This work is licensed under CC BY 4.0

REST



Using HTTP Methods for RESTful Services

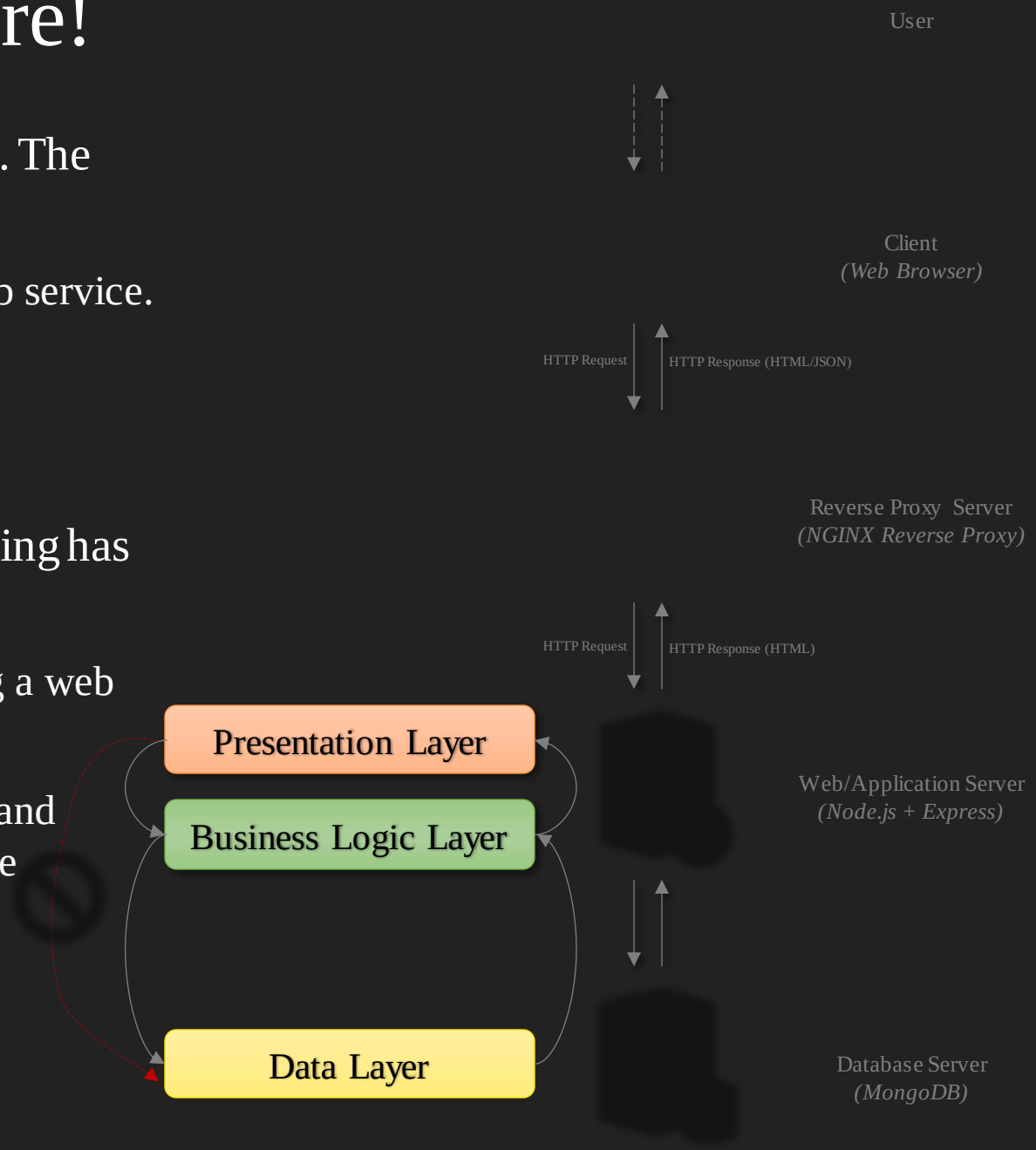
- ✓ Used HTTP verbs (or methods) are POST, GET, PUT, PATCH, and DELETE.
  - These correspond to create, read, update, and delete (or CRUD) operations, respectively.
- ✓ Returned status code depends on the combination of HTTP method and resource URI.

HTTP Verb	CRUD	Entire Collection (e.g. /cars)	Specific Item (e.g. /cars/{id})
`POST`	Create	201 (Created), 'Location' header with link to `/cars/{id}` containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists.
`GET`	Read	200 (OK), list of cars. Use pagination, sorting, and filtering to navigate large lists.	200 (OK), single car. 404 (Not Found), if ID not found or invalid.
`PUT`	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
`PATCH`	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
`DELETE`	Delete	405 (Method Not Allowed), unless you want to delete the whole collection - not often desirable.	200 (OK) ***or 204 (No Content)***. 404 (Not Found), if ID not found or invalid.

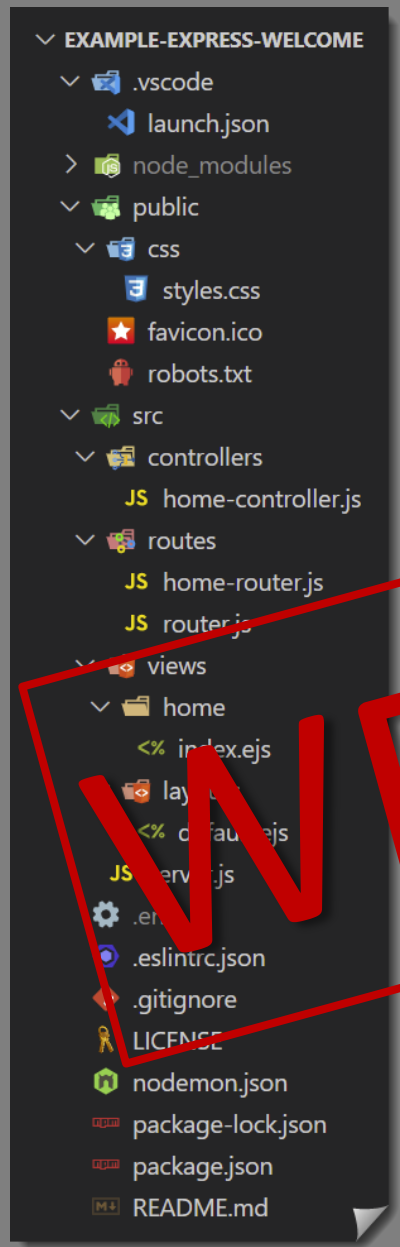


ne same architecture as before!

- The architecture is the same as for web application. The response differ. JSON instead of HTML.
  - No need for a view engine when we create a pure web service.
    - (Express is happy to serve static content.)
    - (NGINX can be configured to serve content.)
- As long as we create a RESTful Web Service, nothing has to be changed.
  - It is the URI that determines if the client is requesting a web service resource or not.
  - The presentation layer still parses the HTTP request, and depending on the requested resource, sends a response containing JSON to the client.



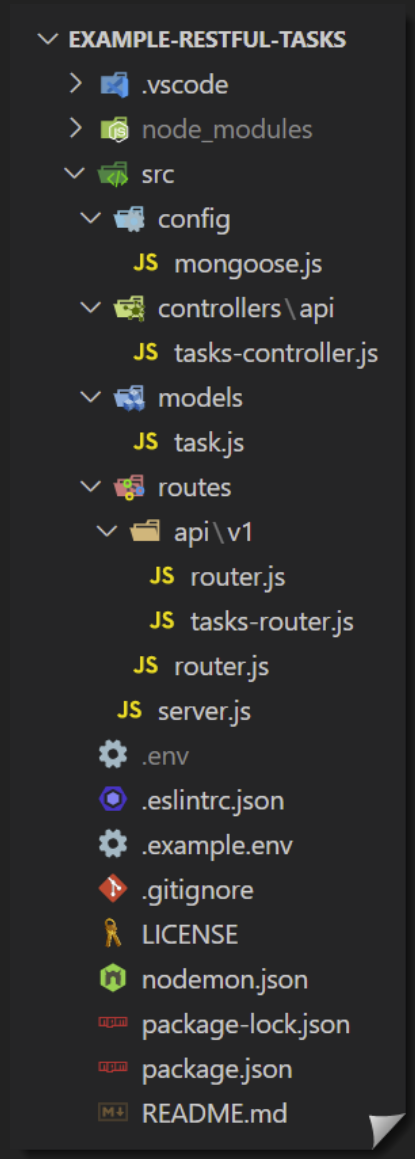
The Final Directory Structure



- ✓ `server.js` is the application's entry point.
- ✓ Routes are stored in separate modules under the `routes/` directory.
- ✓ Controllers are stored under the `controllers/` directory.
- ✓ View templates are stored under the `views/` directory.
- ✓ Static files are stored under the `public/` directory.
- ✓ ... and there are several other opinions.

WEB APPLICATION

The Final Directory Structure



- ✓ `server.js` is the application's entry point.
- ✓ Routes are stored in separate modules under the `routes/` directory.
  - A version has its own directory, which corresponds to the URL.
- ✓ Controllers are stored under the `controllers/api` directory.

The dependencies and devDependencies – `package.json`

```
{
 "name": "example-express-welcome",
 "version": "2.0.0",
 "description": "1DV026 - Example - Express Welcome",
 "type": "module",
 "main": "src/server.js",
 "scripts": {
  "start": "node src/server.js",
  "dev": "nodemon --inspect -r dotenv/config src/server.js",
  "lint": "npx eslint ./src || exit 0",
  "lint:fix": "npx eslint ./src --fix || exit 0"
 },
 "license": "MIT",
 "private": true,
 "dependencies": {
  "date-fns": "^2.27.0",
  "ejs": "^3.1.6",
  "express": "^4.1
  exp s-ejs-lay  "^2.5.
  "morga  "^1.1
 },
 "dev epende  {
  "@In /eslint-config": "^1.1.4",
  "dote v": "^10.0.0",
  "node on": "^2.0.15"
 }
}
```

- ✓ `date-fns`
  - Manage dates.
- ✓ `ejs`
  - A simple template en
- ✓ `e pre s`
  - We framewo for Node.js.
- ✓ `express-ejs-layouts`
  - Layout support for ejs in express.
- ✓ `morgan`
  - HTTP request logger.
- ✓ `dotenv`
  - Loads environment variables from a .env file into process.env.
- ✓ `nodemon`
  - Automatically restarts the application when file changes.

WEB APPLICATION

The dependencies and devDependencies – `package.json`

```
{
 "name": "example-restful-tasks",
 "version": "2.0.0",
 "description": "1DV026 - Example - RESTful Tasks",
 "type": "module",
 "main": "src/server.js",
 "scripts": {
  "start": "node src/server.js",
  "dev": "nodemon --inspect -r dotenv/configsrc/server.js",
  "lint": "npx eslint ./src || exit 0",
  "lint:fix": "npx eslint ./src --fix || exit 0"
 },
 "license": "MIT",
 "private": true,
 "dependencies": {
  "express": "^4.17.1",
  "helmet": "^4.6.0",
  "http-errors": "^1.8.1",
  "mongoose": "^6.1.1",
  "morgan": "^1.10.0"
 },
 "devDependencies": {
  "@lnu/eslint-config": "^1.1.4",
  "dotenv": "^10.0.0",
  "nodemon": "^2.0.15"
 }
}
```

- ✓ express
  - Web framework for Node.js.
- ✓ helmet
  - Helmet helps you secure your Express apps by setting various HTTP headers.
- ✓ http-errors
  - Create HTTP error objects.
- ✓ mongoose
  - MongoDB object modeling tool.
- ✓ morgan
  - HTTP request logger.
- ✓ nodemon
  - Automatically restarts the application when file changes.

The Entry Point – `src/server.js`

```
import express from 'express'
import expressLayouts from 'express-ejs-layouts'
import logger from 'morgan'
import { dirname, join } from 'node:path'
import { fileURLToPath } from 'node:url'
import { router } from './routes/router.js'

// Get the path of the current module's directory
const directoryFullName = dirname(fileURLToPath(import.meta.url))

// Create Express application.
const app = express()

// Set up a morgan logger using the dev format for log entries
app.use(logger('dev'))

// View engine setup
app.set('view engine', 'ejs')
app.set('views', join(directoryFullName, 'views'))
app.use(expressLayouts)
app.set('layout', join(directoryFullName, 'views', 'layouts', 'default'))

// Parse requests of the content type application/x-www-form-urlencoded.
// Populates the request object with a body object (req.body).
app.use(express.urlencoded({ extended: false }))

// Serve static files.
app.use(express.static(join(directoryFullName, 'public')))

// Register routes
app.use('/', router)

// Error handler
app.use(function (err, req, res, next) {
  res
    .status(err.status || 500)
    .send(err.message || 'Internal Server Error')
})

// Starts the HTTP server listening for connections.
app.listen(process.env.PORT, () => {
  console.log(`Server running at http://localhost:${process.env.PORT}`)
  console.log('Press Ctrl-C to terminate...')
})
```

WEB APPLICATION

- ✓ The purpose of `server.js` is to create an instance of an Express application, configure it, and start it as Express server.
- ✓ Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle.
  - Application-level: `app.use()` and `app.METHOD()`, `(req, res, next)`).
  - Router-level: same as application-level (use `express.Router()`).
  - Error-handling: an error-handling middleware function has four parameters, `(err, req, res, next)`).
  - Built-in: `express.static`, `express.urlencoded`
  - Third-party: Add functionality.

The Entry Point – `src/server.js`

```
import express from 'express'
import helmet from 'helmet'
import logger from 'morgan'
import { router } from './routes/router.js'
import { connectDB } from './config/mongoose.js'

try {
  await connectDB()

  const app = express()

  app.use(helmet())
  app.use(logger('dev'))
  app.use(express.json())
  app.use('/', router)

  app.use(function (err, req, res, next) {
    err.status = err.status || 500

    if (req.app.get('env') !== 'development') {
      return res
        .status(err.status)
        .json({
          status: err.status,
          message: err.message
        })
    }

    // Development only!
    return res
      .status(err.status)
      .json(err)
  })

  app.listen(process.env.PORT, () => {
    console.log(`Server running at http://localhost:${process.env.PORT}`)
    console.log('Press Ctrl-C to terminate...')
  })
} catch (err) {
  console.error(err)
  process.exitCode = 1
}
```

- ✓ The purpose of `server.js` is to create an instance of an Express application, configure it, and start it as Express server.
- ✓ Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle.
  - Application-level: `app.use()` and `app.METHOD()`, (`req, res, next`)).
  - Router-level: same as application-level (use `express.Router()`).
  - Error-handling: an error-handling middleware function has four parameters, (`err, req, res, next`)).
  - Built-in: `express.json`, `express.urlencoded`
  - Third-party: Add functionality.

The Routing

```
// routes/home-router.js
import express from 'express'
import { HomeController } from '../controllers/home-controller.js'

export const router = express.Router()

const controller = new HomeController()

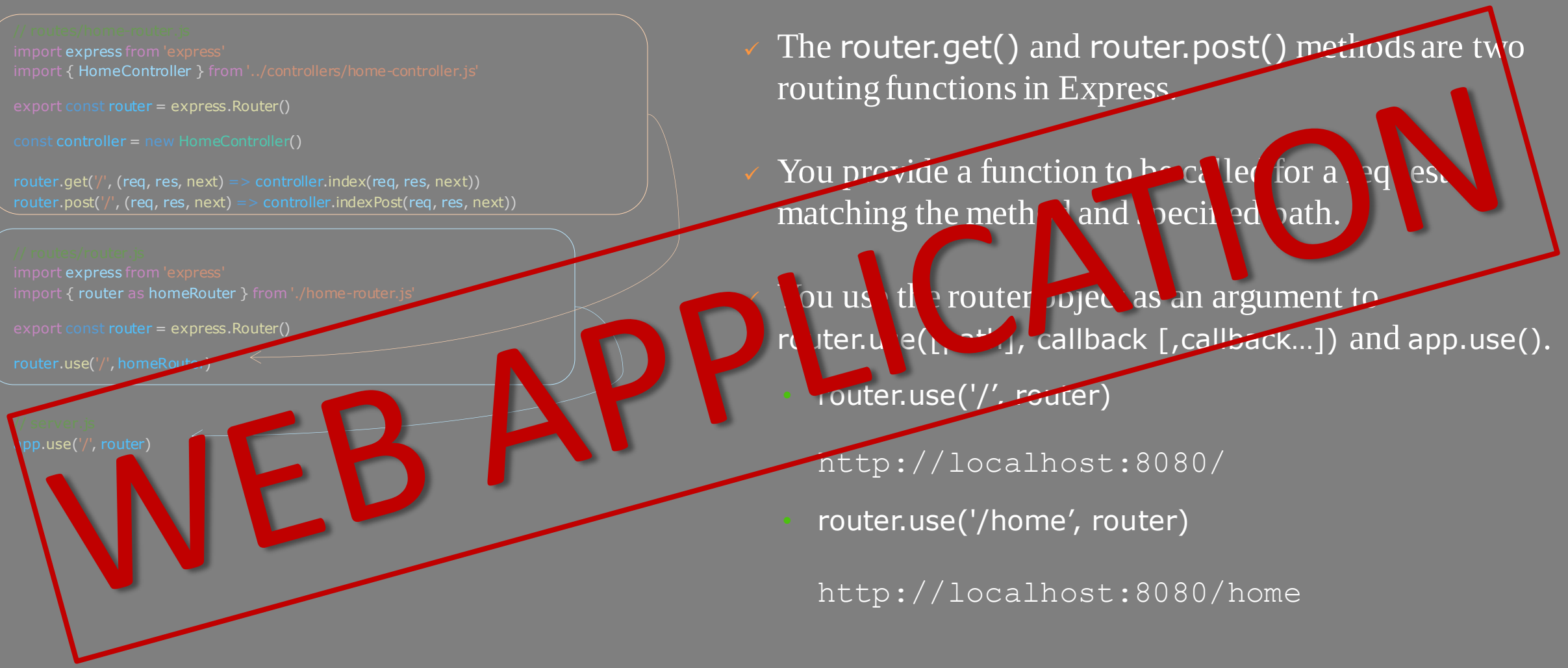
router.get('/', (req, res, next) => controller.index(req, res, next))
router.post('/', (req, res, next) => controller.indexPost(req, res, next))
```

```
// routes/router.js
import express from 'express'
import { router as homeRouter } from './home-router.js'

export const router = express.Router()

router.use('/', homeRouter)
```

```
// server.js
app.use('/', router)
```



- ✓ The router.get() and router.post() methods are two routing functions in Express.
- ✓ You provide a function to be called for a request matching the method and specified path.
- ✓ You use the router object as an argument to router.use([path], callback [,callback...]) and app.use().
  - router.use('/', router)

    `http://localhost:8080/`
  - router.use('/home', router)

    `http://localhost:8080/home`

The Routing

```
// routes/me-router.js
import express from 'express'
import { TasksController } from '../../../controllers/api/tasks-controller.js'
export const router = express.Router()

const controller = new TasksController()

// Provide req.task to the route if :id is present in the route path.
router.param('id', (req, res, next, id) => controller.loadTask(req, res, next, id))

// GET tasks
router.get('/', (req, res, next) => controller.findAll(req, res, next))

// GET tasks/:id
router.get('/:id', (req, res, next) => controller.find(req, res, next))

// POST tasks
router.post('/', (req, res, next) => controller.create(req, res, next))

// PUT tasks/:id
router.put('/:id', (req, res, next) => controller.update(req, res, next))

// DELETE tasks/:id
router.delete('/:id', (req, res, next) => controller.delete(req, res, next))
```

```
// routes/api/v1/router.js
import express from 'express'
import { router as tasksRouter } from './tasks-router.js'
export const router = express.Router()

router.get('/', (req, res) => res.json({ message: 'Hooray! Welcome to version 1 of this very simple RESTful API!' }))
router.use('/tasks', tasksRouter)
```

```
// routes/router.js
import express from 'express'
import createError from 'http-errors'
import { router as v1Router } from './api/v1/router.js'
export const router = express.Router()

router.use('/api/v1', v1Router)

// Catch 404 (ALWAYS keep this as the last route).
router.use('*', (req, res, next) => next(createError(404)))
```

```
// server.js
app.use('/', router)
```

- ✓ The `router.param()` method adds callback triggers to route parameters.
  - When a parameter is present in a route, you may map loading logic to provide the corresponding object to the req object.
- ✓ The `router.METHOD()` methods provide the routing functionality, where `METHOD` is one of the HTTP methods, such as GET, PUT, POST, and so on, in lowercase.
- ✓ You provide a function to be called for a request matching the method and specified path.
- ✓ You use the router object as an argument to `router.use([path], callback [,callback...])` and `app.use()`.

The Controller – `controllers/home-controller.js`

```
import { format } from 'date-fns'

/**
 * Encapsulates a controller.
 */
export class HomeController {
  /**
   * Renders a view and sends the rendered HTML string as an HTTP response.
   * index GET.
   *
   * @param {object} req - Express request object.
   * @param {object} res - Express response object.
   * @param {Function} next - Express next middleware function.
   */
  index (req, res, next) {
    res.render('home/index')
  }

  /**
   * Renders a view, based on posted data,
   * the rendered HTML string as an HTTP response.
   *
   * @param {object} req - Express request object.
   * @param {object} res - Express response object.
   * @param {Function} next - Express next middleware function.
   */
  indexPost (req, res, next) {
    const viewData = {
      name: req.body.name,
      dayName: format(new Date(), 'iiii')
    }

    res.render('home/index', { viewData })
  }
}
```

- ✓ Controller functions...
  - ...get the requested data (if any), ...
  - ...render an HTML page using a template engine, and...
  - ...return HTML to the client.
- ✓ The method `res.render(view [, locals])` renders a view and sends the HTML string to the client.
  - `view`, the path of the view file to render.
  - `locals` (optional), an object that contains data to pass from the controller to the view.

WEB APPLICATION

The Controller – `controllers/api/tasks-controller.js`

```
import createError from 'http-errors'
import { Task } from '../../models/task.js'

export class TasksController {
  async loadTask (req, res, next, id) {
    try {
      const task = await Task.findById(id)

      if (!task) {
        next(createError(404))
        return
      }

      req.task = task

      next()
    } catch (error) {
      next(error)
    }
  }
  // ----------------------------------------
  async find (req, res, next) {
    res.json(req.task)
  }
  // ----------------------------------------
  async findAll (req, res, next) {
    try {
      const tasks = await Task.find()
      res.json(tasks)
    } catch (error) {
      next(error)
    }
  }
  // ----------------------------------------
  async create (req, res, next) {
    try {
      const task = new Task({
        description: req.body.description,
        done: req.body.done
      })

      await task.save()

      const location = new URL(
        `${req.protocol}://${req.get('host')}${req.baseUrl}/${task._id}`
      )

      res
        .location(location.href)
        .status(201)
        .json(task)
    } catch (error) {
      next(error)
    }
  }
  // ----------------------------------------
  async update (req, res, next) {
    try {
      req.task.description = req.body.description
      req.task.done = req.body.done

      await req.task.save()

      res.status(204).end()
    } catch (error) {
      next(error)
    }
  }
  // ----------------------------------------
  async delete (req, res, next) {
    try {
      await req.task.deleteOne()

      res.status(204).end()
    } catch (error) {
      next(error)
    }
  }
}
```

- ✓ The `loadTask()` method provides `req.task` to the route.
  - The `find()`, `update()`, and `delete()` methods can use the Task object `req.task` refers to and do not need to retrieve the object themselves.
- ✓ In the controller, you do not have to think about how a document is converted to JSON because the schema defines it.

```
const schema = new mongoose.Schema({
  description: { ...
  },
  done: { ...
  }
}, {
  timestamps: true,
  toJSON: {
    virtuals: true,
    transform: function (doc, ret) {
      delete ret.__v
      delete ret._id
    }
  }
})

schema.virtual('id').get(function () {
  return this._id.toHexString()
})
```

API Keys

```
IP84UTvzJKds1Jomx8gIbTXcEEJSUilGqpxCcmnx
```

- ✓ Authentication/Authorization?
  - vs. username/password
  - long-lived?
  - Easy to leak public?
  - “API keys are not API Security”
- ✓ Track how the API is being used
  - Use API keys for identification and analytics
  - Rate limit
  - Easy to revoke/renew

Rate Limit



Limit the number of requests to the API

- ✓ X-RateLimit-Limit – Number of calls during the time slot
- ✓ X-RateLimit-Remaining – Number of remaing calls
- ✓ X-RateLimit-Reset – Time when the call counter will be reseted
  - Epoch-/POSIX-/UNIX-tid
- ✓ 429 Too Many Requests

```
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 58
X-RateLimit-Reset: 1486630428
```

Versioning

```
GET http://www.api.example.com/v1/products
GET http://www.api.example.com/v2/products
```

vs.

```
Accept: application/vnd.coursepress-data+json;version=2.0
Accept: application/vnd.github.v3+json
```

vs.

HATEOAS

Filtering/Partial Response

Filtering:

```
GET /users?city=Kalmar
GET /users?type=seller&city=Växjö
```

Partial Response:

```
GET /users/ellen:(firstname,lastname,email)
GET /users/ellen?fields=firstname,lastname,email
```

Pagination:

```
/users?page=3&rpp=25
/users?start=100&count=25
/users?offset=100&limit=25
```

GraphQL

- You can construct complex queries
  - "Get a post and all its comments that are down-voted"
- You will not get data you do not need.



```
{
  hero {
    name
  }
}
```

```
{
  "hero": {
      "name": "Luke Skywalker"
  }
}
```

