Intro to web development

# Key points to cover

* Displaying html pages

Day 1

Day 2

Day 3

Day 4

Day 5

* Json Api
* Project structuring
* Project structuring
* Concept of Controllers
* Mysql
* Web pages and apis with form submission
* Write these data to mysql
* Modify content in db
* Login, signup, forgot password flows

# HTML Template

In the beginning we learned a bit about html. But so far we never used it. In fact, web pages are supposed to be documents with rich content, not just some lines of text.

We could display a web page by doing this

ctx.body=”{long html code here}”

This may look easy now, but what if the html code is 15 pages long. How about when another person is writing the html code.

A simple trick you can do is to keep your html in separate files, read it as string and display it. This could work, but then we have to write a lot of unnecessary code to do all that. That is where template rendering comes in.

A template rendering module does the same thing but in a reusable manner. A template is written for a templating engine to produce the desired html code. So yeah, its one more scripting syntax to learn. We will be using koa-views with ejs for templating

Exercise

git checkout ex3.1

npm install koa-views ejs

we can install multiple modules in a single command with npm. The above command does exactly that. it installs koa-views and ejs (2 different modules) with a single command. You can add more module names to its end to install all of them in a single command.

## Setting up

In index.js you’ll see these additions from line 8

**const** views = require('koa-views');

Imports koa-views module into the app.

*// Must be used before any router is used*

**let** viewOpts = {

extension: 'ejs'

};

This is the configuration options object for koa-views. You can get a detailed api documentation at <https://github.com/queckezz/koa-views#api>

**const** viewsMiddleware = views(\_\_dirname + '/views', viewOpts);

This initialises the view middleware

app.use(viewsMiddleware);

views middleware should be registered before routs are registered.

This is all the setup process to using view templating in koa.

The next step is we create the template files. These files should reside in ./views of project directory. We set this up at line 19 of code. \_\_dirname + '/views' effectively means {directory\_of\_main\_js\_file}/views . In our case means the directory in which index.js resides.

## The template

Next step is to create a views folder and a welcome.ejs file in it.

Here you can write anything that is valid html. But there is more we can do here. At line 4 you will see this

<title>

<%= title %>

</title>

The <%= title %> is templating syntax for ejs. What it does is simply substitute the value for key ‘title’ passed in to the templating engine. You’ll see this payload passing in the next step.

You can read about all the ejs syntaxes in the Tags section of <http://ejs.co/>

## Rendering

The next step is to render this ejs file into html and send it as the response to a url access.

To do that, we go back to routes.js

We have prepared welcome.ejs for the welcome page. So now we have to render this page when someone visits the welcome page.

We previously had ctx.body = "Welcome" in '/' route, which would display the text “welcome”

We change that to the following

**const** payload = {

title: 'Welcome',

user: 'John',

serverName: "Akash's computer"

};

The payload is the custom data that we pass into the template engine to fill in wherever necessary. Remember the <%= title %> from our ejs file? The payload here has a value “Welcome” for the key title. So When rendering the template, ejs will put “welcome” in place of <%= title =>

await ctx.render('welcome', payload);

This basically renders the template file and sends it down the pipeline to the browser.

We are done. Our first web page is ready.

A tiny bit we missed in between is the **async** keyword inserted at line 4.

So in the beginning we talked about how node handles IO operations asynchronously, meaning your io calls would return rightaway but the io operation is still happening in the background. If you should wait till the end of io operation and do something, you should use callback ( which is a function passed as the 2nd parameter usually by node standards). So async-await is a keyword pair introduced to solve this untidy pattern. A call like this:

await functionName();

would effectively stop execution until the async operation inside the function has completed. There are 3 things to keep in mind though.

1. await keyword can only be used inside an async function. To do that you simply add async keyword before your function definition as below   
   **async** (ctx, next) **=>** {}  
   OR  
   **async** function fnName(ctx, next) {}
2. You can’t run any function with await. Well, you can, but it execute synchronously if it is not a proper async function.
3. Any function that returns a promise is automatically an async function. This you don’t have to worry about for now.

Now you should do the same with the other two routes as well. Create 2 more template files and render them on '/about' and '/contact' routes.

Let students create 2 pages for these routes. Let them do experiment a bit. If you can do a demo session of these pages, it’ll be more interesting but optional.

They can use welcome.ejs and the ‘/’ route as reference. Copy them, rename files, make changes to text… that is all it takes. Remember to make route handlers async functions.

# Controllers

Next exercise code:

git checkout ex3.2

`If we look into our routes.js you’l see that its getting a bit crowded there. Routes is where every url paths that can possibly occur in the app will be linked to a handler function. If you write all the handlers in the routes.js itself, your entire app will be in that single file. That is fine for small apps, but as you start building larger apps, it become difficult to manage. We faced this situation earlier when we separated routes from index.js . We’ll follow the same logic here. We’ll separate the handler functions into different controller files. To do that create a /controllers directory. Now add a file named viewControllers.js so that you can move all the view handler function here.

Then one by one copy all 3 view handler function into this file like this

exports.about = **async** (ctx, next) **=>** { …

where **async** (ctx, next) **=>** { … part is the handler function for about page.

exports.about part makes that function available with name about when you require this file somewhere else.

This is very similar to module.exports we used in routes.js, exports.about = SOMETHING is equivalent to

module.exports = {

about: SOMETHING

}

So it’s the same thing but exports.about looks cleaner

Now we went with viewControllers.js because in this example we only have 3 views and we will add a few web api controllers later. But that is about it. Ours is a small app and have very few controllers.

In a real app, the strategy to split controllers depends on a bunch of other factors. If you are familiar with the MVC pattern, we are basically developing a miniature MVC pattern for our self. Going by that thinking, it’s a good idea to create controllers corresponding to models. Now now we don’t have any models. We’ll come to that very soon.

# Mysql

Assuming everyone knows basic sql, I’m not going into a mysql tutorial. In this section I will talk about setting up mysql to be used with nodejs. Make sure you install mysql server, shell/client and workbench

Note: make sure you install mysql version 5.7. Although the latest version is 8, the nodejs client module does not support authentication with mysql v8 yet. So to avoid complication we stick with the older version

To install mysql go to <https://dev.mysql.com/downloads/mysql/> and select your OS. If using a linux distro, you can install mysql from respective repositories.For ubuntu do following.

sudo apt-get install mysql-server

( Talk to someone with experience setting up mysql server. The steps can vary from system to system)

Exercise

git checkout ex3.3

Before getting started we should create a database and table in mysql. Now you can use the mysql workbench to create a schema and then a users-table. To help you get started, I’ve included a mysql dump which you can import into you database using workbench.

While opening up workbench, it’ll ask you for login credentials. Use the root user credentials you ued during mysql installation. Then in the left pane you’ll see Data import/restore. Click that. Choose the tutorial\_users.sql in *import from self-contained file* box. Then click start import.

If you choose to create schema and tables yourself, then remember to change the schema name and table names used in code. Also make sure your user table has all the columns provided in the dump file.

Once installation and database setup is done, next is integrating mysql with the node app. To do that we need a module that can talk to the mysql server we just installed.

npm install mysql

then in index.js you’ll see this section of code

**const** mysql = require('mysql');

require the mysql module

**const** mysqlConfig = {

host: 'localhost',

user: 'irisind',

password: 'qwertyui',

database: 'tutorial'

};

Default port is 3306, if you use a different port, that should also be provided in the config.

**const** connection = mysql.createConnection(mysqlConfig);

configuring a connection.

connection.connect();

Establishes a connection with mysql server

connection.query('SELECT 1 + 1 AS solution', **function** (error, results, fields) {

if (error) throw error;

console.log('The solution is: ', results[0].solution);

});

Connection.query() takes 2 parameters, 1st is the query string and second is a callback to execute when the query has completed executing. The callback will receive 3 params, error, results and fields. If no error occurred, error param will be null.

connection.end();

Once the operation is done, end the connection. A better place to end connection will be inside the callback. This won’t cause a problem in this example, but when the connection is over a network, connection might get disconnected before completing execution.

Everything is good for now. But here is a problem: All these code has to be repeated any time you need to run a query, which doesn’t look so neat. In fact, there is no need to recreate a connection instance for every query, you can reuse them. To reuse something, you should make them a module, or simply put, make it a separate file that exports the its components.

## ORM

For instructor reference

git checkout ex3.4

One could always use sql queries to create, insert and retrieve data from an sql database. But as developers we are always striving to simplify problems. In a production application we are going to use an ORM to handle the DB interfacing.

The tool we are using is called sequelize

npm i sequelize

create a new folder models/

### Setting up Sequelize

create a new file models/index.js

insert the following content

**const** Sequelize = require('sequelize');

**const** sequelize = new Sequelize('tutorial' /\*database, 'irisind', 'qwertyui', {

host: 'localhost',

dialect: 'mysql',

operatorsAliases: false,

pool: {

max: 5,

min: 0,

acquire: 30000,

idle: 10000

},

});

module.exports = sequelize;

***Connection pool***

*A mysql connection is like a pipeline. Every query executes through this pipeline. A disadvantage of this model is that while executing one query, the pipeline is busy, you can’t execute another query until the previous execution completes and the pipeline is free.*

*A connection pool is a collection of connections pre-initialized by the mysql driver so that when a query is issued, the driver will pick an unused connection. (This is obviously a high level overview, there is a lot of thread level details involved which is being skipped)*

*The above example keeps 5 connection in pool. Read full documentation for more information here.*

### Adding a model

A model is what the name suggests. It is a model of how the data is going to be saved in the database. Its like a class definition. Here we will create a model for Users. The ORM takes in this model and creates a table in the database for us. Then further insert and update operations can be done through a class-object based interface, i.e you can call functions with parameters as data and the ORM will create the corresponding SQL queries for us.

create a new file models/user.js

insert the following content

**const** sequelize = require('./index');

Models are defined with sequelize.define('modelName', {attributes}, {options}).

In the following example the name is user. The object following it is attributes. Attributes are basically table field names and their properties. We do not need the options parameter.

You can read the entire documentation [here](http://docs.sequelizejs.com/class/lib/sequelize.js~Sequelize.html#instance-method-define)

**const** User = sequelize.define('user', {

firstName: {

type: Sequelize.STRING

},

lastName: {

type: Sequelize.STRING

},

email: {

type: Sequelize.STRING,

allowNull: false,

unique: true

},

password: {

type: Sequelize.STRING

}

});

The above code will create a model object , but will not do anything to the database. Calling Sync on the model object creates the table and syncs the data with table.

*The examples provided in the documentation follow the promises way of coding.. We follow async await style. We do not want multiple styles of code in our code base. Try to comprehend this difference.*

**const** initFn = **async** () **=>** {

await User.sync();

*// Table created*

return User;

}

module.exports = initFn;

~~create a mysql.js file in the project root. Copy these from index.js to the new file~~

**~~const~~** ~~mysql = require('mysql');~~

*~~// const mysqlConfig = 'mysql://irisind:qwertyui@localhost:3306/tutorial';~~*

**~~const~~** ~~mysqlConfig = {~~

~~host: 'localhost',~~

~~user: 'irisind',~~

~~password: 'qwertyui',~~

~~database: 'tutorial'~~

~~};~~

**~~const~~** ~~connection = mysql.createConnection(mysqlConfig);~~

~~Add the following to mysql.js so that the connection instance can be accessed from outside the file~~

~~exports.connection = connection;~~

~~Now remove the following part from index.js~~

~~connection.connect();~~

~~connection.query('SELECT 1 + 1 AS solution',~~ **~~function~~** ~~(error, results, fields) {~~

~~if (error) throw error;~~

~~console.log('The solution is: ', results[0].solution);~~

~~});~~

~~connection.end();~~

# Signup and login

Next we will use the users table to setup a signup page and a login page.

# Sessions

<https://github.com/koajs/session>

## Middlewares: session checker middleware

# ~~Web APIs~~

~~I hope you all know what an api is. It’s the abbreviation of~~ **~~application programming interface~~**~~, not that it explains anything but just so that you know. An api is like the steering wheel, pedals and switches in a car. You are not capable of moving the car yourself at 90 miles per hour. But there is a powerful engine in the car that can do it for you. Now, you need an interface to give inputs to car to make it work to your liking. This is what the pedals and switches do. They are the interface.~~

~~Like that, when you are creating an application, sometimes there come situations where you have to use dedicated external applications or libraries to get certain things done. To do this, usually the application or library will provide some way to interact with it and these are usually provided in their documentation. For libraries usually it is function calls.~~

~~A Web API is an api in which this interfacing is done though http requests. i.e instead of function calls, your app will make http requests and instead of function returns, it waits for http response.~~

~~As we have already seen, http requests doesn’t have to be html. You can send plain text or binary as an http response. (A file downloaded from some website is a good example of binary http response). To make it more predictable, http has a content-type header to specify what kind of data is being sent.~~

~~Web api’s can take any form. It totally depends on the API designer. But for easy interoperability, people usually follow a standard well defined API patterns. A RESTful API (REST in short) is such a standard. While REST is a standard, most REST APIs in use aren’t purely REST but rather follow only certain patterns specified in REST. A purely REST API is client agnostic, which means, a standard REST client can access every endpoint without reading documentation or programming according to documentation. But this requirement is neither practical in most cases nor followed in most cases.~~

~~For the sake of brevity, we’ll use the term REST API in this tutorial for the simple api we are creating. Although a web api call is a HTTP request, the response is not to be read by humans, it should be machine friendly so that the calling application can read and interpret the data easily. The response of an API call will be data formatted in some standard data representation format like xml or json.~~

~~We will be using json in our api as javascript has native JSON processing capabilities.~~

~~create a file apiControllers.js in controllers directory.~~