Fundamentals of Backend Architecture Design

All information systems have a similar "spine": data storage, a mechanism for processing them, and input/output of data. Let's illustrate this with an unusual example, such as a computer game. It has an entirely similar structure. Game data includes files with graphics, sound, settings files, and user data (saved games). The player controls the program using the keyboard and mouse or joystick. This is the program input. The program's output is a video on the monitor or TV screen accompanied by sound. This is the result that the game is launched for. The combination of video and sound creates dramatic images, empathy, and involvement in the human imagination.

Of course, a computer game is very different from an accounting program. Despite this, as you can see from the example above, the program layout will be the same.

The part facing the user is called the Frontend. The hidden part, where the data is stored and processed, is called the Backend. If we take the theater as a metaphor, the stage is the Frontend, and the space behind the scenes is the Backend. We can develop the metaphor and confidently state that the Backend is the entire theater, minus the stage. Indeed, to ensure a performance, you need electricity, water supply, dressing rooms for the actors, a room for the prompter, doors with locks and fire extinguishers (security measures), accounting, and the director's office.

The same thing happens with distributed applications. Only one part of them is addressed to the user. And the other parts are hidden from him.

It often happens that one Backend works with several Frontends. It is like a focal point, the center of the composition. For example, an application can have several "clients": a web interface, mobile interfaces for Android and iOS. When designing a Backend, consider the principles discussed below, which will be demonstrated with an example.

However, first we need to resolve a possible misunderstanding from mixing the terms Backend and API. Strictly speaking, the Backend is any part of the application that processes user commands and manipulates data. API (Application Programming Interface) is, in fact, an interface through which programs can communicate with each other.

Suppose the communication between a person and a program requires input and output means (a keyboard and a monitor or printer, respectively). In that case, programs can communicate with each other via API. Not every Backend has an API - this is an optional part. Nowadays, developers try to provide APIs to facilitate the integration of systems.

General principles

I recommend building information systems based on the data they must process. You need to decide what data the system stores, who adds it and how, how it can be changed, and how it can be deleted.

Knowing the input and output methods, you can decide on the so-called "endpoints", that is, methods that clients (one or another Frontend or third-party program) can call to manipulate the data. Here, you also need to provide security measures. For example, if the data is not public, then only an authorized user can see it.

Endpoints should hide the implementation details from the user (encapsulation principle). The implementation of the method can change over time, reflecting the development of the system, but the Backend user remains unaware of the internal changes.

Method names should be concise and clear, reflecting the purpose of the method. Additional information can be provided in the documentation or the Swagger help page.

The experience accumulated by developers over decades is expressed in the concepts of DRY and SOLID.

DRY principles

DRY stands for "do not repeat yourself". I would say that this is the key principle of a proper software development culture. Just as each part in a clock mechanism has its place, so in your code, the elements should be exactly where they are needed, without redundancy, and without conflicting with each other for areas of responsibility. Let's consider this in more detail.

Avoid copy-pasting

I have seen this happen more than once, where the same code is copied multiple times to different parts of the system with only minor changes. Modern IDEs can detect and highlight such areas. Pay attention to this. If the same code is required at least twice, it makes sense to separate it into a function (and decide where exactly you will place this function). If the code should be slightly different - no problem. Use function arguments to customize its behavior. It happens that after a series of changes, the function grows with more and more branches, and the number of arguments increases. At some point, this function needs to be split into several.

If the team does not pay attention to this bad pattern, then the code becomes difficult to maintain. After all, you need to identify all the places of copying and fix them all.

Code maintenance

Code divided into functions is easier to maintain. Of course, when the program is extensive, programmers may simply not know that the function they need already exists, and write a similar one of their own. Typically, during the code review stage, a senior programmer or another employee responsible for code integrity should identify the problem and work together to resolve it.

Try to make functions pure. Let me remind you that a function is called "pure" if its result depends only on the input arguments. This makes it much easier to maintain and test, including by automated means.

Functions and objects

Some programming languages support separate functions and classes at the same time (for example, Python). If there is such a choice, consider where it would be better to place some of the functionality. Imagine you have a code that prints the state of any entity to the console. Perhaps it should be incorporated into the class and inherited by all descendants to account for the features? An alternative is to create functions of the same name (if the language supports polymorphism) with arguments of different types. Remember the principles of encapsulation. How will it be easier to use your code? The easiest approach is when the implementation details are hidden, so another engineer doesn't have to figure out which function to call or where to find it to print the object's state.

The DRY principle in databases

The same principle is good for database design. If you are not familiar, please read about database normalization and Boyce-Codd normal form.

The essence of the third normal form is that a table should not contain duplicate information that is already present in another table. For example, if the first table contains information about employees including the year of birth, then there should be no "year of birth" column in the other table.

There are exceptions to this rule. Sometimes, system performance suffers from an excessive number of “joins” between tables (all sorts of inner join, outer join, cross join). The database is designed with an eye on the third normal form, and this is correct. However, the data is overwhelming, or the queries are too "heavy," which leaves stakeholders dissatisfied with the results.

The oldest, historical way to get out of the situation is denormalization. This is when columns are deliberately introduced into the table that are already present in other tables. The number of links between tables in the query decreases, and performance increases. However, there are preferable methods, such as indexes, View, Materialized Views, and Stored procedures. A detailed discussion of them is beyond the scope of this book.

SOLID principles

SOLID is another acronym. Let's break it down in detail.

S – Single Responsibility Principle. We discussed it in detail above in the context of DRY.

O – Open/Closed principle. The principle states that the system architecture should be open for extension, but not for modification. In other words, additional modules can be written for the system, but the core of the system remains unchanged. A good example is Node.js. If the system is open source, then it is possible to change the core through branching.

L – Liskov Substitution Principle. This principle applies to object-oriented programming languages. The point is that the descendant class should not radically change the behavior inherited from the ancestor class. Be careful about side effects. This is when a method performs an action in the background, such as saving files or writing to the system registry. If the ancestor did not do this, and the descendant suddenly began to do so, then this may confuse the programmer. In this case, it is better to create another version of the method with a new name and new functionality.

I – Interface Segregation Principle. Often, users of your code are forced to create their own classes that implement your interfaces. Make it convenient for them: let the interface be responsible for one function, according to the principles described above.

D – Dependency Inversion Principle. First, you need to define interfaces (essentially, abstractions) and not a specific implementation in classes. The implementation of functions should also rely on interfaces, and not assumptions about a particular implementation.

Service layers

A common mistake is to place all the code in one module. As a rule, the functions called by the client are in the one. Such code is difficult to maintain and develop. It is often called "spaghetti code" because various semantic layers, and even code in different languages, are intricately intertwined.

I have seen a situation where HTML is generated on the server using Java code, and right here, right in the markup, queries to the database are executed in SQL. And depending on the results, one or another JavaScript is substituted into the markup.

Of course, such cases should be avoided. Usually, the Backend functionality is divided into logical layers: data storage, data access, external services, auxiliary functions, business logic, and transport layer. Let's look at each of them in more detail.

Data Storage. This is where data is stored. As a rule, the storage exists separately from the code. Most often, this is a database, but there can also be static files. Some systems do not have long-term data storage, relying on programs that process data on the fly. For example, it comes from meteorological sensors.

Data Access. Mechanisms for working with data. It is preferable to make them as abstract as possible. Suppose in the future it is necessary to change the data source. In that case, this will be much easier if the implementation details of the connection between the application and the data source are well encapsulated.

External Services. Perhaps our application receives data not only from our storage, but also from other places, for example, through an API. The mechanisms for obtaining this data should be placed in a separate place. The principle here is the same as in the Data Access Layer. If one data provider needs to be replaced by another, it is easier to do so with proper encapsulation. Sometimes, both layers are combined. After all, the essence is the same - to receive and write data.

Auxiliary Functions. All kinds of helpers, functions for converting objects, their serialization, and other things are collected in one place.

Business Logic. Here are the data handlers according to the application's design. When creating them, you should keep in mind the principles of good code outlined above. Data processing should be covered with unit tests.

Transport Layer. Here are the tools of the selected programming language/technology used to communicate with the outside world (if by this we mean communication with the client). You should not mix the transport layer with data receipt or data processing. They have their place. In this layer, we accept input arguments, verify them and the user's credentials, process the data, and return the result.

Common Services

Different applications have common needs: authentication & authorization, configuration, error handling, and logging. As a rule, mature technologies already have mechanisms for solving these problems. In other words, you don't need to solve them yourself. The specific implementation varies greatly from technology to technology. Here we will briefly outline each of them.

Authentication. This is the process of establishing the user's identity. It is necessary to understand that the system request is made by the right person or system (if we are talking about programmatic access via API).

Authorization. This is the next step after authentication. Once the identity is determined, it is necessary to understand the user's system permissions and the operations they can perform.

Configuration. Almost a mandatory element of every system. The concept of "configuration" can be interpreted very broadly. From configuration files for environment variables to a deep modification of the system's behavior. Although in this case, the word "customization" is more appropriate.

Error handling. The occurrence of both expected and unexpected errors should not lead to a system crash. Errors must be handled correctly, reported to the user, and registered.

Logging. This is journaling of events in the system. At a minimum, this subsystem should register serious events. Each entry is accompanied by a date and time. In case of registration of an error, it is desirable to publish the source of occurrence (stack trace). In addition, logs can be used to debug the system and generalize the user experience.

Geo Quiz Application

To illustrate the principles of code organization, we offer a small project Geo Quiz. Express.JS was used to create it, but there is no fundamental difference.

The essence of the program is to randomly select a geographic question from the knowledge base and send it to the client. In turn, the client can send an answer to the question asked, and the program determines the correctness of the answer.

The program code is divided into logical layers. Let's start with the data, as recommended above. The program operates with questions and answers. For simplicity, a simple file in JSON format is used instead of a database. This is the data layer (\data\questions.json).

To work with data, we need a data access layer (\data\dataManagementLayer.js). This file contains functions for reading data. Note that even in such a simple example, abstraction is used. The name of the data file is assigned to a constant so that it can be changed only in one place. Imagine that the file name has changed, and it is used in twenty functions for reading and writing data. Then you need to change it everywhere to ensure you don't miss anything. In our solution, you only need to change it in one place.

The business logic layer is in \data\quizProcessor.js. There is no reading of data from the knowledge base here intentionally, because each function in this layer is pure. This makes it easier to test and maintain.

And finally, the transport layer is in \routes\quiz.js. Routing is used to distribute responsibilities. Calls related to the quiz are allocated to a separate route. The purpose of this layer is to pass user input into the business logic and supply it with the necessary data. Then get the result and return it to the client.