Contents

[Previous Class Topics 2](#_Toc47200974)

[Class Objectives 3](#_Toc47200975)

[Upcoming topics 3](#_Toc47200976)

[15 Key Takeaways for Good Node 4](#_Toc47200977)

[Rule #1: Correctly organize our files into folders 4](#_Toc47200978)

[Rule #2: Keep a clear separation between the business logic and the API routes 4](#_Toc47200979)

[Rule #3: Use a service layer 4](#_Toc47200980)

[Rule #4: Use a config folder for configuration files 7](#_Toc47200981)

[Rule #5: Have a scripts folder for long npm scripts 8](#_Toc47200982)

[Rule #6: Use dependency injection 8](#_Toc47200983)

[Rule #7: Use unit testing 9](#_Toc47200984)

[Rule #8: Use another layer for third-party services calls 10](#_Toc47200985)

[Rule #9: Use a linter 11](#_Toc47200986)

[Rule #10: Use a style guide 11](#_Toc47200987)

[Rule #11: Always comment your code 12](#_Toc47200988)

[Rule #12: Keep an eye on your file sizes 12](#_Toc47200989)

[Rule #13: Always use gzip compression 12](#_Toc47200990)

[Rule #14: Use promises 13](#_Toc47200991)

[Rule #15: Use promises’ error handling support 14](#_Toc47200992)

Class : Learning Node For Everyone

Start Date : 07/18/2020

Delayed class from July 4th holiday

## Previous Class Topics

* Introduction
  + What You Need
  + What is Node
  + Resources
  + Contacting Instructors
  + Class structure
* Getting Started
  + Installing
  + Setting up
    - Node
    - Mongo DB
    - MySQL
    - Editors (IDE)
* Developing Code (all languages)
  + Introduction to Concepts and Tools
  + Best Practices
* How node works
  + Your first server
    - Testing your first server
  + REPL Terminal
    - REPL commands
    - Using REPL
  + Online editor
    - Using online editor
* Your first homework
  + Starter code
  + Assignment
* Run maze in browser tester
* Talk about comments
* Show developer tools in Chrome
* Assignment

## Class Objectives

* Homework
* Correctly building a NODE project
  + 15 key takeaways for a good NODE project
  + Building our first server node application project. (using starter code)

## Upcoming topics

* Async and Node
* DB with Node (Mongo & MySQL)
* Programming to handle errors
* Securing your code
* Modeling Data
* Using PUG
* Advanced Concepts

# 15 Key Takeaways for Good Node

## Rule #1: Correctly organize our files into folders

Everything has to have its place in a good application, and a folder is the perfect place to group common elements. Compilers and often languages too sometimes require this. In particular, we want to define a very important separation, which brings us to rule number #2:

## Rule #2: Keep a clear separation between the business logic and the API routes

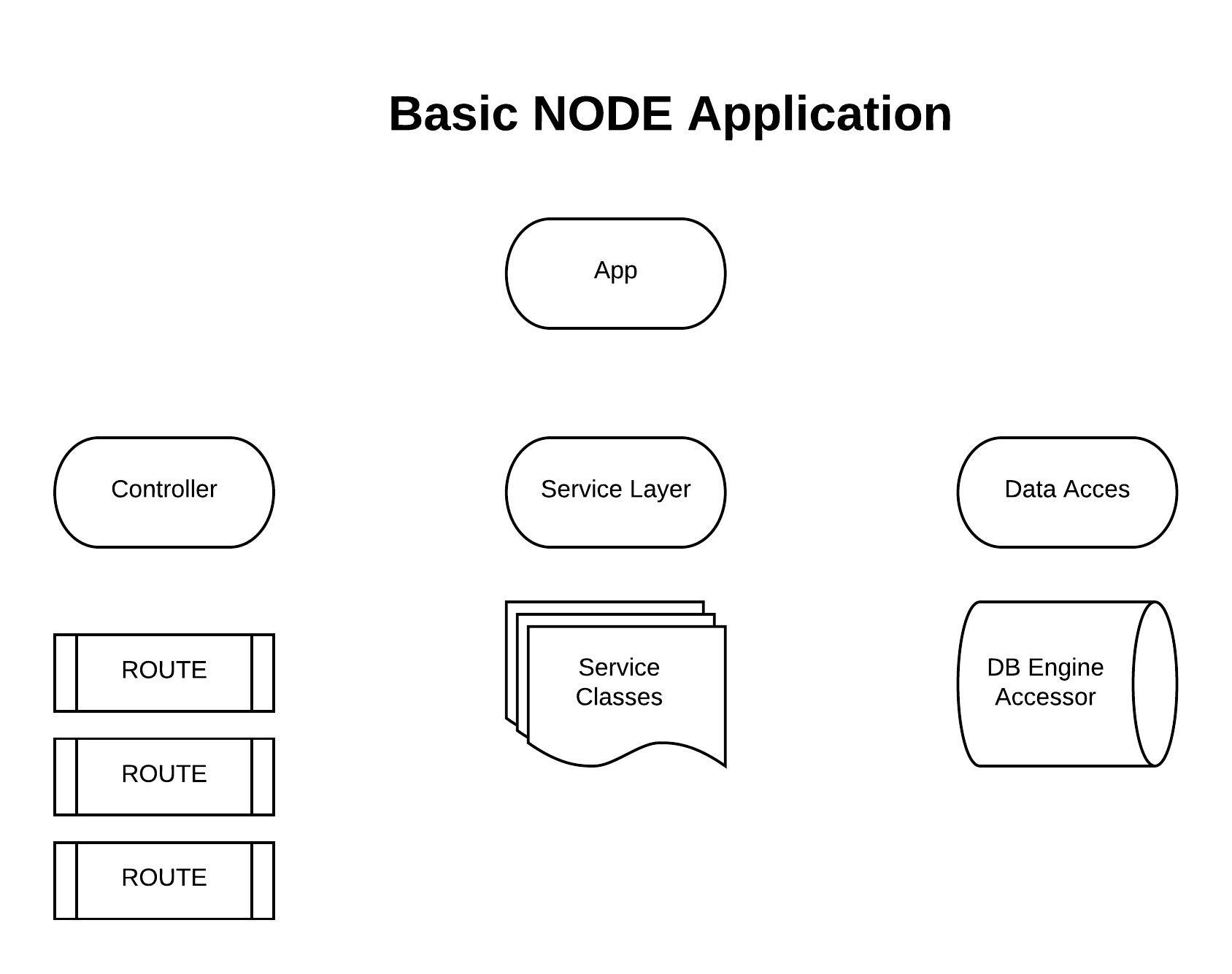
See, frameworks like Express.js are amazing. They provide an incredible set of features for managing requests, views, and routes. With such support, it might be tempting for us to put our business logic into our API routes. But this will quickly make them into giant, monolithic blocks.

This will also damage the testability of the applicationAt this point, you might be wondering, “How do we solve this problem, then? Where can I put my business logic in a clear and intelligent way?” The answer is revealed in rule number #3.

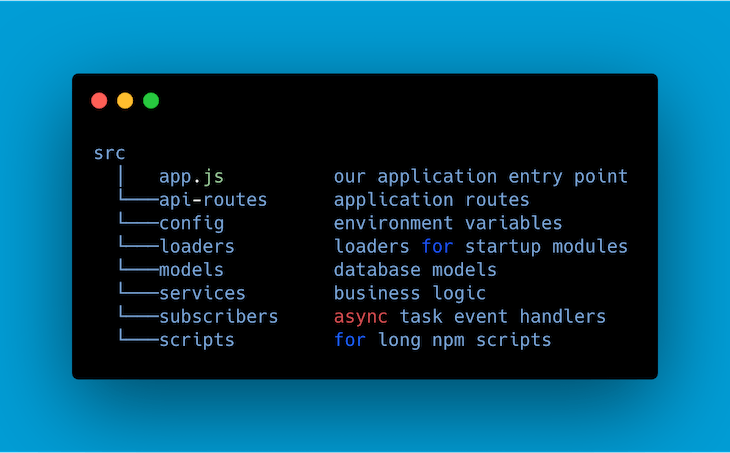
## Rule #3: Use a service layer

This is the place where all our business logic should live. It’s basically a collection of classes, each with its methods, that will be implementing our app’s core logic. The only part you should ignore in this layer is the one that accesses the database; that should be managed by the data access layer.

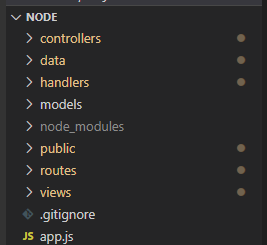
Now that we have defined these three initial rules, we can graphically represent the result like this:



And the subsequent folder structure sending us back to rule #1 can then become:

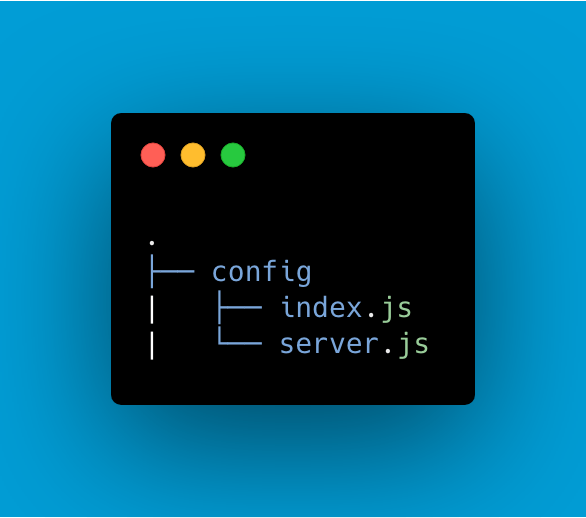


This is a real-world App I recently worked on. This is the structure I built. So you can see naming is not exact. But topically it is obvious. Note that the structure of these folders now includes a folder called “node\_modules”. Your node applications will always have this.

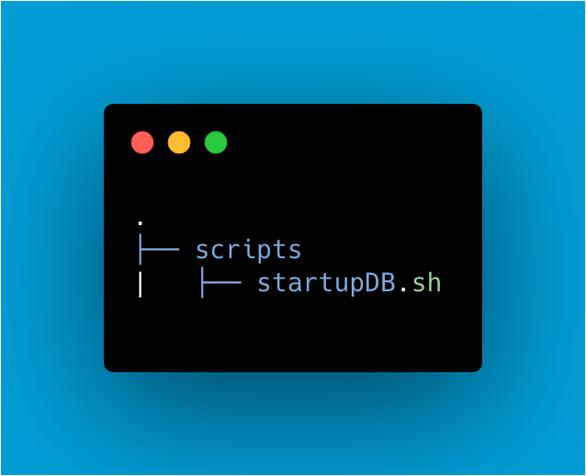


By looking at this last image, we can also establish two other rules when thinking about folder structure.

## Rule #4: Use a config folder for configuration files



## Rule #5: Have a scripts folder for long npm scripts



## Rule #6: Use dependency injection

Node.js has features and tools to make our lives easier. However, as we know, working with dependencies can be quite troublesome most of the time due to problems that can arise with testability and code manageability.

There is a solution for that, and it’s called dependency injection.

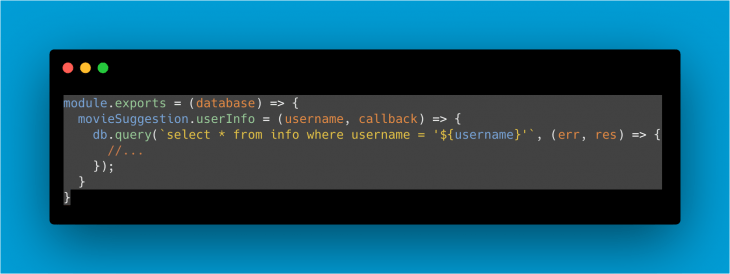
*Dependency injection is a software design pattern in which one or more dependencies (or services) are injected, or passed by reference, into a dependent object.*

By using this inside our Node applications, we:

* **Have an easier unit testing process**, passing dependencies directly to the modules we would like to use instead of hardcoding them
* **Avoid useless modules coupling**, making maintenance much easier
* **Provide a faster git flow**. After we defined our interfaces, they will stay like that, so we can avoid any merge conflicts.



Simple but still not very flexible as an approach to our code. What happens if we want to alter this test to use an example database? We should alter our code to adapt it to this new need. Why not pass the database directly as a dependency instead?



## Rule #7: Use unit testing

Now that we know we have got dependency injection under our belt, we can also implement unit testing for our project.

A common way to test our applications is to test them by units, the goal of which is to isolate a section of code and verify its correctness. When it comes to procedural programming, a unit may be an individual function or procedure. This process is usually performed by the developers who write the code.

Benefits of this approach include:

#### Improved code quality

Unit testing improves the quality of your code, helping you to identify problems you might have missed before the code goes on to other stages of development. It will expose the edge cases and makes you write better overall code

#### Bugs are found earlier

Issues here are found at a very early stage. Since the tests are going to be performed by the developer who wrote the code, bugs will be found earlier, and you will be able to avoid the extremely time-consuming process of debugging

#### Cost reduction

Fewer flaws in the application means less time spent debugging it, and less time spent debugging it means less money spent on the project. Time here is an especially critical factor since this precious unit can now be allocated to develop new features for our product.

In my opinion you can use your unit test to insure you have the design and functionality covered and don’t forget minor things that the application should be doing.

## Rule #8: Use another layer for third-party services calls

Often, in our application, we may want to call a third-party service to retrieve certain data or perform some operations. And still, very often, if we don’t separate this call into another specific layer, we might run into an out-of-control piece of code that has become too big to manage.

A common way to solve this problem is to use the pub/sub pattern. This mechanism is a messaging pattern where we have entities sending messages called publishers, and entities receiving them called subscribers.

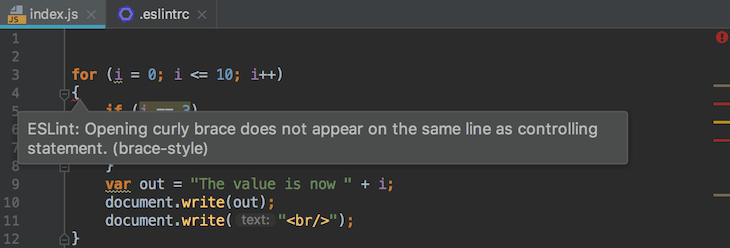
Publishers won’t program the messages to be sent directly to specific receivers. Instead, they will categorize published messages into specific classes without knowledge of which subscribers, if any, may be dealing with them.

In a similar way, the subscribers will express interest in dealing with one or more classes and only receive messages that are of interest to them — all without knowledge of which publishers are out there.

The publish-subscribe model enables event-driven architectures and asynchronous parallel processing while improving performance, reliability, and scalability.

## Rule #9: Use a linter

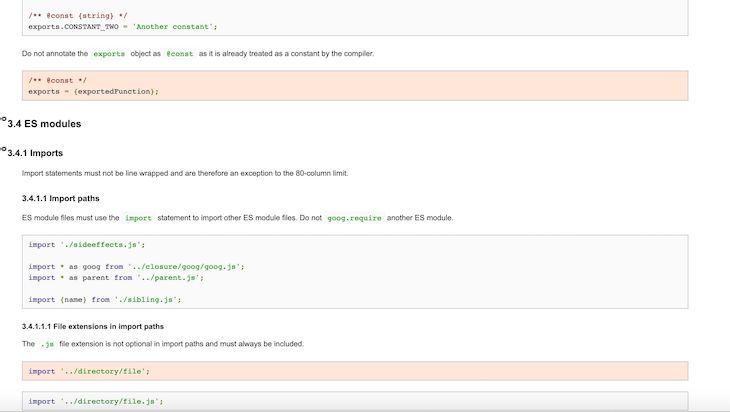
This simple tool will help you to perform a faster and overall better development process, helping you to keep an eye on small errors while keeping the entire application code uniform.



ESLint is a common one and my favorite. <https://eslint.org/>

## Rule #10: Use a style guide

Still thinking about how to properly format your code in a consistent way? Why not adapt one of the amazing style guides that [Google](https://google.github.io/styleguide/jsguide.html) or [Airbnb](https://github.com/airbnb/javascript) have provided to us?



## Rule #11: Always comment your code

Writing a difficult piece of code where it’s difficult to understand what you are doing and, most of all, why? Never forget to comment it. This will become extremely useful for your fellow developers and to your future self, all of whom will be wondering why exactly you did something six months after you first wrote it.

## Rule #12: Keep an eye on your file sizes

Files that are too long are extremely hard to manage and maintain. Always keep an eye on your file length, and if they become too long, try to split them into modules packed in a folder as files that are related together. In other words sometimes splitting a file solely because it is too big will force you to think about the logic it contains and is it really only one topic.

## Rule #13: Always use gzip compression

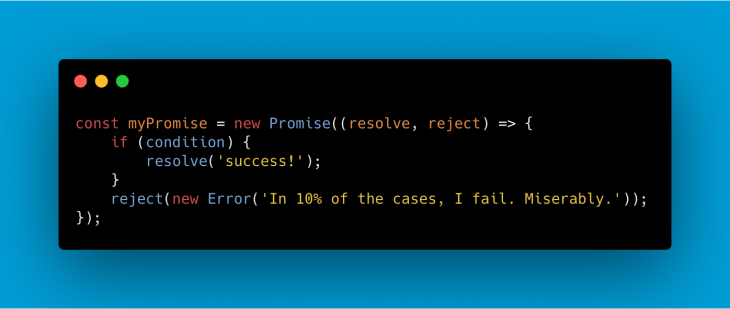
The server can use gzip compression to reduce file sizes before sending them to a web browser. This will reduce latency and lag.



## Rule #14: Use promises

Using callbacks is the simplest possible mechanism for handling your asynchronous code in JavaScript. However, raw callbacks often sacrifice the application control flow, error handling, and semantics that were so familiar to us when using synchronous code. A solution for that is using promises in Node.js.

Promises bring in more pros than cons by making our code easier to read and test while still providing functional programming semantics together with a better error-handling platform.



## Rule #15: Use promises’ error handling support

Finding yourself in a situation where you have an unexpected error or behavior in your app is not at all pleasant, I can guarantee. Errors are impossible to avoid when writing our code. That’s simply part of being human.

Dealing with them is our responsibility, and we should always not only use promises in our applications, but also make use of their error handling support provided by the catch keyword. The following shows it as an arrow function format.

