CSCI 4140 – Tutorial 9

Express: Web Framework for Node.js

Matt YIU, Man Tung (mtyiu@cse)
SHB 118
Office Hour: Wednesday, 3-5 pm

2016.03.29

Outline

- What is Express?
- Getting started
- Routing
- Using middleware
- Miscellaneous

What is Express?

- *"Fast, unopinionated, minimalist **web framework** for Node.js"* (From http://expressjs.com/)
- It is a Node.js module that is designed for **web applications**
  - Available in **npm**
  - Provide **useful utilities** for web applications, e.g., **routing**, **support for middleware**, different **template engine**, …
- It helps *(but does not force)* you organize your web application into an **MVC (Model-View-Controller) architecture** on the server side

Ref.: http://stackoverflow.com/questions/12616153/what-is-express-js

= Important

Getting started

Let's say "Hello World" in Express!

Getting started

- Please refer to the notes "**Installing Node.js and Express on [Windows|Linux or Mac]**" in **Week #7**
- This tutorial assumes that you have installed the **Express application generator**
- All commands in the **terminal** (for Linux or Mac) / **command prompt** (for Windows) will be indicated with a "**$**" sign
- Let's start with an **empty directory**

Create an Express application

- Use the Express application generator to create an application skeleton with the "**express**" command
  - Create at the current directory: `$ express` or
  - Create an Express app named "myapp" under the current directory:

```
$ express myapp
```

- Change the current directory to the Express app directory and install dependencies

```
$ npm install
```

- ***Note:*** You can write Express application without using the application generator, especially when the application structure is simple!

Express app directory structure

- **bin**
  - www
- **node_modules**
  - *(Content omitted)*
- **public**
  - **images**
  - **javascripts**
  - **stylesheets**
    - style.css
- **routes**
  - index.js
  - users.js
- **views**
  - error.jade
  - index.jade
  - layout.jade
- app.js
- package.json

The startup script of the application

The code of the dependencies (generated by **`npm install`**)

Public assets in the application (e.g., images, JavaScript and CSS files)

The routes used in the application (exported as modules)

The view template files to be rendered by **Jade template engine** (which converts the template files into HTML)

Express app directory structure

- **bin**
  - www
- **node_modules**
  - *(Content omitted)*
- **public**
  - **images**
  - **javascripts**
  - **stylesheets**
    - style.css
- **routes**
  - index.js
  - users.js
- **views**
  - error.jade
  - index.jade
  - layout.jade
- app.js
- package.json

The script of your application (loaded by the startup script **bin/www**)

The configuration file of this Node.js project, which holds its metadata.
npm uses this file to identify the project and handle the project's dependencies.

Express app directory structure

- bin
  - www
- node_modules
  - *(Content omitted)*
- public
  - images
  - javascripts
  - stylesheets
    - style.css
- routes
  - index.js
  - users.js
- views
  - error.jade
  - index.jade
  - layout.jade
- app.js
- package.json

Quoted from the Express "Getting started" guide:

"The app structure generated by the generator is just one of the multiple ways of structuring Express apps. Feel free to not use it or to modify it to best suit your needs."

However, I suggest you following this structure at the beginning!

"Hello World" in Express

- The default startup script "**bin/www**" is a bit complicated
- Let's start with a simplified script first (**server.js**):

```
var express = require( 'express' );
var app = express();

app.get( '/', function ( request, response ) {
    res.send( 'Hello World!' );
} );

var server = app.listen( 4140, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log( 'Listening at http://%s:%s', host, port );
} );
```

server.js

"Hello World" in Express

```
var express = require( 'express' );
var app = express();

app.get( '/', function ( request, response ) {
    res.send( 'Hello World!' );
} );

var server = a                                    ) {
    var host =
    var port =
    console.log( 'Listening at http://%s:%s', host, port );
} );
```

server.js

Load the **express** module into a variable.

Initialize the application using the **express** variable.

Set up the application using the methods in **app**.
app.get() is used to set up a route for GET requests.

"Hello World" in Express

```
var express =
var app = expr

app.get( '/', function ( request, response ) {
    response.send( 'Hello World!' );
} );

var serv
    var
    var
    cons
} );
express/
```

Path of the route. In this example, all GET requests to the URL "/" will be handled by the callback set up here.

The callback of the GET requests for the route "/". The callback has the following signature:

```
function( request, response ) { /* ... */ }
```

where **request** and **response** are the objects representing the client's request and server's respond respectively.
They are exactly the same objects that Node provides, so you can use those Node's methods to handle the requests and responds.

"Hello World" in Express

This method binds and listens for connections on the specified host (optional) and port. This method is identical to Node's **`http.Server.listen()`**. It returns an object representing the **`server`** instance (and this will be useful for setting up Socket.IO connections).

It takes one required argument (port number), followed by three optional arguments (hostname, backlog and a callback function).

```
var expr
var app

app.get(
    resp
} );

var server = app.listen( 4140, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log( 'Listening at http://%s:%s', host, port );
} );
```

Port number

Callback function

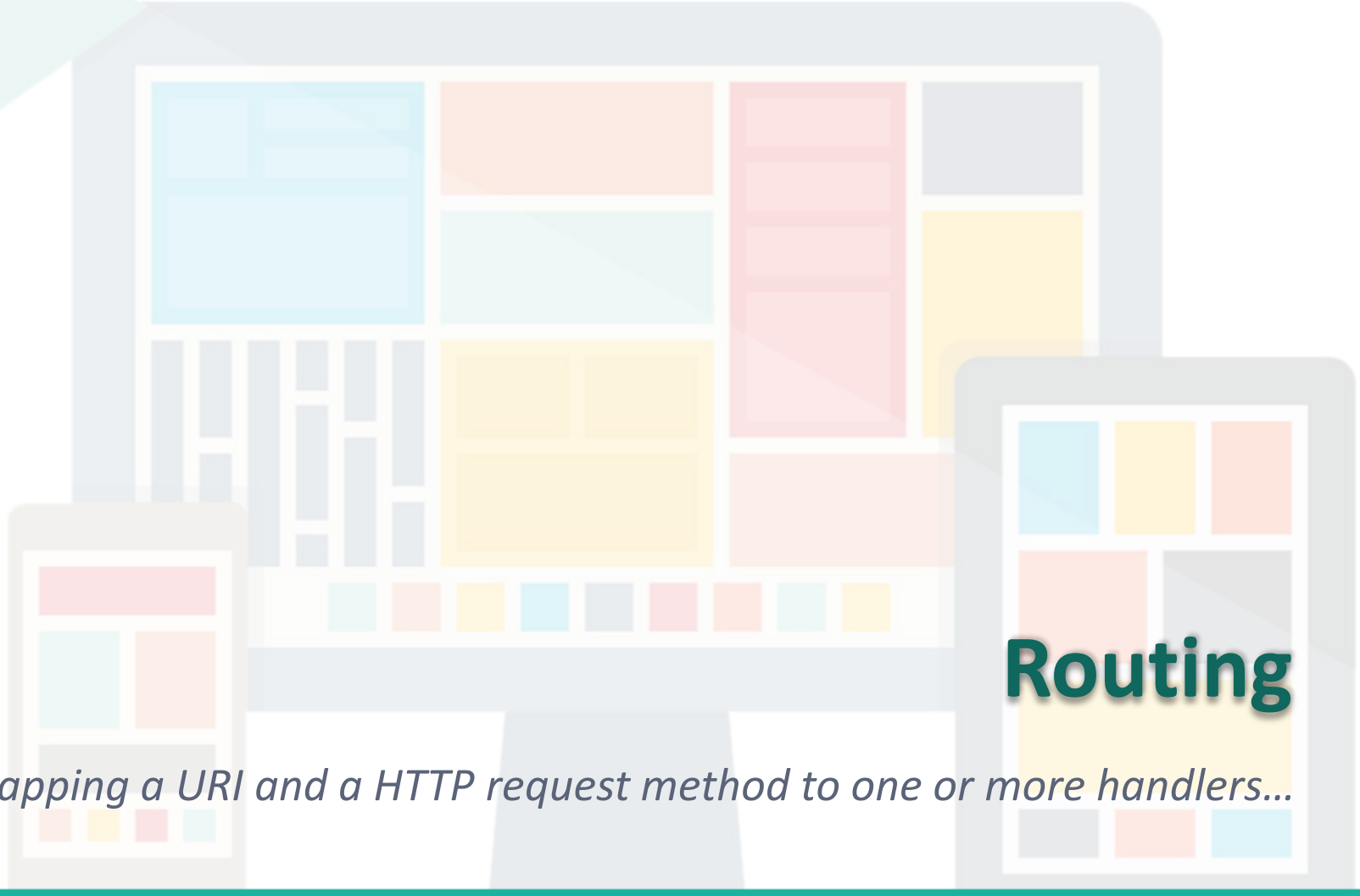
express/server.js

"Hello World" in Express

- Let's run our first Express application:

```
$ node server.js
```

- Visit http://localhost:4140/ to see the output



Routing

Mapping a URI and a HTTP request method to one or more handlers…

Routing

- Routing refers to determining how an application **responds to a client request** to a **particular endpoint**, which is a **URI** (or **path**) and a specific **HTTP request method** (e.g., GET, POST, etc.)
  - Node.js does not provide a convenient way to define routes without a third-party modules
- The route can be specified as a **string**, a **string pattern**, or a **regular expression**!
- The HTTP request methods supported are:
  - GET, POST, DELETE, PUT, …
- For each route, we provide one or more **handlers** (i.e., callback functions)

Routing: Route methods

- Use **app.get()** and **app.post()** to specify routes for GET and POST requests respectively:
  - The URI ("/" in this example) can be the same for different routes as long as the HTTP request methods are different

```
app.get( '/', function ( request, response ) {
    response.send( 'GET request received' );
} );

app.post( '/', function ( request, response ) {
    response.send( 'POST request received' );
} );
```

  - To define a route which only consider the URI without the HTTP request methods, use **app.all()**

Routing: Route paths – Strings

- **Route paths**, in combination with a request method, define the **endpoints** at which requests can be made to
- They can be **strings**, **string patterns**, or **regular expressions**
- Examples of route paths based on **strings**:

```
// Will match GET requests to the root
app.get( '/', function ( request, response ) {
    res.send( 'root' );
} );
// Will match GET requests to /about
app.get( '/about', function ( request, response ) {
    res.send( 'about' );
} );
```

Routing: Route paths – String patterns

- Examples of route paths based on **string patterns**:

```
// Will match acd and abcd
app.get( '/ab?cd', function( request, response ) {
    res.send( 'ab?cd' );
} );
// Will match abcd, abbcd, abbbcd, and so on
app.get( '/ab+cd', function( request, response ) {
    res.send( 'ab+cd' );
} );
// Will match abcd, abxcd, abRABDOMcd, ab123cd, and so on
app.get( '/ab*cd', function( request, response ) {
    res.send( 'ab*cd' );
} );
// Will match /abe and /abcde
app.get( '/ab(cd)?e', function( request, response ) {
    res.send( 'ab(cd)?e' );
} );
```

Routing: Route paths – Regular expressions

- Examples of route paths based on **regular expressions**:

```
// Will match anything with an a in the route name:
app.get( /a/, function( request, response ) {
  res.send( '/a/' );
} );

// Will match butterfly, dragonfly,
// but not butterflyman, dragonfly man, and so on
app.get( /.*fly$/, function( request, response ) {
  res.send( '/.*fly$/' );
} );
```

Routing: Route parameters

- It is useful to include **parameters** in a route
  - E.g., "`/users/3`" refers to a route that load the user information whose ID is 3
- We can define a **route parameter** using the "`:[PARAMETER NAME]`" syntax
- The values of the route parameters can be read from `request.params`
- The route parameters can be matched with **regular expressions**

Routing: Route parameters

- Examples of **route parameters**:

When a client send a GET request to "**/message/sosad**", the message "**sosad**" will be stored in **request.params.message**.

```
app.get( '/message/:message', function ( request, response ) {
    response.send( 'The message: ' + request.params.message );
} );

app.get( '/room/:id([0-9]+)', function ( request, response ) {
    response.send( 'The room number is: ' + request.params.id );
} );
```

express/server.js

This route checks whether the string after "**/room/**" in the GET request contains numbers only.
If this route matches , the parameter value is available in **request.params.id**.

Routing: Route handlers

- We already see how to set up a single callback function in the Hello World example
- Express allows us to have **multiple callback functions**:

```
app.get( '/example/b', function ( request, response, next ) {
    console.log( 'First handler' );
    next();
}, function ( request, response ) {
    response.send('Second handler');
} );
```

You need to specify the **`next`** object when there are more than one handlers.
Before a handler returns, call "**`next()`**" such that the next handler can be executed.

- You can also pass an **array** of callback functions to `app.get()`
  - Read the examples on http://expressjs.com/guide/routing.html

Response methods

- To send a response to the client, use the **`response`** object in the callback function of a route
  - Useful in Assignment 2:
    - `response.json()`
    - `response.redirect()`
    - `response.sendFile()`

Method	Description
download()	Prompt a file to be downloaded.
end()	End the response process.
json()	Send a JSON response.
jsonp()	Send a JSON response with JSONP support.
redirect()	Redirect a request.
render()	Render a view template.
send()	Send a response of various types.
sendFile()	Send a file as an octet stream.
sendStatus()	Set the response status code and send its string representation as the response body.
set()	Set the response's HTTP header.

Response methods in Assignment 2

- In Assignment 2, there are only 2 pages:
  - **Index page:** For redirecting users to a dedicated session
    - No HTML files are needed – Use `response.redirect()`
  - **Session page:** For displaying the video player, remote control & playlist
    - Everything is loaded asynchronously with JavaScript
    - The HTML file is a static page, meaning that there are no dynamic contents generated on the server → No template engine is required
    - Use `response.sendFile()` instead!
- To serve the static HTML page to the client:

```
var path = require( 'path' );
response.sendFile( path.resolve( __dirname +
'/../views/index.html' ) );
```

Absolute path to the HTML file is required

Serving static assets

- Static assets refer to the static contents of an application, including images, CSS and JavaScript files
  - By default, they are stored under the "**/public**" directory
- To serve static assets in an Express application, **app.js** contains the following line:

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

Everything inside this folder will be served to the client without further configuration.

- After that, you can refer to the assets in your web page:

```
<img src="/images/software.jpg">
```

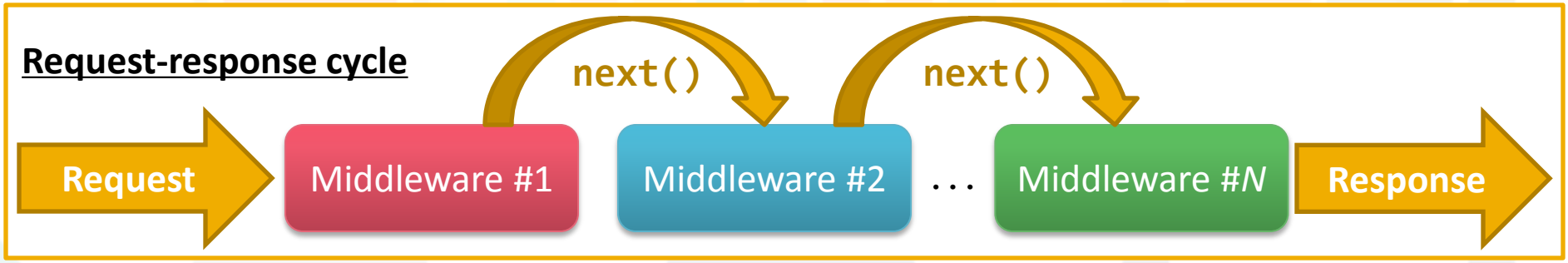
Do you notice the "**/**" at the beginning of the path? Yes, you need to refer to the image with **absolute path**!

Using middleware

"An Express application is essentially a series of middleware calls."

Using middleware

- Middleware is a function with access to
  - The **`request`** object
  - The **`response`** object
  - The **`next`** middleware
- The complete series of middleware calls forms a **request-response cycle**
  - Calling **`next()`** passes control to the next middleware; otherwise the request will be left **hanging**



Using middleware

- An Express application can use the following kinds of middleware:
  - **Application-level** middleware
  - **Router-level** middleware
    - We will only discuss this one
  - **Built-in** middleware
  - **Third-party** middleware

Read http://expressjs.com/guide/using-middleware.html for more details

Router-level middleware

- In order to better organize your application code by routes, we can separate them into multiple files
- Let's say, we arrange the source code into these files:

```
routes
    alice.js    (all routes prefixed with "/alice")
    bob.js      (all routes prefixed with "/bob")
server.js       (startup script which loads alice.js and bob.js)
```

- Note that we do not have access to the app variable in `alice.js` and `bob.js`
  - How can we set up routes in these files?
  - **Answer:** Use **`require( 'express' ).Router()`** to get the **router level middleware**

Router-level middleware: Example

- In the external route file…

```
var router = require( 'express' ).Router();

router.get( '/', function( request, response ) {
        response.send( '[/alice] Home' );
});

router.get( '/whoami', function( request, response ) {
        response.send( '[/alice/whoami] I am Alice!' );
});

module.exports = router;
```

routes/alice.js

Router-level middleware: Example

- In the external route file…

```
var router = require( 'express' ).Router();

router.get( '/', function( request, response ) {
        response.send( '[/alice] Hom
});

router.get( '/whoami', function( request, response ) {
        response.send( '[/alice/whoami] I am Alice!' );
});

module.exports = router;
```

routes/alice.js

Load the router-level middleware

Set up the routes for GET requests (same as `app.get()`, but on the **`router`** object)

Export the **`router`** object such that it can be used as a middleware when the file is loaded as a module

Router-level middleware: Example

- In the startup script…

```
// ...Omitted

// Load external routes
// --------------------
var alice = require( './routes/alice' );
var bob = require( './routes/bob' );
app.use( '/alice', alice );
app.use( '/bob'  , bob );
```

server.js

Router-level middleware: Example

- In the startup script…

```
// ...Omitted

// Load external routes
// --------------------
var alice = require( './routes/alice' );
var bob = require( './routes/bob' );
app.use( '/alice', alice );
app.use( '/bob' , bob );
```

server.js

Load **routes/alice.js** and **routes/bob.js** into the variables **alice** and **bob** respectively.

Path

Middleware functions

Mount the middleware functions at the specified paths.

Router-level middleware: Example

- Routes handled by `routes/alice.js`:
  - `/alice`
  - `/alice/whoami`
- Routes handled by `routes/bob.js`:
  - `/bob`
  - `/bob/whoami`
- Routes handled by `server.js`:
  - `/`
  - `/preview`
  - `/message/:message`
  - `/room/:id([0-9]+)`

Before deploying an Express application

Though not required, it is a good practice to clean up unused code!

Before deploying to the production server

- Add a **.gitignore** file to the project root directory to exclude the "**node_modules**" directory from the Git repository
  - When you deploy your application to the production server, you need to execute "**npm install**" to install all dependencies
  - OpenShift handles this for you

```
node_modules
node_modules/*
```

Sample .gitignore file

- Remove all **unused dependencies** in **package.json**
  - In this example, only the "**express**" module is needed
- Remove all **unused code** that is generated by the Express application generator

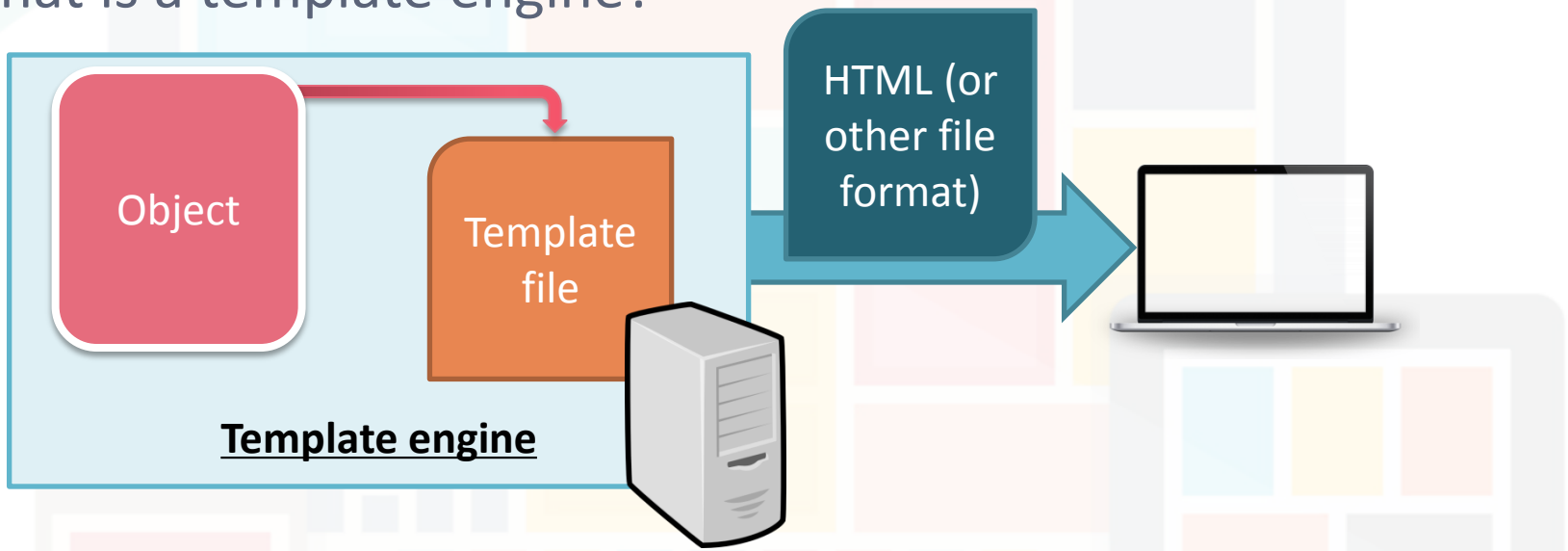
Note: The example code has gone through the above procedure

Miscellaneous

Note: These topics are not involved in Assignment 2

Template engines

- What is a template engine?



- Most web frameworks support rendering template files with template engines
  - E.g., Django (for Python), Symfony (for PHP), Ruby on Rails

Jade template engine: Example

- Express use **Jade** template engine

```
html
  head
    title!= title
  body
    h1!= message
```

views/index.jade

```
app.set( 'views', __dirname + '/views' );
app.set( 'view engine', 'jade' );
app.get( '/hi', function ( request, response ) {
    response.render( 'index',
                    { title: 'Hey', message: 'Hello there!' } );
} );
```

Set up the template engine

app.js

Jade template engine: Example

- Express use **Jade** template engine

Template file

```
html
  head
    title!= title
  body
    h1!= message
```

views/index.jade

```
app.set( 'views', __dirname + '/views' );
app.set( 'view engine', 'jade' );
app.get( '/hi', function ( request, response ) {
    response.render( 'index',
                     { title: 'Hey', message: 'Hello there!' } );
} );
```

Render the template file

Object

app.js

Jade template engine: Example

- Express use **Jade** template engine

```
html
  head
    title!= title
  body
    h1!= message
```

views/index.jade

```
<html><head><title>Hey</title></head><body><h1>Hello there!</h1></body></html>
```

Output HTML

```
app.set( 'views', __dirname + '/views' );
app.set( 'view engine', 'jade' );
app.get( '/hi', function ( request, response ) {
    response.render( 'index',
                     { title: 'Hey', message: 'Hello there!' } );
} );
```

app.js

Template engines in Express

- Express uses Jade by default
  - Read http://expressjs.com/guide/using-template-engines.html for more details
- I prefer **Swig** instead
  - It uses similar methodologies as Django, Jinja2, and Twig template engines
  - Ref.: http://paularmstrong.github.io/swig/

Other interesting topics in Express

- Third-party middleware
  - http://expressjs.com/resources/middleware.html
  - Some of them are installed by default
  - You can install or remove them using **npm**
- Error handling
  - http://expressjs.com/guide/error-handling.html
- Debugging
  - http://expressjs.com/guide/debugging.html
- Database integration
  - http://expressjs.com/guide/database-integration.html

– End –