<https://github.com/Sairyss/backend-best-practices#architecture>

1.File structure

Back end:

dist(compiled files which converte from typescript to javascript)

src

controllers(Restfull API)

models

test(test each function)

functions (get, delete, update, search function)

repositories

app.ts

server.ts

document

.env

Rule convention: follow Mr. Thang

**#1 — Meaningful Naming for Variables & Classes**

* **Use intention-revealing names.**

✖ int d;

✔ int elapsedTimeInDays;

✔ int daysSinceCreatedDate;

✔ int daysSinceLastModifiedDate;

✖ function calc (int num1, int num2) {return num1\*num2}

✔ function multiply (int num1, int num2) {return num1\*num2}

* **Make meaningful distinctions.**

**Avoid ordering / number series naming**

✖ string date1, date2;

✔ string startDate, endDate;

**Avoid mispelling**

✖ float pft, profit

✔ float profitBeforeTax, profitAfterTax

**Avoid synonyms / closely-related terms**

✖ let productData, productInfo

✔ let productData, productDescription

* **Choose descriptive and unambiguous names.**

✖ string sd, ed;

✔ string startDate, endDate;

* **Use pronounceable names.**

✖ float chqRetVal

✔ float chequeReturnsValue

* **Use searchable names.**

**Replace magic numbers with named constants**

✖ 7

✔ int MAX\_CLASSES\_PER\_STUDENT = 7

* **Use comments to reveal additional information such as complex business logic, behaviours, assumptions, future suggestions etc.**

✖ string requestId;

✔ string requestId; // must be unique

* **Use nouns for classes, packages, and variables.**

✖ class HandleAccounts {...}

✔ class AccountsHandler {...}

* **Use verbs/verb phrases for functions.**

✖ function accounts() {...}

✔ function getAccounts() {...}

* **Pick a single word per concept and use it consistently throughout the source code.**

✖ function findUserByID(string id) {...}

✖ function getRoleByKey(string key) {...}

✖ function fetchAllUsers() {...}

✖ function findRoles() {...}

✔ function getUserById(string id) {...}

✔ function getRoleById(string id) {...}

✔ function getAllUsers() {...}

✔ function getAllRoles() {...}

* **Prioritize using domain-related terms.**

✖ transactionsStorage

✔ transactionsCache

* **Avoid dis-information.**

**Avoid using a word with an accepted meaning for something else**

✖ User[] activeUsersList;

✔ User[] activeUsers;

* **Avoid using implementation details** (such as data structure, container type, length) **in the variable name.**

✖ User[] activeUsersArray;

✔ User[] activeUsers;

✖ User[] fiveAdminUsers;

✔ User[] adminUsers;

* **Avoid unpopular acronyms and other names that don’t make sense.**

✖ float topm, topw;

✔ float turnoverPerMonth, turnoverPerWeek;

* **Avoid using comments for introducing variables/functions.**

✖ int d; // elapsed time in days

✔ int elapsedTimeInDays;

* **Avoid using confusing letters for names** (e.g. l (simple L), I (capital i), O (capital O)).

✖ int l, O;

✔ int limit, option;

* **Don’t encode additional details in names.**

**Hungarian notation to encode data type is bad. Instead, use types.**

✖ var strFirstName;

✔ string firstName;

✖ var nUsers;

✔ int usersCount;

* **Avoid meaningless prefixes or suffixes.**

✖ var orderInfo;

✔ var order;

✖ var userData;

✔ var user;

* **Related classes and variables should follow a common pattern.**

✖ UserDataService, UserRecordsDAO, UserInfoExporter

✔ UserService, UserDAO, UserExporter

* **Follow a common case convention. (Using camel case)**

**Method: begin with a lower case letter (camelCase)**

> getHelloWorldWithSnow()

**#2 — Functions**

* **A function should be readable from top to bottom, as a paragraph.**
* **A set of functions should be readable from top to bottom, as a set of paragraphs.**
* **Keep functions small — short** (Ideal: 4 lines, Maximum: 60 lines, the whole function must fit into the screen, so that it’s easy to read without vertical scrolling) and **not too wide** (Up to 70 to 120 characters, it’s easy to read the whole line without horizontal scrolling).
* **Less number of arguments (≤3) is better.** Make argument lists/objects/object arrays to pass large data chunks into functions.
* **A function should remain at a constant level of abstraction.** It shouldn’t deal with both low level and high-level stuff at the same time.
* **Avoid duplications** (DRY — Don’t Repeat Yourself).
* **A function should do only one thing** (Keep it atomic). **Avoid causing side effects** (Do what its name suggests and nothing else).
* **Error handling is one thing. Keep functions that do only that.** Start them with try and end with catch and/or finally.
* **Use try/ catch instead of conditions if possible** (Asking for forgiveness is easier than requesting permission).
* **At lower levels, throw exceptions instead of returning error codes.**
* **Avoid nested control structures.** Replace such scenarios with functions or alternative strategies.
* **Avoid switchstatements** (Hint: use polymorphism and bury the switch statement in an abstract factory).
* **It’s better to have many functions than to pass some code into a function to select a behaviour.**
* **Avoid using boolean variables as function arguments.** Using booleans with other arguments usually means that the function does more than one thing. Always split such functions into smaller functions.

✖ function getUsers(boolean status) {...}

✔ function getActiveUsers() {...}

✔ function getInactiveUsers() {...}

✖ function removeOrders(int id, boolean cleanCache, boolean updateLog) {...}

✔ function removeOrders(int id) {...}

✔ function cleanOrdersCache(int id) {...}

✔ function updateOrdersLog(int id) {...}

* **Avoid output arguments** (Hint: If returning something is not enough, then your function is probably doing more than one thing).

**Pass by reference for the purpose of modification is bad.**

✖ activate(user);

✔ user.activate();

* **A function can be a command or a query, but not both.**

**A setter is a command function. It should not return any value.**

✖ function setRole(...): boolean {...}

✔ function setRole(...) {...}

* **A boolean function must answer yes/no.**

✔ function isActiveUser(...): boolean {...}

✔ function usersAreValid(...): boolean {...}

* **Keep boolean functions in a positive tone.**

✖ function isInactive(): boolean {...}

✔ function isActive(): boolean {...}

**#3 — Code Organization & Formatting**

* **Group code by their functionality.** Related functions must be close. Related code must appear vertically dense.
* **Declare temporary variables close to their usage.**
* **Adapt company/team-wide code conventions. Agree on a common standard for code-formatting.** Either use an external tool or IDE options for auto-formatting.
* **Avoid too long files.** 1000–2000 lines are okay. Shorter, the better.
* **Avoid too-wide code lines**. Make sure they fit into the screen (Up to 70 to 120 characters). If they don’t fit, try to split them into multiple lines.
* **Write high-level code first in a file and keep lower-level implementations towards the end of the file.** (Good files are like newspaper articles, with a heading, the important stuff first, and details later)
* **Make sure the code is ordered as of the calling sequence.** The caller should be before the callee.
* **Follow proper indentation across code files.** Use Prettier extension in VS Code for formatting

**#4 — Objects and Data Structures**

* **A data structure must expose data while having no behavior.**
* **An object must hide data while exposing the behaviors** (via methods).
* **Avoid accessing unknown data via method chaining**. Only access the immediate methods (methods of own class, of objects, just created), immediate parameters (of instance variables or objects just created), but not further methods/parameters through chaining them (a.k.a. Law of Demeter / Principle of Least Knowledge)
* **Functions and data structures can have some coupling by nature**. It may be easier to add new functions than changing data structures (adding new data structures can cause breaking changes in functions).
* **Functions and classes can have some coupling by nature**. It may be easier to add new classes than changing functions (adding new functions can cause breaking changes in classes).

**#5 — Error Handling**

* **At lower levels, throw exceptions instead of returning error codes.** Keep error codes for communication between different layers, interfaces, or systems.
* **Errors and exceptions must provide adequate context** (such as intent, failure stage, error type, failed values etc.)
* **Map foreign errors to adhere to common standards.** Wrap all errors and exceptions raised from external systems, third-party libraries, and APIs. Use a generic error type for unknown/unhandled cases.
* **Reduce the reliance on return type and null checks** (since they strongly couple failure path information to main flow). Instead of returning null/false, return a proper error object or throw an exception. Passing null to a function / next stage is also bad, avoid such practices.

**#6 — Boundaries**

* **Map foreign behaviours into wrapper classes when integrating external systems and third-party code.** Expose only limited capability for local use.
* **Use the Adapter pattern to handle things that don’t exist** (e.g. external systems not yet implemented / integrated). Make adapters consume the agreed API via proper use of interfaces. When a foreign API changes, only the adapter must change, without any major impact to local code.

**#7 — Classes & Interfaces**

* **Inside a class/interface, arrange code in order**: static variables in order of visibility (start from public) → instance variables in order of visibility (start from public) → constructors → methods grouped by functionality → getters, setters, equals, variable conversions, utility & helper methods etc.
* **Encapsulate utility methods**. Make them private (or protected if they are exposed to tests).
* **Make classes small**. Shorter, the better (Ideal: 100 lines, Maximum: 1000 lines).
* **Classes having exactly one responsibility and one reason to change are easy to manage.** a.k.a. The Single Responsibility Principle.
* **The class name must show its responsibility.** Class names with Processor, Manager, Super often hints unfortunate aggregation of responsibilities.
* **Multiple small classes are preferred over very few large classes.** During change requests, such code reduces the affected code surface due to high separation of concerns.
* **Organize code based on abstractions, not concrete implementation details**. Build abstract agreements to facilitating interactions between code.
* **Maintain high cohesion**. In general, classes have a fewer number of instance variables and they are co-dependant with methods as a whole. If that cohesion is getting low, break the class apart.
* **Too much inheritance is bad**. Try to use composition more often.

**#8 — Concurrency**

* **Keep concurrency-related code separate from other code.**
* **Severely limit access to any data that may be shared**. Unless there’re performance concerns, using copies of data is more preferred than sharing.
* **Try to maintain data as independent subsets that can be processed by separate threads**.
* **Avoid using more than one Synchronized method on a shared object**.
* **Keep synchronized sections as small as possible.**
* **Think about shut-down early and get it working early.**

**#9 — Comments**

* **Express yourself in the code, but not in the comments**. Keep comments for delivering any extra information. If the code is readable, the need for comments is very low.
* **Warn the consequences of potential scenarios/actions**. If the code is too brittle, instead of commenting, fix the code.
* **Avoid noise and weak comments** (negativity, complaints, humour, misleading opinions, redundant statements, changelog, too much information, irrelevant facts, comments needing additional explanations).
* **Avoid commenting out code chunks.** If code is bad, rewrite it.Use version control systems for handling code that is expired, deprecating, or experimental. (Protip: If you comment out code for a very short term, consider using a TODO comment as a reminder to clean things up later).

**#10 — Application Logging**

* **Follow a common logging practice across code.**
* **Use correct log levels and provide the adequate context in log messages** (e.g. timestamp, log level, code location, state information, parameter values, error descriptions, system banners)**.**
* **Do not log sensitive information** (e.g. personally identifiable information, financial values, business data, auth info).