Start

- I am yujian from NUS Hackers, and I will conduct this workshop on web development
- Make sure you all have install them
- do so while I talk
- helpers around
- will be using sublime text for our editor
- facilitators around, feel free to raise your hand if you have questions
- for this workshop, simple pre-reqs

Intro

- This workshop is about express, a node.js framework
- And we will build a blog today with it
- Things I will cover are
  - the model of the web
    - Basically a very simplified model of what happens when you surf the web
  - then I will teach a bit of javascript since today almost every code will be written in javascript
    - show of hand how many alr know javascript
  - Then Nodejs, the server-side platform
    - writing server in javascript
  - Then I will introduce how we handle urls
    - So that you can visit different pages
  - introduce a template language call ejs
    - will cover what templates are and the rationale of using them
  - introduce database
    - no-sql, mongodb
    - ok if you don't know sql
  - Lastly, user sessions
    - how to determine if a user is logged in

Model of the web

- here's an over simplified model
- For our blog today
- the server will be written with the express framework
- so express will handle the request and response
- So how does the server actually respond
- First we have many routes
  - routes are just urls
- so the server looks at the url and match it with routes

  - it also gets information from the url
  - based on that, 1 retrieve information from database
  - with the server side information, goes to 2
  - make use of the information and the templates, generates html
  - Templates describes how the page should present the info
  - with the generated html, now goes to 3, send the html to the browser

- so this is mvc
  - mvc stands for model-view-control
  - it means a separation of concerns
  - models contains only the data
  - view contains the way data gets shown
  - and controller controls how the application behaves by using the model and view
  - here databse is the model, it does. . . .

- So this is how we are going to implemnt our blog,
  - model-view-control
  - databse-templates-controller

javascript

- let's start with the language we will be using
- I will just quickly run through it, since this is not a workshop on javascript
- we will do hands on for express
- first off, very important feature, output or logging
- we use this function console.log
- then variables
- In javascript, you do not need to declare types, so just var things, var stands for variable
- so as shown here, you can have numbers, strings, booleans as the basic types
- then for nothingness, you can use null
- and we can form an array of variables, with the bracket
- then one more useful data type is object
  - it is like a dictionary, which allows you to point a string of property name to some other variable
  - look at it like your C struct, but you are free to add new properties in runtime!
- lastly, one thing to note is that arrays can contain variable of different types, because js does not require you to declar types
- ok, now I think most of you should know about C, or other languages that looks similar to C
- js also borrows a lot of syntaxes from C
- we have if, switch, and for and others, just showing you some examples here
- just note that in for loop, we use var instead of int
- functions in js
- again, since no need to declare types

- we just use the key word function to declare function
- arguments also won't have type
- but in js, functions can be treated as variables too
- so here I can declare a variable a_func, and assign it to a function like so
- then I can assign it to another variables
- and i can call them both
- it's the same as calling one function twice
- small q & a

Express

- Finally let's start with the tools
- nodejs platform and express, a framework on nodejs
- So node.js is a javascript run-time, need not be just a server
- and something particular about it is that it favors async style
- so what's async? let's start with callback first
- I mentioned that in js functions can be treated as variables
- so as variables, they can of course be passed as parameters
- we usually can these function parameters as callbacks
- because we usually let the main function call the function back
- here's an example
- **demo with node**
- So here's the idea of async
- by using callbacks, we pass a task as a function to the program
- and the function will be called at the correct time
- and meanwhile, the program can handle other things
- OK, so next is express
- It's a framework in nodejs which is really easy to get start with.
- It uses the MVC structure we talked about just now ...
- last thing before we start
- Introducing npm, which stands for node package manager
- as the name says, it can help you install libraries locally to your directory
- I will show you how to work with it
- OK finally let's get started.
- we will first set up our application
- now for windows users, run the node command line, for mac users fire up your terminal
- `npm init`

//package.json

```
{
    "name": "MyBlog",
    "description": "My blog",
    "version": "0.0.1",
    "dependencies": {
        "express": "^4.0.0",
        "mongoose": "^4.1.12"
    }
}
```

run command

- how to install new packages?

break to resolve issues # after break

- create a file called config.json
- this will contain all the configuration for our server
- reason for this is that you may have the server running on different places

//config.json

```
{
    "domain": "localhost",
    "port": 3000
}
```

- ok, now it's time for our server! create a file call app.js

```
// app.js
// explain require
var config = require("./config.json");
var express = require("express");

var app = express();

// http get method, there are also other methods
// get is when you want to just get a page.
app.get("/", function(request, response){
    response.send("Greetings world!");
});

// later
app.get("/login", function(req, res){
    response.send("<h1>This is not the login page.</h1>");
})

// later later
app.get("/post/:id", function(req, res){
    res.send("ID is " + req.params.id);
});

app.listen(config.port);
console.log("listening on port", config.port);
```

Routing

- Alright, so we have just set up a server that routes the requests
- To recap, our first route is the direct url slash, blah
- there can also be something after the url
- we can also extract relevant information from the url

EJS

- what we have seen just now is really simple string response
- most of the time, we want to create some html page
- so we can use templates
- what are templates? here's an example of template
- we have a full html here, except that we mark some places with variables
- so it tells the program in run-time that the program should replace those variable names with their content
- thus generating a real, meaningful html
- why use templates?
- Of course you can just create the html with string concatenation like so
- but that is really messy
- hard to break line
- no indentation
- what if you have a really complicated page structure?
- you can write templates like how you write normal html
- also danger of html injection
- our blog is for personal use, but we can be building sites that lets others post content
- a malicious user can post something with ill intend
- here's an example
- we use string concatenation
- but what if the user put some script tags?
- he can do a lot of things on the page, including hijacking the user's identity
- of course there are tools to sanitize html
- but a template engine typically have such functionality built-in, so less worry
- and today we will be using ejs
- here's what it's like
- it has these two special markups
- one for js, one for putting javascript values into the page
- rest just html
- here's an example
- just for loop, then in each loop we generate an li tag that contains the title

hands on

```
$ npm install --save ejs
```

- go back to our app.js

```
//var config = require("./config");
//var express = require("express");
var ejs = require('ejs');

//var app = express();

app.set('views', __dirname + '/views');
// __dirname returns the current directory of the app
app.engine('html', ejs.renderFile);

//app.get("/", function(request, response){
    response.render("index.ejs", {title: "My blog", content: "Greetings visitor!"});
//});

//app.get("/post/:id", function(req, res){
//    res.send("ID is " + req.params.id);
//});

//app.listen(config.port);
//console.log("listening");
```

- as we promise in the code
- create a folder called 'views'
- then edit a file called index.ejs in it

```
<html>
<head>
    <title>
        <%= title %>
    </title>
</head>
<body>
    <%= content %>
</body>
</html>
```

- run, ask
- Ok let's make our index more meaningful
- normally we want to show our blog posts in the main page
- let's go back to index.ejs

```
<html>
<head>
    <title>
        <%= title %>
    </title>
</head>
<body>
```

```
    <header>
        <h1>My Blog</h1>
    </header>
    <ul>
    <% for(var i = 0; i < posts.length; i++){ %>
        <li>
            <div class="title">
                <%= posts[i].title %>
            </div>
            <div class="body">
                <%= posts[i].body %>
            </div>
        </li>
    <% } %>
    </ul>
</body>
</html>
```

- done with the view, now back to app.js
- let's just make some fake data

```
app.get("/", function(request, response){
    response.render("index.ejs", {
        posts: [{
            title: "Post 1",
            body: "Some content"
        }, {
            title: "Post 2",
            body: "Some content again"
        }],
        title: "My Blog"
    });
});
```

- run
- let's also make the page for blog post
- Edit blogPost.ejs

```
<html>
<head>
    <title>
        <%= title %>
    </title>
</head>
<body>
    <header>
        <h1>My Blog</h1>
    </header>
    <article>
    <header><h2><%= post.title %></h2></header>
    <div class="body">
        <%= post.body %>
    </div>
</article>
```

```
</body>
</html>
```

- realise that header and footer the same
- use include, copy paste

```
<% include header.ejs %>
```

- add copy right footer

```
<footer> Copyright &copy; 2015 Me! </footer>
```

- again, let's setup our app.js for the blog page

```
app.get("/posts/:id", function(req, res){
    res.render("blogPost.ejs", {
        post: {
            title: req.params.id,
            body: "Content for " + req.params.id
        },
        title: "My Blog | " + req.params.id
    });
});
```

- just very simple data
- run

Static assets

```
app.use(express.static('public'));
```

MongoDB

- So it kinda works, but to manage data, we wouldn't want to store things in variables
- variables are in memory, they are lost once the server restart
- we put data into databases, which handles storing, retrieving and indexing of data for us
- and today we will be using mongodb
- So mongodb is one of the popular nosql databse
- sql stands for standard query language
- nosql means they deviate from such standard
- so in mongo, data is stored as json document
- mongo also provides full index suppose
- indexing means that if you want to search or rank something, you can do it really fast
- and again as nosql, mongo provides the nosql perks. mainly that it can scale well
- with distributed servers, you can read more on this if interested
- but today we are going to use something on top of mongo
- Mongoose, blah

- Make a directory called test
- **set up** `bin/mongod --dbpath ../test`
- create a file blogPost.js

```
// node modules use this
// use a function to construct our model
// in defining the model, we do not have info about the database
module.exports = function(db){
    var mongoose = require("mongoose");
    mongoose.connect("mongodb://" + db.host + "/" + db.database);

    var schema = new mongoose.Schema({
        title: String,
        body: String,
        date: { type: Date, default: Date.now }
    });

    return mongoose.model("BlogPost", schema);
};
```

- now set up our database in config

```
{
    "domain": "localhost",
    "port": 3000,
    "secret": "thegame",
    "db": {
        "host": "localhost",
        "database": "MyBlog"
    }
};
```

- now let's update our app.js

```
var BlogPost = require("./blogPost")(config.db);

//later
// explain mongo id is not string
var ObjectId = require('mongoose').Types.ObjectId;

app.get("/", function(req, res){
    BlogPost.find().sort({date: -1}).exec(function(err, data){
        res.render('index.ejs', { title: "My Blog", posts: data });
    });
});

app.get("/post/:id", function(req, res){
    var id;
    try{
        id = new ObjectId(req.params.id);
    }catch(e){
        res.sendStatus(404);
        // res.status(404);
        // res.render("404.ejs", { title: req.params.id + " not found" });
        return;
```

```
    }
    BlogPost.find({_id: id}).exec(function(err, data){
    // _id is what we use to reference a unique element in mongodb
        if(err || data.length === 0){
            res.status(404);
            res.render("404.ejs", { title: req.params.id + " not found" });
        }else{
            res.render("blogPost.ejs", { title: data[0].title, post: data[0] });
        }
    });
});
```

- But we also want the user to update blogs
- first the front end
- let's edit blogForm.ejs

```
<% include header %>
<form method="post">
// post means that we are sending data to the server and the server may update
// its data based on the request
    <ul>
        <li>
            <label for="entryTitle">Title: </label>
            <input id="entryTitle" name="entryTitle" />
        </li>
        <li>
            <label for="entryBody">Body: </label>
            <textarea id="entryBody" name="entryBody"></textarea>
        </li>
    </ul>
    <input type="submit" />
</form>
<% include footer %>
```

- So to make use of this, in app.js

```
app.get("/post", function(req, res){
    res.render('blogForm.ejs');
});
```

- to really make use of the form

```
$ npm install --save body-parser
```

- add names to forms

```
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.post('/post', function(req, res) {
    BlogPost.create({
        title: req.body.entryTitle,
        body: req.body.entryBody
    }, function(err, data) {
        if (!err) {
            res.redirect('/post/' + data._id);
```

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

- enhance

```
// index
<div class="title">
    <a href="<%= '/post/' + posts[i]._id %>">
    <%= posts[i].title %>
    </a>
</div>

// header
<header>
    <h1><a href="/">My Blog</a></h1>
    <a href="/post">new entry</a>
</header>
```

- Congrats, now you have a functioning blog! Although it's really ugly and insecure!
- Because anyone can create a post!

Session

- So now let's implement something to identify the user
- blah
- with cookies
- setup config

```
$ npm install --save cookie-parser
$ npm install --save express-session
```

```
var cookieParser = require('cookie-parser');
var session = require('express-session');

app.use(cookieParser());
app.use(session({
    secret: config.secret,
    resave: true,
    saveUninitialized: false
}));

// and post
app.get("/post", function(req, res){
    if(!req.session.loggedIn){
        res.redirect("/login");
    }else{
        res.render('blogForm.ejs', { title: "My Blog | New Post" });
    }
});
```

- also setup login

```
app.get("/login", function(req, res){
    res.render('loginForm.ejs', { title: "My Blog | Login" });
});

app.post('/login', function(req, res) {
    if (req.body.password === 'whosyourdaddy') {
        req.session.loggedIn = true;
        res.redirect('/post');
    } else {
        req.session.loggedIn = false;
        res.redirect('/login');
    }
});
```

```
<% include header %>
<form method="post">
    <ul>
        <li>
            <label for="username">User name </label>
            <input type="text" name="username" id="username" />
        </li>
        <li>
            <label for="password">Password </label>
            <input type="password" name="password" id="password" />
        </li>
    </ul>
    <input type="submit" />
</form>
<% include footer %>
```

- run