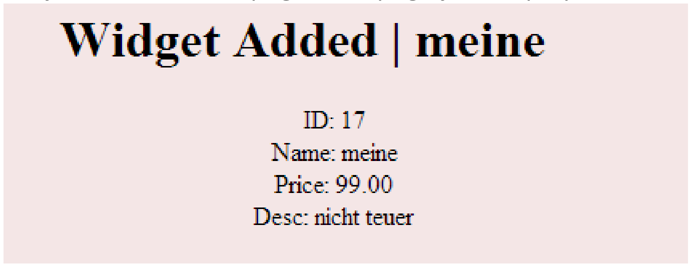
Our point of focused here is the widget creation. Exports,req, res and res.render performs same functions as described in (1) above. We created a json data called widget, notice that the name of the fields are the same as those described in the mongoose widget we created before. Then we use req.body.id etc to read the data that we are sending to the server into the json fields. This is how you read field values posted with post method in node.js. After doing that we register it to our widget via

```
var widgetObj = new Widget(widget);
```

Up to this point, nothing has been save to the database, this is only done when we call the save function on the widgetobj like this

```
widgetObj.save(function(err, data) {
if (err) {
res.send(err);
} else {
console.log(data);
res.render('added.html', {title: 'Widget Added', widget: data});
}
});
};
```

This is were it will attempt to save the document into our database, if one of the criteria fails, and err will be returned to the call back function. We can also send the error back to the browser via res.send else we just print the data we created in our console. Now the render method will display a page called added.html remember we created this before when we were talking about ejs templates, if you forget, you can still look into it to see what the page was displaying again we passed that data we just saved to the page. This page just displays something like this


Widget Added | meine

ID: 17
Name: meine
Price: 99.00
Desc: nicht teuer

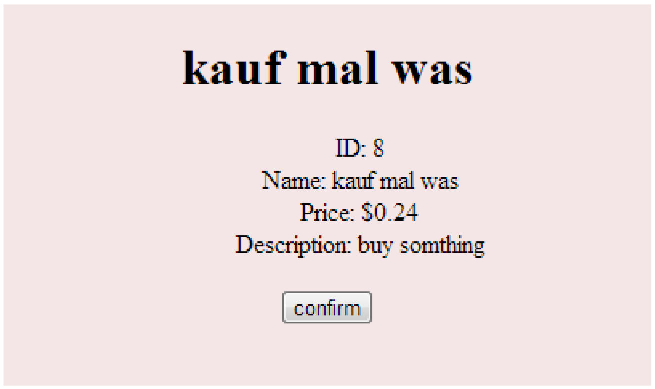
3)

```
app.get('/widgets/:id/delete',routes.del);
```

Again this happens when we click on the delete link next to each widget, but now something happens we are passing the id of the widget in the url, this is done in node.js with :id or what ever name you called your parameter this will actually send something like /widgets/1/delete to the database supposed the id is one, again our del function is called from index.js located in routes

```
// serach and dislplay the delete confim page
exports.del=function(req,res){
    var n =parseInt(req.params.id);
    //query the database for the widget containing the id
     Widget.findOne({id:parseInt(req.params.id)},function(err, doc){
     if(err)
     {
         res.send("No Widget with id"+" "+ n)
     }
     else
     res.render("delete.html",{widget:doc});
 });
 }
```

We have already see much of this, we have just some few things to talk of, that is the widgt.findOne, on like find, findOne only looks for one document in the database, we are using parseInt(req.param.id) to read the id that we passed in the url the parseInt converts it to a number(integer), so if something goese wrong we send a polite massage to the browser else we render the delete.html page passing it the document we got from the database. The delete page only display the information about widget you want to delete so you confirm it before deleting. It looks like this


kauf mal was

ID: 8
Name: kauf mal was
Price: $0.24
Description: buy somthing

confirm

So clicking on confirm will pass this url to the server /widgetsdel/:id remember id will be passed as a number to the server which is received via

```
app.post('/widgetsdel/:id',routes.destroy);
```

No big deal here i believe you can guess what is happening?, yes destroy function is called again from index.js

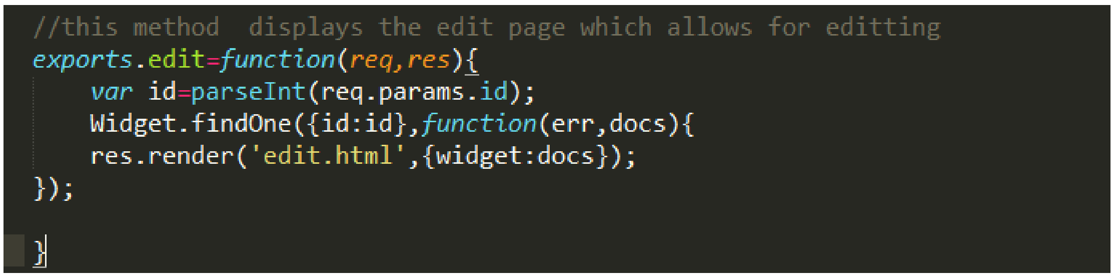
```
exports.destroy=function(req,res){
    Widget.remove({id:parseInt(req.params.id)},function(err){
        if(err)
         res.send(err);
        else{
            res.send('Widget with id'+" "+parseInt(req.params.id) +" "+"deleted");
        }
    });
}
```

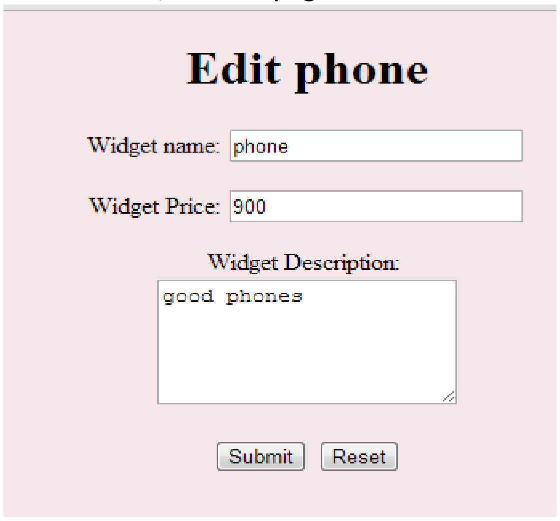
Widget.remove deletes a document completely from the database res.send here is sending an error to the browser is something goes wrong or sending a massage to confirm that the widget has been deleted.

3) Finally

```
app.get('/widgets/:id/edit',routes.edit);
```

Nothing much to talk about , lets just look at the edit function


```
//this method  displays the edit page which allows for editting
exports.edit=function(req,res){
    var id=parseInt(req.params.id);
    Widget.findOne({id:id},function(err,docs){
    res.render('edit.html',{widget:docs});
});

}
```
Here what we need to take node of is that we are looking for a particular document with findOne that is why we bas it the id field, this will look for a document with that particular id then we render the edit page with the document(docs) returned from the database, the edit page looks like this



Edit phone

Widget name: phone

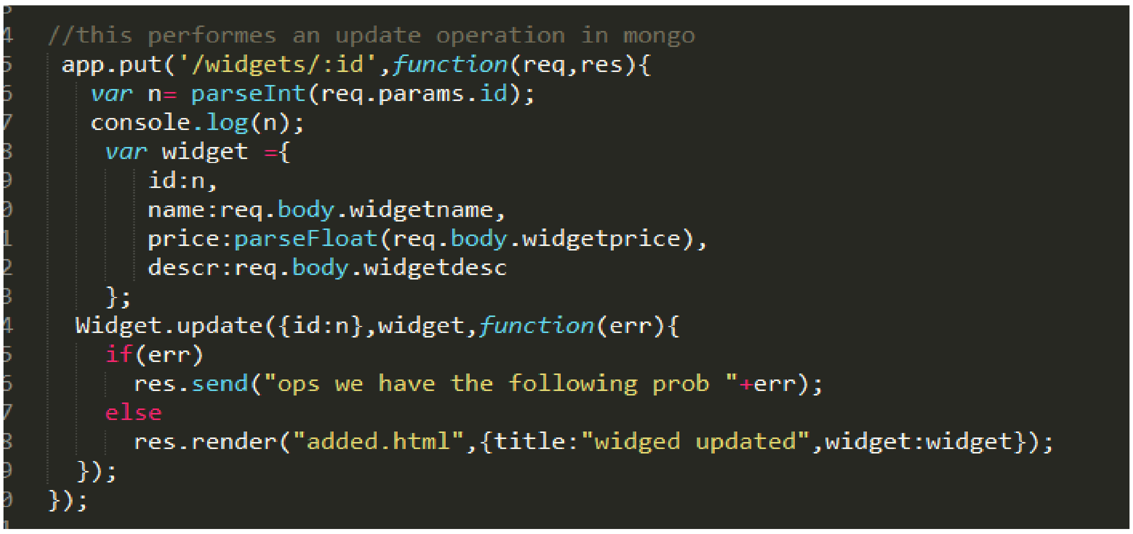
Widget Price: 900

Widget Description:

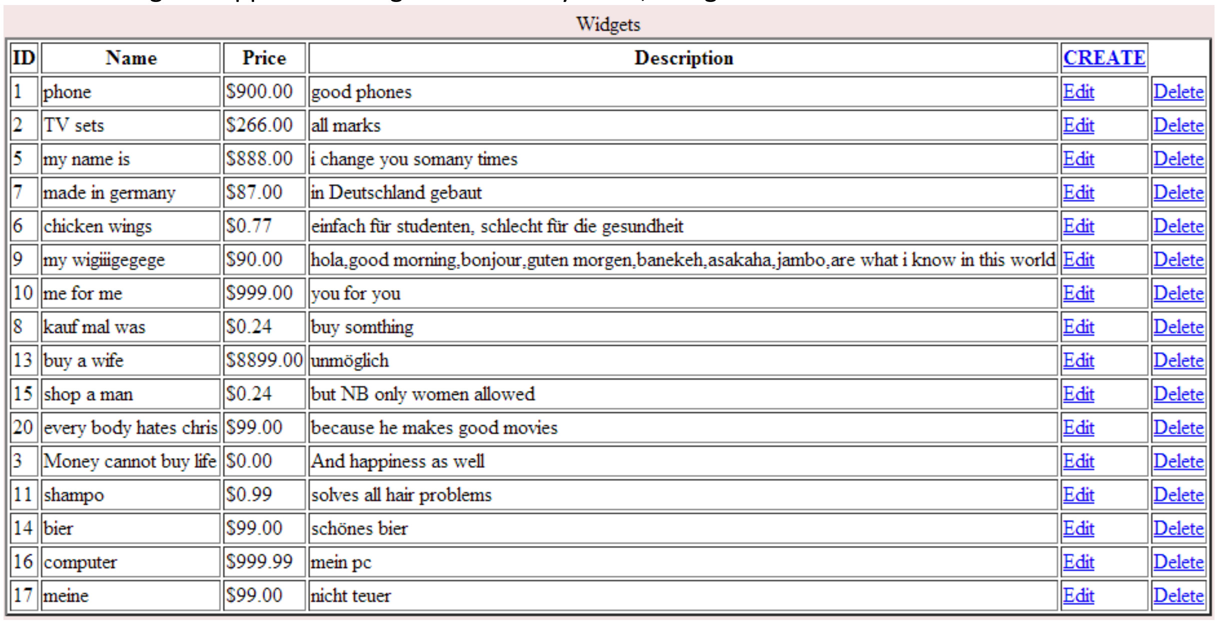
good phones

One can then change this parameters and click on submit. Which will sent this url to the database

/widgets/:id nothing new here right? But something has happened we are not receiving it with get or post again this time we are using put. Put is also an html verb that is usually used especially when one is doing update operations, and this time around I didn't call the handler function from the index.js file I wrote it as a function directly in the put verb and it looks like this


```
//this performes an update operation in mongo
app.put('/widgets/:id',function(req,res){
    var n= parseInt(req.params.id);
    console.log(n);
     var widget ={
        id:n,
        name:req.body.widgetname,
        price:parseFloat(req.body.widgetprice),
        descr:req.body.widgetdesc
     };
   Widget.update({id:n},widget,function(err){
     if(err)
       res.send("ops we have the following prob "+err);
     else
       res.render("added.html",{title:"widged updated",widget:widget});
   });
 });
```
The code looks almost like that which was used to create a new widget, the only difference is that we are now using widget.update. Update will perform an update operation in the database, two parameters plus our normal call back function are passed to it, these are id or the widget, and the widget json data which is then used

in the database to change the data which was already existing there, then we render our added page to show that something new was added if all goes well else the error massage will be displayed.N:B your database most be running else you get into trouble look at mongodb above if you don't remember how to start the database.
After running this app and adding some dummy stuff , we get this results



Widgets

ID	Name	Price	Description	CREATE	
1	phone	$900.00	good phones	Edit	Delete
2	TV sets	$266.00	all marks	Edit	Delete
5	my name is	$888.00	i change you somany times	Edit	Delete
7	made in germany	$87.00	in Deutschland gebaut	Edit	Delete
6	chicken wings	$0.77	einfach für studenten, schlecht für die gesundheit	Edit	Delete
9	my wigiiigegege	$90.00	hola,good morning,bonjour,guten morgen,banekeh,asakaha,jambo,are what i know in this world	Edit	Delete
10	me for me	$999.00	you for you	Edit	Delete
8	kauf mal was	$0.24	buy somthing	Edit	Delete
13	buy a wife	$8899.00	unmöglich	Edit	Delete
15	shop a man	$0.24	but NB only women allowed	Edit	Delete
20	every body hates chris	$99.00	because he makes good movies	Edit	Delete
3	Money cannot buy life	$0.00	And happiness as well	Edit	Delete
11	shampo	$0.99	solves all hair problems	Edit	Delete
14	bier	$99.00	schönes bier	Edit	Delete
16	computer	$999.99	mein pc	Edit	Delete
17	meine	$99.00	nicht teuer	Edit	Delete

End of tutorial
Hope this piece of work helped you thanks for reading. Remember man is not perfect if you see a bug somewhere try to fix it.