# From Almost Zero to a Start Project


*When you create a web-project, using the Express Framework, it will generate a number of files and folders, which at first glance can be difficult to grasp. This exercise will create a minimal Express application, and then; step-by-step add changes, so you will end up with a project similar to the one you get using the Expresses-Generator.*
*The important thing in this exercise (as in all exercises) is NOT to complete it as quickly as possible, but to understand each line generated in an Express Start Application.*
*Finally this exercise will guide you through the steps, necessary to deploy the project up to OpenShift*

1) Install Express like:

- `NPM install express –g`
- `NPM install express-generator -g`

Create a new folder (somewhere easy to locate), and in that folder run: `express firstExpress.`

Investigate the generated content

In the following we will create a similar project, starting with an absolute minimum solution and then add stuff, step by step, until we end with, and understand the start project Express can generate.

While you complete the steps below, you should repeatedly return to this project, created by Express, since this will serve as a reference for almost everything we will do.

This will be the only time you will create a project like this. When this exercise is completed, you should understand and use the template files created by Express Generator

**2 - Create a new project in your Favorite editor(empty project)**
Create a new project expressDemo, using the empty project option.

**3 - Create the project folders.**

Take a look at the directories generated by Express in step 1. We need our project to mirror this design, so add the following folders to the root of your project:

- bin
- node_modules (do not add this folder, it will be generated)
- public
  - images
  - javascripts
  - stylesheets
- routes
- views

**4 - Create the project files**
In the project created by Express in step-1, a number of files were generated. In this step we will focus only on the files generated in the root folder.
The package.json file will, as we have seen before, be generated by NPM, so just create a javaScript file **app.js** (by convention, the name used in an Express application) and add the following content:

```
var express = require('express');
var app = express();
app.get('/', function(req, res){
  res.send('hello world');
});
app.listen(3000);
```

**5 - Create the package.json file:**

Open the terminal, navigate to the root of your project (if not already there) and do the following:

**`npm init`** → Provide a name for the project and your name as author. Select defaults for all the rest
Verify that the file *package.json* has been generated.
Run: **`npm install express --save`**
Verify that a folder *node_modules*, including the Express project, has been created and that the dependencies section in *package.json* has been updated.

Add this to *package.json* (if it already include a scripts block, just add the content)

```
"scripts": {
    "start": "node ./app"
  },
```

Verify that you can start the server via a: **npm start**.

**6 - Starting the server**

There are several problems with the way we just started the server, with the hardcoded port number as the most serious one.
We will start the server the "Express way".

Navigate to the *bin* folder in the project created in step-1 and copy the file in this folder (*www*) into the *bin* folder of our new project.
This file relies on the external module debug, so install this with: `NPM install debug –save`

The file also relies on our *app* file (`require('../app')`) so we need to do three things:

1. Remove the hardcoded `app.listen(..)` in app.js
2. Export our app by adding this line as the last line in *app.js*: `module.exports= app;`
3. Change the start script in package.json into: `"start": "node ./bin/www"`

Verify that you can start the server using `npm start`, or with the debug variable set:

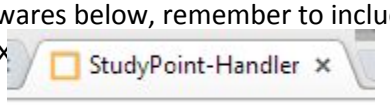
```
set DEBUG=YOUR_DEBUG_NAME:* & npm start
```

This way of providing the port number, via the `process.env.PORT` variable is required by many hosting platforms.

For additional information about the debug module, see: https://github.com/visionmedia/debug + http://expressjs.com/guide/debugging.html

**Adding Middleware**
Remember that "order matters" when it comes to middleware. So be sure to insert lines in the following exactly as stated (or better, use the app.js from part-1 as your guide)

For most of the middlewares below, remember to include the necessary package using **npm**:
`npm install xxxxxx`




**7 - Adding a favicon**

Add the following require statement to the start of your project:

```
var favicon = require('serve-favicon');
var path = require('path');
```

Add this right after the `app = express()` line:

```
app.use(favicon(path.join(__dirname, 'public','images','favicon.ico')));
```

`Make sure you understand the purpose/return-value of the path.join(..) call.`

Get a favicon from one on the online sites, for example: http://www.favicon-generator.org/ and test. Don't spend a lot of time on this. Just download a favicon from the list of existing favicons.

**8 - Logging requests**

Logging is an extremely import part of any professional server.
Start the server and a browser pointing to http://localhost:3000. Press F5 a few times and notice that nothing is written in the server-console.

Add the following *require* statement to the start of your project:
```
var logger = require('morgan');
```

Add this line right after line that uses the favicon middleware:
```
app.use(logger('dev'));
```

Restart the server and observe the log messages now are reported.

By default these messages are written to STDOUT, but you can easily write the messages to a file, and also change the way things are logged.
See: https://www.npmjs.org/package/morgan for details.

**9 - Error Handling:**
Start the server and access a non-existing address via your browser: http://localhost:3000/IDoNotExist

Express will report, that this page does not exist, but in most cases (always) we would like to determine how errors are reported.
a)
Add a very last middleware to your pipeline (just above the `module.exports..` line) which should be a "catch all" handler for any request that doesn't match any other routes, as sketched below:

```
app.use(function (req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);  //Make sure you understand this line
});
```

Restart the server and test it with the non-existing URL.
b)
This looks even worse than what we had before. Luckily however, we have the middleware pipeline, so we can add our own user defined middleware after the error handler (because of the `next(err)` line).
Add the following lines after the error handler and test it with the non-existing URL.

```
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function (err, req, res, next) {
    res.status(err.status || 500);
    res.send("<h1>Sorry there was a problem: " + err.message + "</h1><p>" + err.stack + "</p>");

  });
}
//Will not print stacktrace
app.use(function (err, req, res, next) {
  res.status(err.status || 500);
  res.send("<h1>Sorry there was a problem: " + err.message + "</h1><p>" + err.message + "</p>");
```

```
});
```

c)
To see an exception message, add the following line to the route method (`app.get....`):
`throw new Error("UPPS");`

To see an (production) example without the stack trace, start the server via **npm** as sketched below:
`set NODE_ENV=production & npm start`

For most hosting platforms, this variable will be provided by the platform.

**10 - Handling Post requests and parameters**

The route methods in the following are not provided by the Express template, but they will show you how to use the *body-parser* included in the template.
Add the following content (observe that GET and POST are served by the same path):

```
var names = [];
app.get('/form', function (req, res) {
  res.send("Hi: "+names.join(",")+"<form method='post'><input name='name'></form>");
});

app.post('/form', function (req, res) {
  names.push(req.body.name);
  res.redirect('/form');
});
```

Enter a name and press return. This should course an error since Express really ships with minimum capabilities out of the box. We need some middleware to read POST arguments.

Add the following require statement to the start of your project:
`var bodyParser = require('body-parser');`

Add this, right after the line that uses the Logger middleware:

```
// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }))
```

Restart, and enter a few names via the browser to verify that we can access the POST parameters.

**11 - Handling JSON POST data.**

We can also handle incoming data in JSON-format.
Add this line after the previous `app.use(...)`

`app.use(bodyParser.json());`

And this under the existing POST route handler:

```
app.post('/names', function (req, res) {
  names.push(req.body); //We receive it as a JavaScript object
  console.log(JSON.stringify(req.body));
  res.redirect('/form');
});
```

Test this route, using either **Postman** or via an additional node-app, where you perform a programmatic POST Request, using the this JSON-data `["peter","Ole","John"]`. Remember to set the *Content-Type* to *application/json*.

**12 - Using cookies**

Using the cookie-parser module, we can easily use cookies in our project.
We will not do this in this exercise, but add the following the right places (see project generated in part-1) since our goal is to create a project similar to what was generated by Express.

```
var cookieParser = require('cookie-parser');
```
and
```
app.use(cookieParser());
```

Use https://www.npmjs.org/package/cookie-parser for information of use.

**13 - Serving Static Content**

All content we have returned up until now has been dynamically generated. Static files can however be easily served as sketched below:

Add this right after line that uses the cookie-parser middleware:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Observe that this is a middleware (the only one) that ships directly with Express.

Create a dummy html-file in the public folder and verify that the server can serve this static file.

**14 - Templates and the Model-View-Controller Pattern**

Up until now, we have used the response-objects, `send` method whenever we sent something back to the client.
This is obviously not the way to go for real projects. Here, we will use the MVC pattern, which in Express is implemented via a *view-engine* and *views* written in a Domain Specific Language, understood by the selected view-engine.

If you take a look at the project generated in part-1, you will find three **.jade* files in the *views* folder. This is because Jade is the default view-engine for Express, so Jade was included and the files generated when we generated the project with `express firstExpress.`

Select the three files (in the *views* folder in the project generated in part-1) and copy the files into the *views* folder of our project.

The file *layout.jade* creates a skeleton html-file as sketched below when it gets compiled.
It is not meant to be used by itself. The other two files "extends" this file and provides the actual content for the content part.

| Pug (previously Jade) | Generated HTML |
|---|---|
| `doctype html`<br>`html`<br>`  head`<br>`  title= title`<br>`  link(rel='stylesheet',`<br>`ref='/stylesheets/style.css')`<br>`  body`<br>`  block content` | `<!DOCTYPE html>`<br>`<html>`<br>`  <head>`<br>`  <title>Express</title>`<br>`  <link rel="stylesheet"`<br>`  href="/stylesheets/style.css">`<br>`  </head>`<br>`  <body>`<br>`  content`<br>`  </body>` |

| | |
|---|---|
| | </html> |

The file *index.jade* will provide the content for our default route handler. The #{title} will display the JavaScript title value we provide to the script.

| Pug (Jade) | Generated HTML |
|---|---|
| extends layout<br><br>block content<br>  h1= title<br>  p Welcome to #{title} | <h1>Express</h1><br> (Value of title) |

The file *error.jade* will provide the content for our error messages.

| Pug (Jade) | Generated HTML (for a not found) |
|---|---|
| extends layout<br><br>block content<br>  h1= message<br>  h2= error.status<br>  pre #{error.stack} | <h1>Not Found</h1><br><h2>404</h2><pre>Error: Not Found<br>...<br></pre> |

Now, let us use the templates ☺

a)
Install Jade, the usual way: `npm install jade –save`
To simplify the use of Jade, we need to set the view folder and "tell" express which (Jade) view engine to use.

Add this code, just after; var app = express():

```
app.set('views', path.join(__dirname, 'views'));//Actually the default view folder
app.set('view engine', 'jade');//allow us to leave out the extension
```

Now locate the route function `app.get('/', function( req, res){...`
Replace the `res.send(..)` method with:

```
res.render('index', { title: 'Express' });
```

This will call the *index.jade* template in the views folder, and pass in the JSON object with the *title* used by the template.

<u>In this simple example the MVC components are as follows:</u>

- The view → index.jade
- The model → { title: 'Express' }
- The Controller → app.js (The app.get("/"...) method)

In part 15, you will see how we can isolate the controller, using the Router object.

b)
Change the two error handlers in the bottom of app.js to use the *index.jade* template. If you are in doubt, use the project generated in part-1 for help.

c)
Create a new template which should be used by the `app.get("/form"...)` route method.

Replace the `res.send(..)` method in the handler with a `res.render(..)` that should use the new template and pass in the required data.

**15 – Using the Router object**

A router is an isolated instance of middleware and routes. Routers can be thought of as "mini" applications, capable only of performing middleware and routing functions. Every express application has a built-in app router.

One of the benefits we get from this object is the ability to create our routes as separate modules, without any need for the Express app-instance.

Locate the file *index.js* in the routes folder of the project generated in part-1. And copy it into the corresponding folder in our project.
The content is very simple, it:

- Gets the router instance
- It performs a get(..) on the router
- It exports the router

Require the module in the require section:

```
var routes = require('./routes/index');
```

Locate the `app.get("/", ...)` function in *app.js*, uncomment it, and insert the following instead:

```
app.use('/', routes);
```

Restart the server, and verify that everything works as expected.

If you feel like, "OK and so what is does exactly the same as before", you are excused ;-)

But this is only because the example was simple.
Imagine for an example, a REST API with all the four CRUD operations.
Here we can create this in a separate module and control the URL as we like, when we use it (try changing the path in the `app.use('/',routes)` statement).

This is it. Now you should have an understanding of the basics of an Express application and be able to use the skeleton project created by Express or WebStorm.

Complete the next (last) step to see what is necessary in order to deploy the application to OpenShift.

**16 – Deploy the Server to OpenShift**

TBD