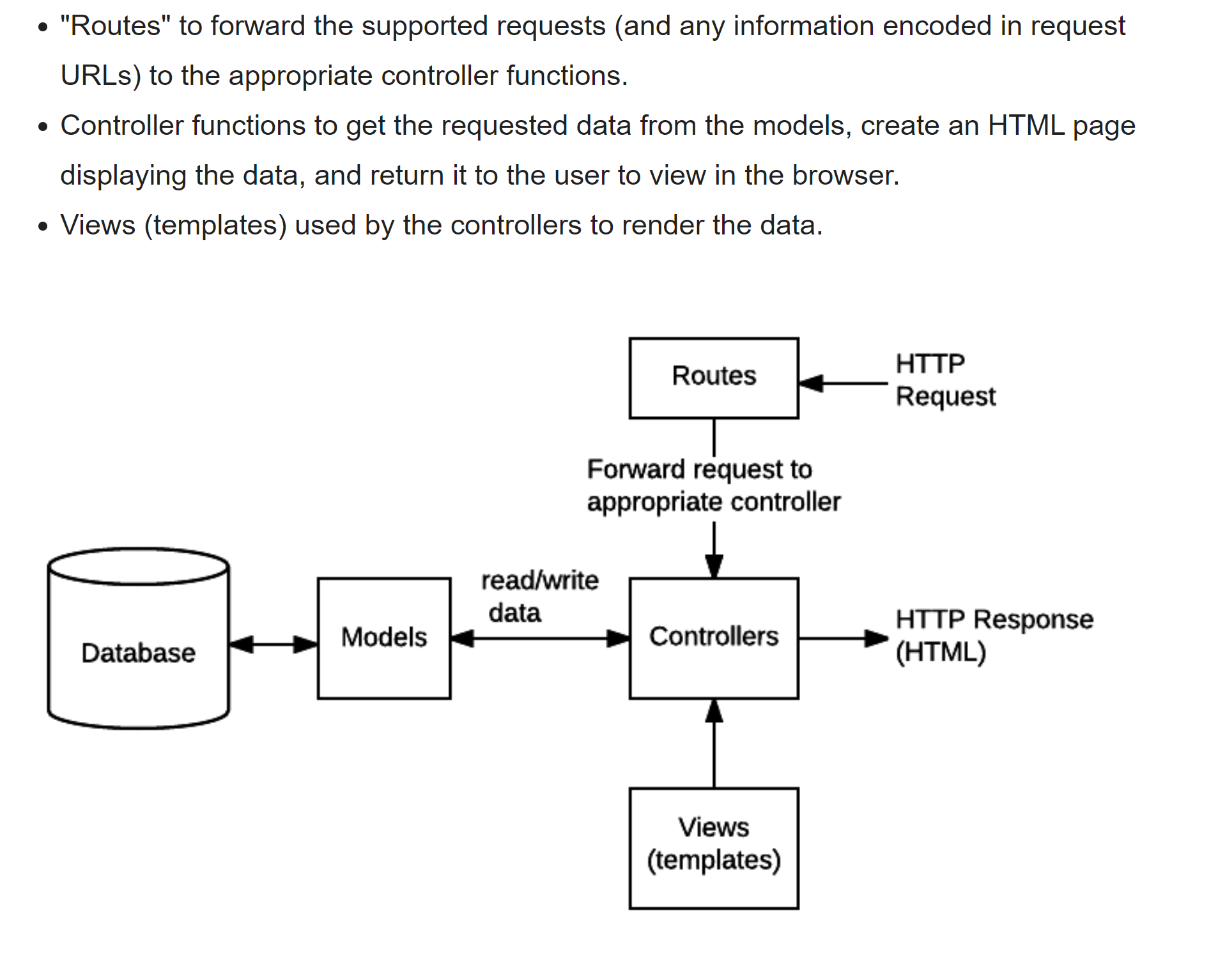
****

[**Routes primer**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#routes_primer)

A route is a section of Express code that associates an HTTP verb (GET, POST, PUT, DELETE, etc.), a URL path/pattern, and a function that is called to handle that pattern.

There are several ways to create routes. For this tutorial we're going to use the [express.Router](http://expressjs.com/en/guide/routing.html#express-router) middleware as it allows us to group the route handlers for a particular part of a site together and access them using a common route-prefix. We'll keep all our library-related routes in a "catalog" module, and, if we add routes for handling user accounts or other functions, we can keep them grouped separately.

First we create routes for a wiki in a module named **wiki.js**. The code first imports the Express application object, uses it to get a Router object and then adds a couple of routes to it using the get() method. Last of all the module exports the Router object.

// wiki.js - Wiki route module.

var express = require('express');

var router = express.Router();

// Home page route.

router.get('/', function (req, res) {

res.send('Wiki home page');

})

// About page route.

router.get('/about', function (req, res) {

res.send('About this wiki');

})

module.exports = router;

To use the router module in our main app file we first require() the route module (**wiki.js**). We then call use() on the Express application to add the Router to the middleware handling path, specifying a URL path of 'wiki'.

var wiki = require('./wiki.js');

// ...

app.use('/wiki', wiki);

The two routes defined in our wiki route module are then accessible from /wiki/ and /wiki/about/.

### [**Route functions**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#route_functions)

Our module above defines a couple of typical route functions. The "about" route (reproduced below) is defined using the Router.get() method, which responds only to HTTP GET requests. The first argument to this method is the URL path while the second is a callback function that will be invoked if an HTTP GET request with the path is received.

router.get('/about', function (req, res) {

res.send('About this wiki');

})

The callback takes three arguments (usually named as shown: req, res, next), that will contain the HTTP Request object, HTTP response, and the next function in the middleware chain.

**Note:** Router functions are [Express middleware](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction#using_middleware), which means that they must either complete (respond to) the request or call the next function in the chain. In the case above we complete the request using send(), so the next argument is not used (and we choose not to specify it).

The  router function above takes a single callback, but you can specify as many callback arguments as you want, or an array of callback functions. Each function is part of the middleware chain, and will be called in the order it is added to the chain (unless a preceding function completes the request).

The callback function here calls [send()](https://expressjs.com/en/4x/api.html#res.send) on the response to return the string "About this wiki" when we receive a GET request with the path ('/about'). There are a [number of other response methods](https://expressjs.com/en/guide/routing.html#response-methods) for ending the request/response cycle. For example, you could call [res.json()](https://expressjs.com/en/4x/api.html#res.json) to send a JSON response or [res.sendFile()](https://expressjs.com/en/4x/api.html#res.sendFile) to send a file. The response method that we'll be using most often as we build up the library is [render()](https://expressjs.com/en/4x/api.html#res.render), which creates and returns HTML files using templates and data—we'll talk a lot more about that in a later article!

### [**HTTP verbs**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#http_verbs)

The example routes above use the Router.get() method to respond to HTTP GET requests with a certain path.

The Router also provides route methods for all the other HTTP verbs, that are mostly used in exactly the same way: post(), put(), delete(), options(), trace(), copy(), lock(), mkcol(), move(), purge(), propfind(), proppatch(), unlock(), report(), mkactivity(), checkout(), merge(), m-search(), notify(), subscribe(), unsubscribe(), patch(), search(), and connect().

For example, the code below behaves just like the previous /about route, but only responds to HTTP POST requests.

router.post('/about', function (req, res) {

res.send('About this wiki');

})

### [**Route parameters**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#route_parameters)

Route parameters are named URL segments used to capture values at specific positions in the URL. The named segments are prefixed with a colon and then the name (e.g. /**:**your\_parameter\_name/. The captured values are stored in the req.params object using the parameter names as keys (e.g. req.params.your\_parameter\_name).

So for example, consider a URL encoded to contain information about users and books: http://localhost:3000/users/34/books/8989. We can extract this information as shown below, with the userId and bookId path parameters:

app.get('/users/:userId/books/:bookId', function (req, res) {

// Access userId via: req.params.userId

// Access bookId via: req.params.bookId

res.send(req.params);

})

**Note:** The URL /book/create will be matched by a route like /book/:bookId (which will extract a "bookId" value of 'create'). The first route that matches an incoming URL will be used, so if you want to process /book/create URLs separately, their route handler must be defined before your /book/:bookId route.

## [Create the route-handler callback functions](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#create_the_route-handler_callback_functions)

Before we define our routes, we'll first create all the dummy/skeleton callback functions that they will invoke. The callbacks will be stored in separate "controller" modules for Books, BookInstances, Genres, and Authors (you can use any file/module structure, but this seems an appropriate granularity for this project).

### [**Controllers**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#author_controller)

Open the **/controllers/authorController.js** file and type in the following code:

var Author = require('../models/author');

// Display list of all Authors.

exports.author\_list = function(req, res) {

res.send('NOT IMPLEMENTED: Author list');

};

// Display detail page for a specific Author.

exports.author\_detail = function(req, res) {

res.send('NOT IMPLEMENTED: Author detail: ' + req.params.id);

};

// Display Author create form on GET.

exports.author\_create\_get = function(req, res) {

res.send('NOT IMPLEMENTED: Author create GET');

};

// Handle Author create on POST.

exports.author\_create\_post = function(req, res) {

res.send('NOT IMPLEMENTED: Author create POST');

};

// Display Author delete form on GET.

exports.author\_delete\_get = function(req, res) {

res.send('NOT IMPLEMENTED: Author delete GET');

};

// Handle Author delete on POST.

exports.author\_delete\_post = function(req, res) {

res.send('NOT IMPLEMENTED: Author delete POST');

};

// Display Author update form on GET.

exports.author\_update\_get = function(req, res) {

res.send('NOT IMPLEMENTED: Author update GET');

};

// Handle Author update on POST.

exports.author\_update\_post = function(req, res) {

res.send('NOT IMPLEMENTED: Author update POST');

};

The module first requires the model that we'll later be using to access and update our data. It then exports functions for each of the URLs we wish to handle (the create, update and delete operations use forms, and hence also have additional methods for handling form post requests — we'll discuss those methods in the "forms article" later on).

All the functions have the standard form of an Express middleware function, with arguments for the request and response. We could also include the next function to be called if the method does not complete the request cycle, but in all these cases it does, so we've omitted it. The methods return a string indicating that the associated page has not yet been created. If a controller function is expected to receive path parameters, these are output in the message string (see req.params.id above)

## [Route module](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#create_the_catalog_route_module)

Ceate routes for all the URLs [needed by the LocalLibrary website](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#local_libary_routes), which will call the controller functions we defined in the previous section.

The skeleton already has a **./routes** folder containing routes for the index and users. Create another route file — **catalog.js** — inside this folder, as shown.

### [**Update the index route module**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#update_the_index_route_module)

We've set up all our new routes, but we still have a route to the original page. Let's instead redirect this to the new index page that we've created at the path '/catalog'.

Open **/routes/index.js** and replace the existing route with the function below.

// GET home page.

router.get('/', function(req, res) {

res.redirect('/catalog');

});

**Databases**

// Access model field values using dot notation

console.log(awesome\_instance.name); //should log 'also\_awesome'

// Change record by modifying the fields, then calling save().

awesome\_instance.name="New cool name";

awesome\_instance.save(function (err) {

if (err) return handleError(err); // saved!

});

// Searching for Records

var Athlete = mongoose.model('Athlete', yourSchema);

// find all athletes who play tennis, selecting the 'name' and 'age' fields

Athlete.find({ 'sport': 'Tennis' }, 'name age', function (err, athletes) {

if (err) return handleError(err);

// 'athletes' contains the list of athletes that match the criteria.

})

// Use Query object to build up a query

// // find all athletes that play tennis

var query = Athlete.find({ 'sport': 'Tennis' });

// selecting the 'name' and 'age' fields

query.select('name age');

// limit our results to 5 items

query.limit(5);

// sort by age

query.sort({ age: -1 });

// execute the query at a later time

query.exec(function (err, athletes) {

if (err) return handleError(err);

// athletes contains an ordered list of 5 athletes who play Tennis

})

//Import the mongoose module

var mongoose = require('mongoose');

//Set up default mongoose connection

var mongoDB = 'mongodb://127.0.0.1/my\_database';

mongoose.connect(mongoDB, {useNewUrlParser: true, useUnifiedTopology: true});

//Get the default connection

var db = mongoose.connection;

//Bind connection to error event (to get notification of connection errors)

db.on('error', console.error.bind(console, 'MongoDB connection error:'));

//Define a schema

var Schema = mongoose.Schema;

// New Scheme Object -> define fields

var ModelSchemaName = new Schema({

a\_string: String,

a\_date: Date

});

// Compile model from schema

var modelName = mongoose.model('modelNamel', ModelSchemaName );

// Create an instance of model SomeModel

var awesome\_instance = new SomeModel({ name: 'awesome' });

// Save the new model instance, passing a callback

awesome\_instance.save(function (err) {

if (err) return handleError(err);

// saved!

});

// You can also use create() to define the model instance at the same time as you save it. The callback will return an error for the first argument and the newly-created model instance for the second argument.

SomeModel.create({ name: 'also\_awesome' }, function (err, awesome\_instance) {

if (err) return handleError(err);

// saved!

});

**Favicon**

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

const favicon **=** require('express-favicon');

const app **=** express();

app.use(favicon(\_\_dirname **+** '/public/favicon.png'));

*// Add your routes here, etc.*

const server **=** app.listen(3000, function(){

    console.log('server is running at %s .', server.address().port);

});