

Application folder structure

Application folder structure

- ❖ We previously discussed better practices while developing a real application, where we recommended the use of the **package.json** file over directly installing your modules.
- ❖ However, this was only the beginning; once you continue developing your application, you'll soon find yourself wondering how you should arrange your project files and break them into **logical units** of code.
- ❖ JavaScript, in general, and consequently the Express framework, are agnostic about the structure of your application as you can easily place your entire application in a single JavaScript file.
- ❖ This is because no one expected JavaScript to be a full-stack programming language, but it doesn't mean you shouldn't dedicate special attention to organizing your project.

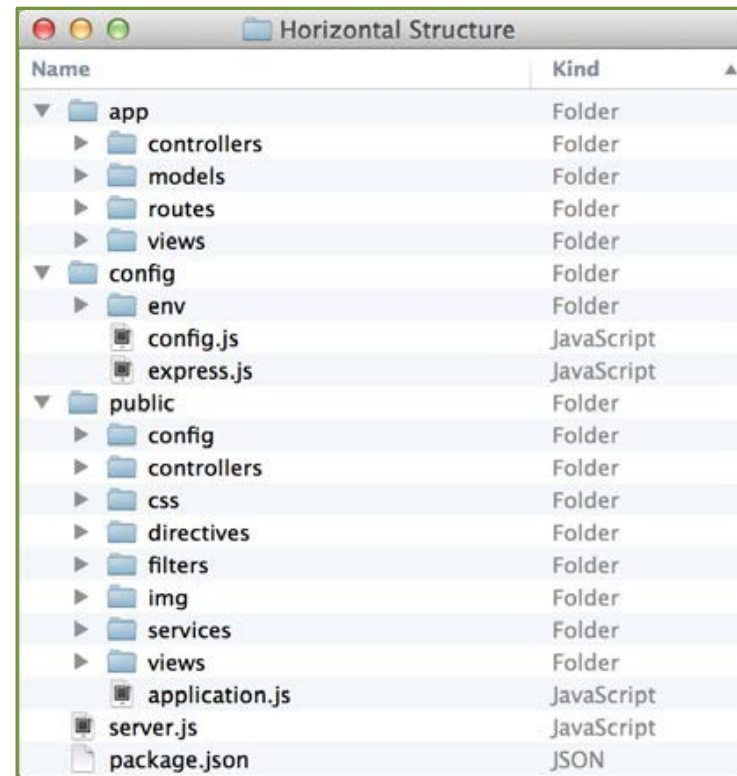
Application folder structure (continued)

- ❖ Since the MEAN stack can be used to build all sorts of applications that vary in size and complexity, it is also possible to handle the project structure in various ways.
- ❖ The decision is often directly related to the estimated complexity of your application.
- ❖ For instance, simple projects may require a **leaner folder structure**, which has the advantage of being clearer and easier to manage, while complex projects will often require a **more complex structure** and a better breakdown of the logic since it will include many features and a bigger team working on the project.
- ❖ It would be reasonable to divide it into two major approaches:
 - a **horizontal** structure for smaller projects
 - a **vertical** structure for feature-rich applications.

Application folder structure (continued)

Horizontal folder structure

- ❖ A **horizontal project structure** is based on the division of folders and files by their **functional role** rather than by the feature they implement, which means that all the application files are placed inside a **main application folder** that contains an MVC folder structure.
- ❖ This also means that there is a **single controllers folder** that contains all of the application controllers, a **single models folder** that contains all of the application models, and so on.
- ❖ An example of the horizontal application structure is as follows:



Application folder structure (continued)

Horizontal folder structure (continued)

- ❖ Let's review the folder structure:
 - The **app** folder is where you keep your Express application logic and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The **controllers** folder is where you keep your Express application controllers
 - The **models** folder is where you keep your Express application models
 - The **routes** folder is where you keep your Express application routing middleware
 - The **views** folder is where you keep your Express application views

Application folder structure (continued)

Horizontal folder structure (continued)

- The **config** folder is where you keep your Express application configuration files. In time you'll add more modules to your application and each module will be configured in a dedicated JavaScript file, which is placed inside this folder.
- Currently, it contains several files and folders, which are as follows:
 - The **env** folder is where you'll keep your Express application environment configuration files
 - The **config.js** file is where you'll configure your Express application
 - The **express.js** file is where you'll initialize your Express application

Application folder structure (continued)

Horizontal folder structure (continued)

- The **public** folder is where you keep your static client-side files and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The **config** folder is where you keep your AngularJS application configuration files
 - The **controllers** folder is where you keep your AngularJS application controllers
 - The **css** folder is where you keep your CSS files
 - The **directives** folder is where you keep your **AngularJS** application directives
 - The **filters** folder is where you keep your AngularJS application filters
 - The **img** folder is where you keep your image files
 - The **views** folder is where you keep your AngularJS application views
 - The **application.js** file is where you initialize your AngularJS application

Application folder structure (continued)

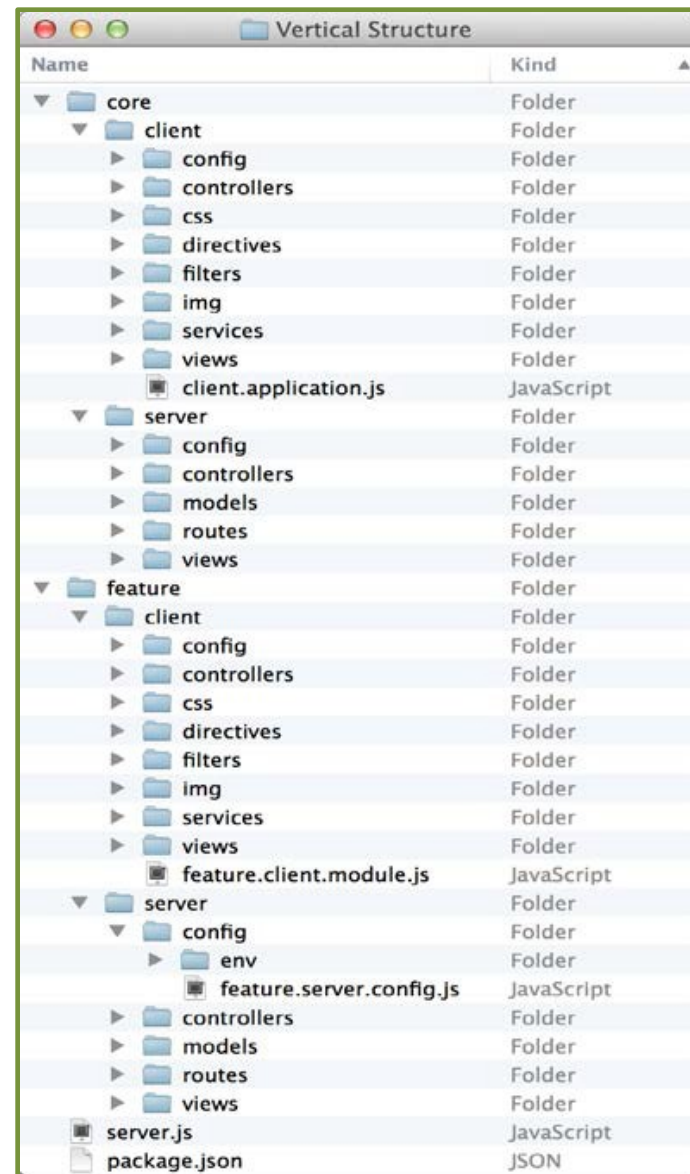
Horizontal folder structure (continued)

- The **package.json** file is the metadata file that helps you to organize your application dependencies.
- The **server.js** file is the main file of your Node.js application, and it will load the **express.js** file as a module to bootstrap your Express application.
- ❖ As you can see, the **horizontal folder structure** is very useful for **small projects** where the number of features is limited, and so files can be conveniently placed inside folders that represent their general roles.
- ❖ Nevertheless, to handle large projects, where you'll have many files that handle certain features, it might be too simplistic.
- ❖ In that case, each folder could be overloaded with too many files, and you'll get lost in the chaos.
- ❖ A better approach would be to use a **vertical folder structure**.

Application folder structure (continued)

Vertical folder structure

- ❖ A **vertical project structure** is based on the division of folders and files **by the feature they implement**, which means each feature has its own autonomous folder that contains an MVC folder structure.
- ❖ An example of the vertical application structure is as follows:



Application folder structure (continued)

Vertical folder structure (continued)

- ❖ As you can see, each **feature** has its own **application-like folder structure**.
- ❖ In this example, we have the **core feature folder** that contains the main application files and the feature folder that include the feature's files.
- ❖ An example feature would be a **user management** feature that includes the authentication and authorization logic.

Application folder structure (continued)

Vertical folder structure (continued)

- ❖ To understand this better, let's review a single feature's folder structure:
 - The **server** folder is where you keep your feature's **server logic** and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The **controllers** folder is where you keep your feature's Express controllers
 - The **models** folder is where you keep your feature's Express models
 - The **routes** folder is where you keep your feature's Express routing middleware

Application folder structure (continued)

Vertical folder structure (continued)

- The **views** folder is where you keep your feature's Express views
- The **config** folder is where you keep your feature's server configuration files.
 - The **env** folder is where you keep your feature's environment server configuration files
 - The **feature.server.config.js** file is where you configure your feature

Application folder structure (continued)

Vertical folder structure (continued)

- The **client** folder is where you keep your feature client-side files and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The **config** folder is where you keep your feature's AngularJS configuration files
 - The **controllers** folder is where you keep your feature's AngularJS controllers
 - The **css** folder is where you keep your feature's CSS files

Application folder structure (continued)

Vertical folder structure (continued)

- The **directives** folder is where you keep your feature's AngularJS directives
 - The **filters** folder is where you keep your feature's AngularJS filters
 - The **img** folder is where you keep your feature's image files
 - The **views** folder is where you keep your feature's AngularJS views
 - The **feature1.client.module.js** file is where you initialize your feature's AngularJS module
- ❖ As you can see, the **vertical folder structure** is very useful for large projects where the number of features is **unlimited** and each feature includes a substantial amount of files.
- ❖ It will allow **large teams to work together** and maintain each feature separately, and it can also be useful to share features between different applications.

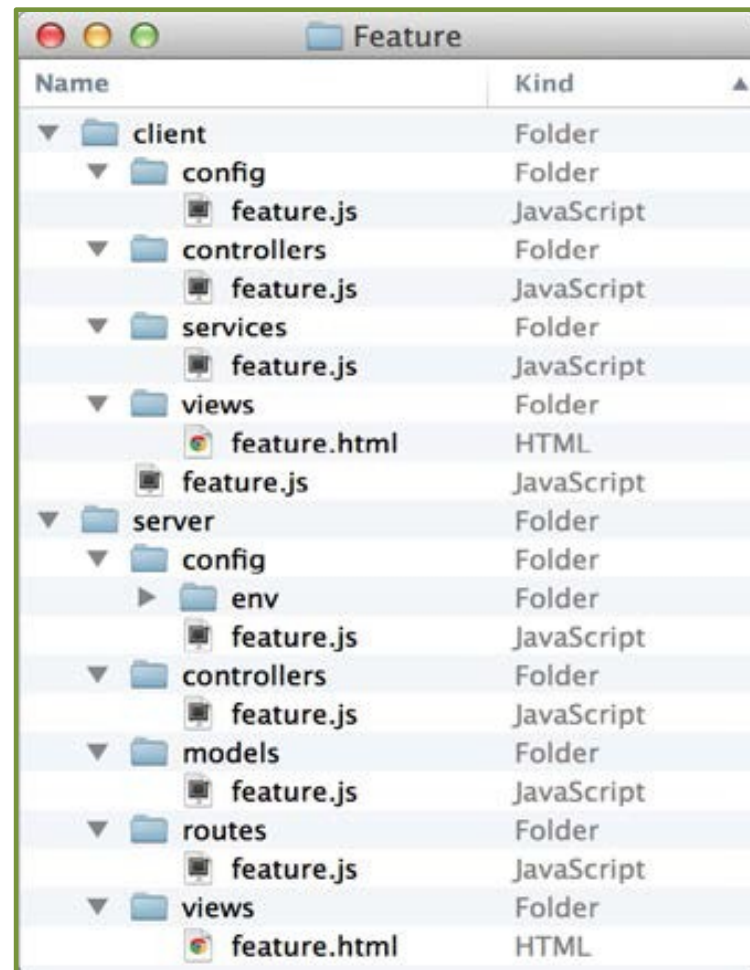
Application folder structure (continued)

- ❖ Although these are two distinctive types of most application structures, the reality is that the MEAN stack can be assembled in many different ways.
- ❖ It's even likely for a team to structure their project in a way that combines these two approaches, so essentially it is up to the project leader to decide which structure to use.
- ❖ In this module, we'll use the **horizontal approach** for reasons of simplicity, but we'll incorporate the AngularJS part of our application in a vertical manner to demonstrate the flexibility of the MEAN stack's structure.
- ❖ Keep in mind that everything presented in this module can be easily restructured to accommodate your project's specifications.

Application folder structure (continued)

File-naming conventions

- ❖ While developing your application, you'll soon notice that you end up with many **files with the same name**.
- ❖ The reason is that MEAN applications often have a **parallel MVC structure** for both the **Express** and **AngularJS** components.
- ❖ To understand this issue, take a look at a common vertical feature's folder structure:
- ❖ This issue can cause some **confusion** for the development team, so to solve this, you'll need to use some sort of a naming convention.



The screenshot shows a file explorer window titled "Feature" with a table-like view of the directory structure. The table has two columns: "Name" and "Kind".

Name	Kind
client	Folder
config	Folder
feature.js	JavaScript
controllers	Folder
feature.js	JavaScript
services	Folder
feature.js	JavaScript
views	Folder
feature.html	HTML
feature.js	JavaScript
server	Folder
config	Folder
env	Folder
feature.js	JavaScript
controllers	Folder
feature.js	JavaScript
models	Folder
feature.js	JavaScript
routes	Folder
feature.js	JavaScript
views	Folder
feature.html	HTML

Application folder structure (continued)

File-naming conventions (continued)

- ❖ The **simplest solution** would be to **add each file's functional role to the file name**, so a feature controller file will be named **feature.controller.js**, a feature model file will be named **feature.model.js**, and so on.
- ❖ However, things get even more complicated when you consider the fact that MEAN applications use JavaScript MVC files for both the Express and AngularJS applications.
- ❖ This means that you'll often have two files with the same name; for instance, a **feature.controller.js** file might be an Express controller or an AngularJS controller.

Application folder structure (continued)

File-naming conventions (continued)

- ❖ To solve this issue, it is also recommended that you extend files names with their execution destination.
- ❖ A simple approach would be to name our Express controller **feature.server.controller.js** and our AngularJS controller **feature.client.controller.js**.
- ❖ This might seem like **overkill** at first, but you'll soon discover that it's quite helpful to quickly identify the role and execution destination of your application files.

Application folder structure (continued)

Implementing the horizontal folder structure

- ❖ To begin structuring your first MEAN project, create a new project folder with the following folders inside it:



Application folder structure (continued)

Implementing the horizontal folder structure (continued)

- ❖ Once you created all the preceding folders, go back to the application's root folder, and create a **package.json** file that contains the following code snippet:

```
{  
  "name" : "MEAN",  
  "version" : "0.0.3",  
  "dependencies" : {  
    "express" : "~4.8.8"  
  }  
}
```

Application folder structure (continued)

Implementing the horizontal folder structure (continued)

- ❖ Once you created all the preceding folders, go back to the application's root folder, and create a **package.json** file that contains the following code snippet:

```
{
  "name" : "MEAN",
  "version" : "0.0.3",
  "dependencies" : {
    "express" : "~4.8.8"
  }
}
```

- ❖ Now, in the app/controllers folder, create a file named **index.server.controller.js** with the following lines of code:

```
exports.render = function(req, res) {
  res.send('Hello World');
};
```

Application folder structure (continued)

Handling request routing

- ❖ Express supports the routing of requests using either the `app.route(path).VERB(callback)` method or the `app.VERB(path, callback)` method, where **VERB** should be replaced with a lowercase HTTP verb.

- ❖ Take a look at the following example:

```
app.get('/', function(req, res) {  
    res.send('This is a GET request');  
});
```

- ❖ This tells **Express** to execute the middleware function for any HTTP request using the **GET** verb and directed to the root path.
- ❖ If you'd like to deal with **POST** requests, your code should be as follows:

```
app.post('/', function(req, res) {  
    res.send('This is a POST request');  
});
```

Application folder structure (continued)

Handling request routing (continued)

- ❖ However, Express also enables you to define a **single route** and then chain several middleware to handle different HTTP requests.
- ❖ This means the preceding code example could also be written as follows:

```
app.route('/').get(function(req, res) {  
    res.send('This is a GET request');  
}).post(function(req, res) {  
    res.send('This is a POST request');  
});
```

Application folder structure (continued)

Handling request routing (continued)

- ❖ Another cool feature of Express is the ability to **chain** several middleware in a **single routing definition**.
- ❖ This means middleware functions will be called in order, passing them to the next middleware so you could determine how to proceed with middleware execution.
- ❖ This is usually used to **validate requests** before executing the **response logic**.

Application folder structure (continued)

Handling request routing (continued)

- ❖ To understand this better, take a look at the following code:

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

var hasName = function(req, res, next) {
  if (req.param('name')) {
    next();
  } else {
    res.send('What is your name?');
  }
};

var sayHello = function(req, res, next) {
  res.send('Hello ' + req.param('name'));
};

var app = express();
app.get('/', hasName, sayHello);

app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Application folder structure (continued)

Handling request routing (continued)

- ❖ In the preceding code, there are two middleware functions named **hasName()** and **sayHello()** .
- ❖ The **hasName()** middleware is looking for the name parameter; if it finds a defined name parameter, it will call the **next** middleware function using the next argument.
- ❖ Otherwise, the **hasName()** middleware will handle the response by itself.
- ❖ In this case, the next middleware function would be the **sayHello()** middleware function.
- ❖ This is possible because we've added the middleware function in a row using the **app.get()** method.

Application folder structure (continued)

Adding the routing file

- ❖ The next file you're going to create is your first routing file. In the **app/routes** folder, create a file named **index.server.routes.js** with the following code snippet:

```
module.exports = function(app) {  
    var index = require('../controllers/index.server.controller');  
    app.get('/', index.render);  
};
```

- ❖ All that you have left to do is to create the Express application object and bootstrap it using the controller and routing modules you just created.
- ❖ To do so, go to the config folder and create a file named **express.js** with the following code snippet:

```
var express = require('express');  
module.exports = function() {  
    var app = express();  
    require('../app/routes/index.server.routes.js')(app);  
    return app;  
};
```

Application folder structure (continued)

Adding the routing file (continued)

- ❖ To finalize your application, you'll need to create a file named **server.js** in the root folder and copy the following code:

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

```
var app = express();  
app.listen(3000);  
module.exports = app;
```

```
console.log('Server running at  
http://localhost:3000/');
```