Gentleman Programming Book

> A clean programmer is the best kind of programmer
>
> *- by Alan Buscaglia*

Chapter 1

Clean Agile

- Problems of waterfall
- Why agile?
- Why you think you are doing agile but in reality...you don't
- Extreme Programming
- TDD
- Atomic design, Front End point of view
- Functional Programming
- User Story and TDD

Chapter 2

Communication First and Foremost

- Companies are not grounded to a certain location
- Coordinating across time zones
- There are some other easy ways we improve this mindset
- Creating Rapport

Chapter 3

Hexagonal Architecture

- Hexagon and its Actors
- Ports and Resources
- Types of Logic in a Service
- Application Example: The Hexagonal Pizza Shop
- Recommended Steps for Working with Hexagonal Architecture with Application Example

Chapter 4

GoLang

- How to Use GoLang
- Advantages
- Recommended Structure
- How Does GoLang Work?
- Data Types
- Structs

- Arrays
- Make Method
- Pointers
- Default Values
- Range Loop
- Maps
- Mutating Maps
- Functions
- Function Values
- Closures
- Methods
- Interfaces
- Interface Values with Nil
- Empty Interfaces
- Type Assertion
- Type Switches
- Stringers
- Errors
- Readers
- Images
- GoRoutines
- Channels
- Buffered Channels
- Range and Close

- GoRoutines Select
- Mutex

Chapter 5

NVIM Gentleman Guide

- What is NVIM ?
- Why I Love NVIM ?
- Installation
- Configuration
- Basics
- Visual Mode
- Block Visual Mode
- Visual Line Mode
- Insert Mode
- Command Mode
- NVIM Motions
- Buffers
- Marks
- Recordings

Chapter 6

Algorithms the Gentleman Way

- Big O Notation
- Needed Previous Knowledge

- Types of Big O Notation
- Examples Using Code
- Worst-case, Best-case, and Average-case Complexity
- Space Complexity
- Arrays
- How to think?
- Linear Search
- Binary Search
- Bubble Sort

Chapter 7

Gentleman of Code: Mastering Clean Architecture

- Introduction to the Book "Gentleman of Code: Mastering Clean Architecture"
- Chapter 1: "Discovering Clean Architecture!"
- Chapter 2: "Separation of Concerns: The Key to an Efficient Architecture"
- Chapter 3: "Design Patterns vs. Architectures: Understanding the Differences"
- Chapter 4: "Maintainability and Scalability with Clean Architecture"
- Chapter 5: "Plugins in Architecture: Flexibility and Adaptability"
- Chapter 6: "Disadvantages and Temporal Considerations of Clean Architecture"
- Chapter 7: "The Structure of Clean Architecture: A Layered View"

Chapter 8

Applying Clean Architecture in the Front End

- Functional Components: Like a Cooking Recipe
- Virtual DOM
- Change Detection: Understanding the Flow
- Mastering Custom Hooks
- Correct Usage of useEffect: Avoiding Common Mistakes
- Component Communication with children Using the Composition Pattern
- Communication Between Components: Composition vs Context vs Inheritance
- Using Context
- Understanding useRef, useMemo, and useCallback
- Making API Requests and Handling Asynchronous Logic
- Concept of Portals
- Adding Styles to Components
- Routing with react-router-dom
- Error Handling with Error Boundaries
- Improving Your Skills with Axios Interceptor

Chapter 10

Typescript Con De Tuti

- Introduction
- Why TypeScript?
- Fundamental Concepts of TypeScript
- TypeScript in Practice
- Best Practices and Patterns

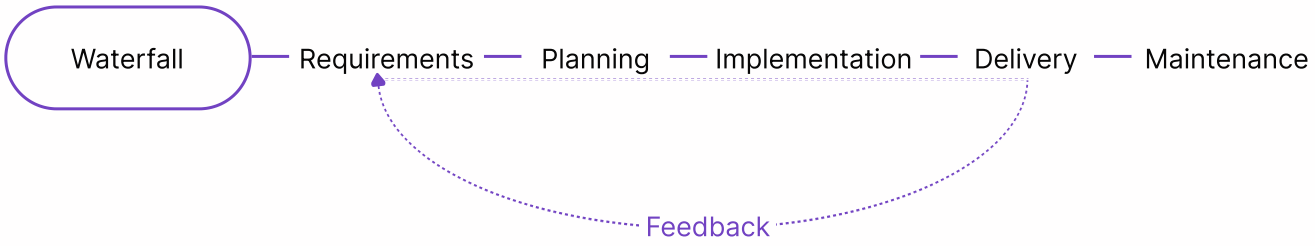
- What is Transpiling?
- Understanding Typing in JavaScript and TypeScript
- TypeScript: The Superhero of Development
- Practical Example in TypeScript: The Use of `any` and the Importance of Typing
- Primitive Types in TypeScript
- Type Inference in TypeScript
- Classes, Interfaces, Enums, and Const: How to Use Them for Typing in TypeScript
- Type vs Interface in TypeScript: When and How to Use Them
- The Concept of Shape in TypeScript
- Understanding union and intersection in TypeScript
- Understanding `typeof` in TypeScript
- Exploring `as const` in TypeScript
- Adventure in TypeScript: Type Assertion and Type Casting
- Functional Overloading in TypeScript: Pure Magic
- TypeScript Utilities: Essential Helpers
- Generics in TypeScript
- The Magic of Enums

Clean Agile

Fantastic! Everything is agile today, all companies love agile, the entire world does agile, but ...are they?

Problems of waterfall:

Let's talk about waterfall, yeah, the bad boy in town, the one everybody hates. Waterfall in The main idea is:



- We get the requirements, what we want to do, the stakeholders needs.
- You plan how to do it, often comes with an analysis and design of the solution.
- You implement it, creating working software.
- You deliver the software and wait for feedback, creating documentation during the process.
- Maintenance of the working solution.

Once we deliver the solution, we ask for feedback and if we need to touch anything, we start the process all over again.

This is awesome as long as the requirements are super stable and we know they will not change during implementation, something that in the real world is practically impossible as they are ALWAYS changing.

If we were a fabric we wouldn't have any of these issues, as we know the specific required materials to create something, we put them in the machine, and the result is always going to be the same.

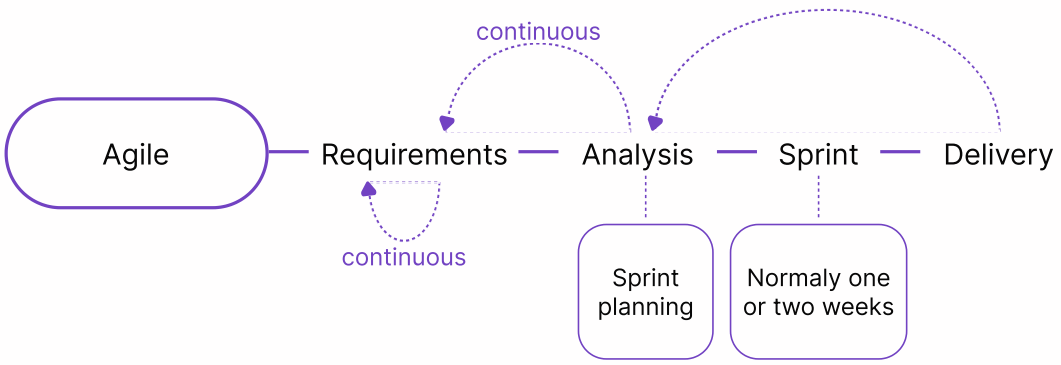
But we work with software solutions to meet people's needs, and these are always changing, evolving. Here comes the biggest problem with the waterfall methodology. We have to wait until the end of the implementation to receive feedback and then start the whole process again, so what if the user needs have changed in the meantime ? We've just wasted a lot of useful time.

One great analogy is the airplane's pilot one, you would like to be informed as soon as possible if there's any problem with the plane and not wait until the engine fails, or even worse, the airplane crashes to receive a notification.

Why agile?

Now here's the thing, a project it's an always evolving succession of events, for example, the analysis never ends ! so getting feedback as soon as possible is the main key of the Agile methodology. Here, we search for the stakeholders involvement in the whole process, delivering minimal amounts of functionalities waiting for a, let's hope positive, response; and if it's a negative one, no problems at all, we can attack them as soon as possible without having to wait for the end of the world to do so.

So how companies usually use agile is the following:



- We continuously get requirements, as we work on small functionalities we can choose what are the most critical needs and implement a plan of action to deliver small parts that can fulfill this end.
- We continuously do an analysis of the requirements to prepare future work. The amount will correspond to the timeframe we already decided according to the needs.
- Now with everything prepared we are comfortable to start working on our tasks inside a sprint. It represents the time frame we decided in which we compromise ourselves to deliver a certain amount of work and it can be variable depending on the need.
- We deliver the functionalities and continue with the process again. The main difference is that we start the work with stakeholders feedback from the previous iteration.

We can fail to deliver inside an adjusted timeframe, and that's not a problem at first, as we measure the team and recollect feedback to adjust to the next sprint. After some iterations we can estimate the correct amount of work the team can deliver in a certain context.

Why you think you are doing agile but in reality...you don't

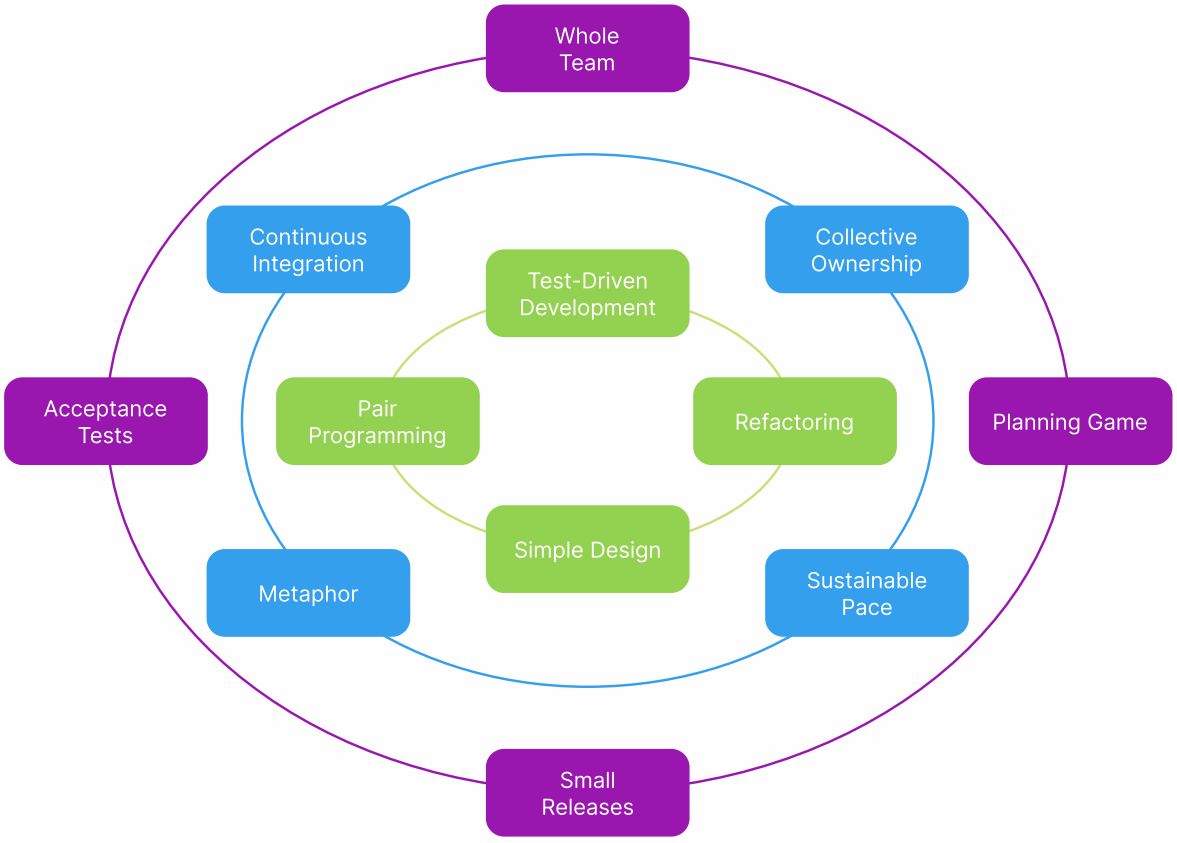
This is normal, you think you are doing agile because you have dailies and that makes the team agile, but in fact that is just one ceremony from many. You can't define being or not agile by the ceremonies that take place inside the project, as it's more of a way of thinking.

You may be working on sprints, using scrum, having retros, and all those amazing things but you also may be working in huge features, not delivering each sprint, not accepting any change until you finish your work or even being owner of your knowledge and not sharing it with the team. If you find yourself in any of these last items...ey...you are not doing agile.

Extreme Programming

This is an amazing practice that according to R.C Martin, co-founder of the agile manifesto and author of Clean Agile, is the true essence of it.

It consists of the organization of practices by three rings called the "Circle of Life". Each ring represents a different aspect of project development:



The "Outer Ring" represents the aspect of business, it contains all business focused practices which when put together creates the perfect environment for project development.

- **Planning Game:** grabbing the project and breaking it down into smaller pieces for better understanding and organization. Features, stories, tasks, etc.
- **Small Releases:** here comes what I was saying about attacking a whole functionality at once and how that could bring a waterfall approach to something that should be agile. We should always try to identify and prioritize the smaller pieces of value and work around them, being the work we want to deliver as soon as possible. The smaller the piece, the faster we will receive feedback and act accordingly.

- **Acceptance Tests:** now this is an easy one to understand but difficult to implement, we need to work in what we consider as a team as done, what are the requirements to really say something has been completely done, or at least on the agreed limitations. One recommendation is to think again in the minimum value we want to, and can, attack with the provided team. If we grab something really big it will be difficult to implement as we need to consider too many things, resulting in missed requirements, vague definitions and miscommunication. Consider creating functionalities that have a clear start and end, the result needs to be something that provides value on its own.
- **Whole team:** inside a proper project team, each member provides a certain functionality, we have our front-ends, back-ends, designers, product owners, project managers, etc. The main problem is always the same, how to communicate the work when it's so different and at the same time, dependent on each other (we will come back to this point in a little while).

The "Middle Ring" represents the aspect of the team, containing all team focused practices to improve team collaboration and interaction:

- **Sustainable Pace:** if you ask, when do you want this done ? the result will always be...well, as soon as possible ! and of course that's really difficult to do so, no because the team can't do it, most of them can, the problem is doing it every single time maintaining the same pace, it's impossible. Your team will be burned out at the third or fourth iteration and then no work will be done, the delivery speed will be greatly reduced. Again, think in small, contained functionalities, that can be delivered at a comfortable speed.
- **Collective Ownership:** how many times do you have to ask your pairs what the product owner is talking about in a meeting because you don't have the correct amount of context ? I'm not talking about those times you are gaming during the daily, but the vortex sucking all information that should be shared with the team and no one seems to know what's going on because no one ever told them. This is a super known issue in companies, the information happens at private quarters so only the people that were in the conversation know what's going on, and later on, try to communicate as best as possible the result with the rest of the team but they create a broken phone game in the process. The project needs to have a communication strategy to attack this kind of situation.
- **Continuous Integration:** as programmers we should be commiting code as much as possible, having the limit of not leaving a single day without new changes in the repository. There's always the possibility of working together in a functionality and missing communication efforts with each other. For example, one creates a method that does exactly the same as another, that was already created by a pair but didn't reach out to notify about it. Commiting changes as fast as possible will deliver feedback to your teammates and keep them updated to always be working on the "latest changes". If we wait to deliver new code after the functionality is done, we will enter waterfall territory all over again.
- **Metaphor:** If we will be working together in a project, we should all understand the context around in the same way, having definitions of each item. Having the same exact name to describe a certain item will bring a higher level of team understanding and leave out the confusion that could bring referring to the exact same item in more than one way. You could think of this as if each project is a different country, there are some of them that communicate in the exact same languages but they use different metaphors. For example, the United States and Londong both use English as their main language, but to represent being upset, American English uses "Disappointed" and British English uses "Gutted".

The "Inner Ring" represents the technical aspect, containing all practices related to improving technical work.

- **Pair Programming:** Sitting with each other to resolve a problem is not only going to make reaching a solution faster, but at the same time you are sharing your point of view between your pairs and also gaining theirs, with also the benefit of reaching a middle ground and creating a set of conventions that the team will follow after. Communication is key when working in a team, and having the possibility to work together to resolve a problem will bring feedback and context around the implementation.
- **Simple Design:** Here we go again, work small. We have already talked about this one but let's bring a little tip, let's say that we want to bring a certain functionality that represents a pretty big challenge to the team, we should always search for a way to provide the same amount of value by giving a much easier alternative. Sometimes we challenge ourselves and deliver a really complex but beautiful proposition of value, but the problem is that maybe that proposition goes nowhere because requirements change and we may find out that in reality the user doesn't want it, that's why also working as simple as possible is the way to go. You can always provide a simple but elegant solution to find the proper value and then iterate on something better.
- **Refactoring:** we all love the phrase "if it works, don't touch it", but that's not the correct mindset as we will enter in a spiral of legacy code by reusing no longer maintainable code. We need to refactor as much as possible. Technical debt is pretty much unavoidable, we always generate some bad quality code because of deadline's time constraints by implementing the fast, but not so correct, solution. One good way of dealing with it is to use part of the start or end of the sprint, according to the priority, to refactor the code, also this could be done using pair programming to use the benefits previously described.
- **Test-Driven Development:** We will talk about it later on, but we can define it as a process where we write our tests before even coding a single line. The main idea is that the requirements of the task define the tests we want to do and in result guide what we code.It can be a great ally when refactoring as we will understand later.

TDD

EVERYONE hates doing tests, for example clients hate PAYING companies for their Front End devs "wasting" time doing tests, and in the end... money. So why we, the wasters of time and money, should want to implement testing right?

Well, there are some things in my mind that can make up for all that hate:

- Code quality
- Code maintenance
- Coding Speed (yeah, you are reading this item correctly)

Understandable right ? you write tests so they pass the use cases, to write them you need to be organized because if not... it will be impossible to test your code. But there are things to consider, how do we write tests in a way that really increases the quality of our code in any meaningful way ?

First let's see what code quality means, and then I will tell you my take on what code quality means to ME.

If you look for an answer this is the one you may find:

"A quality code is one that is clear, simple, well tested, bug-free, refactored, documented, and performant"

Now, the measure of quality goes by the company requirements and the key points are usually reliability, maintainability, testability, portability, and reusability. It's really difficult to create code with 100% of quality, even Walter White couldn't create meth with more than 99.1% of purity; development problems, deadlines and other context and time consuming situations will arise endangering your code quality.

You can't write readable, maintainable, testable, portable, and reusable code if you are being rushed to finish a 4 point story task in just a morning (I really hope that's not your case, and if it is...you got this)

So here comes my take on what code quality is for me. Doing it, it's a mix of doing your best with the current tools, good practices and experience, against the existing context boundaries to create the cleanest code possible. My recommendation to all my students is to reach the objective first and then, if you have time, use it to improve the quality as high as possible. It is better to deliver an ugly thing than an incomplete, but beautiful, functionality. The quality of your code will increase with your experience along the way, as you gain more of it, you will know the best steps to reach an objective in the least amount of time and with the best practices.

Quality code also relates to the level of communication you can provide to your teammates or anyone in a simple glance. It's easy to see a code and say..wow, this is great ! and also say...wow, what a mess ! So when you code, you need to think that you are not the only one working on it, even if you are working alone as a single dev army, that will help a lot.

So let me give you some tools to write better code, first let's open your mind a little.

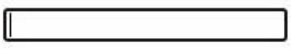
Atomic design, Front End point of view

Separate your code in the minimum piece of logic as possible, the smaller the code the easier to test. This also brings more benefits, like reusage of the code, better maintenance and even better performance; as the code gets smaller and better organized and depending on the language/framework we use, we could end in less processing cycles.

Maintenance will be greatly improved, as we are coding small pieces of work, each one with the loosest coupling and highest cohesion as possible, we can track and modify the code with the minimum number of problems.

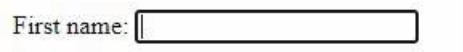
Let me show you how to think atomically, and how you can reach a complete app starting from a small input.

First we have our input:



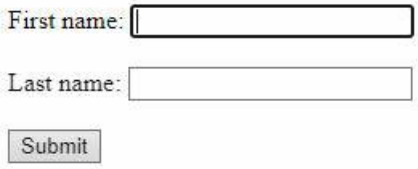
simple stuff, now that is what we call an Atom, the minimum piece of logic as possible. If you code it atomically you can reuse this input everywhere in your app and, later on, if you need to modify its behavior or its looks you just modify one little atom with the result of having an impact on the whole application.

Now, let's say you add a label to that input:



Congrats ! Now you have what it's called a Molecule, the mix between atoms, in this case a label and an input. We can continue going forward and reducing granularity.

We can use the input with the label inside a Form creating an Organism, the mix between molecules:



If we mix Organisms, we will get a **Template**:

And a collection of templates creates our Page, and then using the same logic, our App.

Using this way of thinking will result in your code being really maintainable, easy to browse to track errors and more than anything...easy to test !

If you write anything other than an Atom, it would be really difficult to test anything, as the logic would be of high coupling and therefore impossible to separate enough so that you can check specific cases.

One example would be testing a high coupled code, to validate just a simple thing one would need to start including one piece of code...and then another...and another, and after you finish you will see that you included almost the whole code because there were just too many dependencies from one place into another.

And that's the key to include a mvp (most valuable player) in all of this.

Functional Programming

So functional programming it's a paradigm that specifies ways of coding in a way that we divide our logic into declarative, with no side effects methods. Again...think atomically.

When we start learning how to code we normally do it in an Imperative way, where the priority is the objective and not the way we reach it. Even if it's faster than functional programming, which it is, it can bring a lot of headaches apart from leaving aside all the benefits from the other one.

Let's write a comparison in Javascript.

Imperative way of searching an element inside an array:

```
var exampleArray = [
    { name: 'Alan', age: 26 },
    { name: 'Axel', age: 23 },
];

function searchObject(name) {
    var foundObject = null;

    var index = 0;

    while (!foundObject && exampleArray.length > index) {
        if (exampleArray[index].name === name) {
            foundObject = exampleArray[index];
        }

        index += 1;
    }

    return foundObject;
}

console.log(searchObject('Alan'));

// { name: 'Alan', age: 26 }
```

And now the functional way of reaching the same objective:

```
const result = exampleArrayMap.find(element => element.name === name);

console.log(result);
```

Is not only shorter, it's also scalable. The map method we are applying into the array is a declarative one from ECMAScript, that means that every single time we run the method using the same parameters, we will always get the same result. We also won't modify anything outside the method, that's what's called side effects, the method returns a new array with the elements that comply with the condition.

So if we create methods representing the minimal units of logic as possible, we can reuse working and tested code across the app and maintain it if needed. Again...think atomically.

Now that we know the way of thinking to create high quality and easy maintenance code, let's go into what an User Story is.

🕶 User Story and TDD

What a title right ? Everyone knows about user stories, how to define them, what we need to do with them, but no one follows the same way of writing one or even its structure.

A user story is the explanation of a feature from an user point of view. It normally looks something like this:

What? As an user (who), I want to have the possibility of...(what) to...(why)

How? (use cases) 1- step 1 2- step 2 3- step 3 ...

So as you can see, we define who...the user, what he wants to do...the functionality, why we want this functionality...which while writing it we can even discover that it doesn't even make sense creating it because the objective is not clear, and how we will create it...the use cases.

The use cases represent the number of requirements that we need to fulfill to clearly say that a user story is done, they normally tell the story of the happy path to follow. There are also places where the entities related to the user story and the corner cases (sad path) are also described inside them and I believe that's a really good practice, but the same as when writing high quality code...we need to identify the boundaries of our context to see how can we write the content as specific as possible without transforming our task into a really difficult to follow and time consuming document.

Now TDD, Test Driven Development, it's a process where we define our tests before even coding a single line, so...how do we test something that's not even created right ? Well, that's the magic of it, you can grab your use cases and define what you need to reach each one of them, create tests around them, make them fail, and then fix them to pass..simple as that.

The main idea behind TDD is to think about:

- What do you want to do?.
- What are the core requirements?
- Write tests that will fail around them.
- Create your code knowing what you want to do.

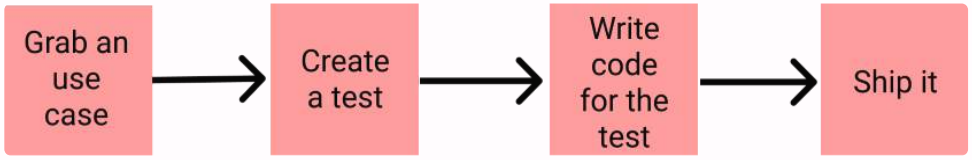
- Make the test pass.

If you think testing is time consuming, well it is, but because you may have previously done it in the classic bad way of first coding everything and then trying to test your code. Remember what we were talking about code quality, good practices, etc ? Well, those are the main elements that will help you test your code and if you are not implementing them correctly we will end in an impossible to separate and test functionality.

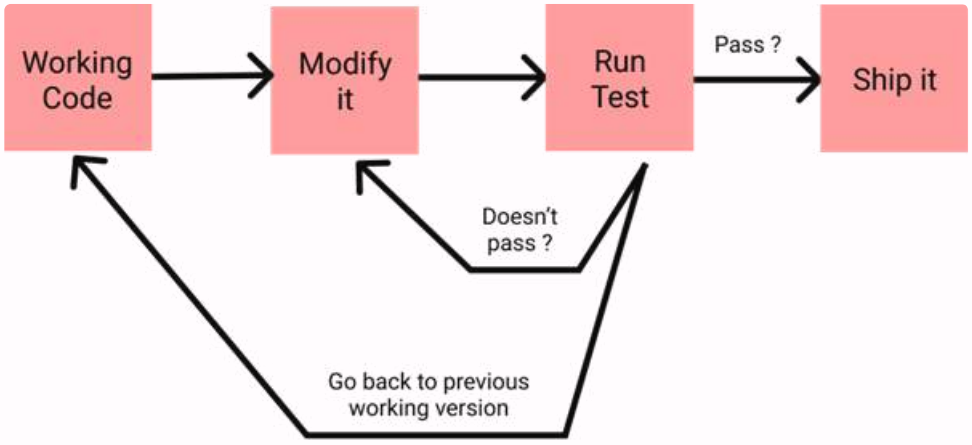
That's why coding knowing what you want to test, using functional programming and an atomic way of thinking can be so beneficial, because you will be creating logic, pinpointing the requirements and in the end... increasing the coding speed.

So here it is, testing also helps increase the coding speed, as you write more manageable code, it's easier to modify a requirement (use case) of your functionality as you have it identified by a test that will tell you if your refactor went correctly. It also reduces the possibility of bugs, so less time fixing problems later on.

TDD flow:



Here's a TDD flow on how to improve code quality without breaking anything:



Communication First and Foremost

We are reaching a new era! Remote work is coming strong and as the commodities increase also problems communicating between distributed teams.

Companies are not grounded to a certain location

Companies are not grounded to their office's location as they now play between the whole world's rules, so we need to change the mindset to understand this new paradigm. A great example is giving a job offer to a candidate, if we consider a salary limited to the candidate's location we run the risk of it being declined as he may have received offers from all over the globe, more tempting and better paid.

This concept may induce problems at any organizational level, people will compare themselves and what they do to professionals from all over the world and may think that they are not being offered the same level of benefits or that they just are not being paid enough. So how do you deal with this problem? making them feel part of something awesome, helping them overcome blockers to personal growth and most important of them all, keeping them learning new stuff.

Not everything that shines is gold goes the phrase, and we can apply the same concept between a company and its employees. People do not always search for money! knowledge is one of the greatest values one can provide as I always preach the following: “Knowledge first, money second; The more you know, the more someone is willing to pay you for it”

Coordinating across time zones

Being a distributed first company is not an easy task, managing work synergy across individuals that are not even in the same time zone can result in a bigger challenge than anticipated.

I recommend learning how to work asynchronously, it is just putting the puzzle pieces together, but the challenge is to find which are those pieces. Along with my professional experience, I detected that the most important and also difficult piece is generating a balanced amount of context among the team.

The way I have been doing it over the years is by understanding that communication is key and secondary to none, your team needs to be in the same boat or it will not work.

Second, you need to know your sources of truth, a place where you can check to clear doubts and search for context because it's always up to date. From my experience I found that there are two different kinds:

- One represents the reality of each business logic, a great example is the usage of Notion or Confluence as sources of truth, where we detail what we expect, why we are doing what we are doing, requirements, corner cases, etc.
- There's also the need for a second type related to workloads and what the team has agreed to do, and is one you already know, the created tickets that the team will complete over the passage of time.

We need two different types because while the first one gives us all the context we may need, we also have the necessity to put a stop at some point and decide when to start the implementation process. This last one details what the team has agreed on doing with the correct amount of context so they can provide enough value and don't be blocked in the process, and we need it so we can continue working on improving and evolving.

Let's think about what happens when having doubts about a certain task, a lot of people would create a comment inside the ticket, leave a message inviting communication over a chat channel, etc. This usually ends in a lot of meaningful chatter, confusion and more doubts, as we can't fully understand the intention of the text. How to fix it? Just sort the order of the elements in a different way:

- Ask for a call, talk with each other and adapt in a way that needs are fulfilled on both sides.
- Document the results over the related ticket and leave a history that can be tracked for future similar situations.
- It's ok to use chat channels for simple subjects but move into a meeting as soon as you see that the conversation is going nowhere.
- Meetings are not meant to add context or new information, please update the source of truth and share it as a first choice.
- Meetings are not meant to add context or new information, please update the source of truth and share it as a first choice.
- Include the related source of truth link inside the tickets, but also try to add all the needed information from it directly into the ticket to leave a definition of what has been agreed at the moment of creation.

There are some other easy ways we improve this mindset

We can start by over-communicating decisions across all geographies, this will result in people understanding what's going on and the whys.

Minimizing the friction in setting up a work environment, having documentation and some sort of guidelines will improve new additions to the team processes of getting to know the way they are expected to work and getting up to the required level to start working.

Clearly define the definition of done, this one relates to the need of having a workload source of truth, what are the acceptance criteria we need to complete to move a task as "done". Also, remember what I said before, a feature needs to have a start and an end of its own in a way that can provide value while being independent.

Using some of the concepts already provided in the previous chapters, for example doing pair programming or code reviews helps distribute knowledge between offices, helps generate a structure between global teams and minimizes the amount of collaboration required.

Creating Rapport

Reaching an accepted level of affinity between distributed teams can prove to be challenging but there are some things we can do to increase our possibilities:

- Communicate even minute details until both offices find a healthy groove.
- Communicate decisions.
- Everyone needs to understand the decision and why it was made.
- Don't use emails as they are an easy way of losing information.
- Use a content management system, for example, a wiki.
- Create channels for individuals and teams to communicate and see updates, another great idea would be to create channels for a certain future and the involved people.
- Spend time creating a simple but effective "Getting Started" guide.

Hexagonal Architecture

Hexagonal architecture is widely used and for good reason. This architecture advocates for "separation of concerns," meaning that business logic is divided into different services or hexagons, which communicate with each other through adapters to the resources they need to fulfill their purposes.

Hexagon and its Actors

It is called hexagonal because its shape resembles a hexagon with a vertical line dividing it in half. The left half refers to primary actors, who initiate the action that starts the hexagon's operation. These actors do not communicate directly with the service; instead, they use an adapter. The right side represents secondary actors, who provide the resources needed for the hexagon to execute its internal logic.

Adapters are key components in hexagonal architecture as they mediate communication between two entities so that they can interact comfortably. For example, in a service that provides user information, the term "username" might be used to identify the user's name, while in another service, the term "userIdentifier" might be used for the same action. Here is where the adapter intervenes to perform a series of transformations, allowing the information to be used in the most convenient way for each entity. In essence, adapters facilitate the integration of different components of the system and interoperability between them.

When an adapter communicates with a primary actor, it is called a driver, while when communicating with a required resource, it is called driven. The drivens are on the right side of the vertical line in the hexagonal diagram, and it is important to note that they can also represent other services. In this case, communication between services should occur through the corresponding adapters, drivers for the service providing the resource and drivens for the service requesting it. Thus, a service can act as the primary actor for another service.

Ports and Resources

The next important concept to understand is that of ports. These indicate the limitations that both our service and adapters have and represent the different functionalities they must provide to primary and secondary actors to meet requests and provide necessary resources.

Types of Logic in a Service

To understand the different types of logic within a service, it is useful to distinguish between business logic, organizational logic, and use cases. An example illustrating this distinction is an application that manages user bank accounts and must allow registration of users over 18 years old.

- **Business Logic:** This is logic that comes from the product and is not affected by external changes. In this example, the requirement that users must be over 18 years old does not stem from a technical limitation but from a specific need of the application being developed. Also, the need to create a user record is business logic, as it is a specific requirement of the application.

- **Organizational Logic:** It is similar to business logic but is reused in more than one project within the same organization. For example, the methodology used to validate and register credit cards in our application could be organizational logic used in several projects within the same company.
- **Use Cases:** These are cases that have a technical limitation and can change if the application's use changes. For example, the requirements for registering a user might not be a use case, as there is no technical limitation to validate the fields of a form. However, the arrangement of the error message, its color, size, etc. can affect the application's use, making these elements use cases. The position and form of displaying fields on the screen can also be a use case, as it affects the application's SEO if there is a change in the content layout shift.

Application Example: The Hexagonal Pizza Shop

One of the most well-known examples is that of a pizzeria where a person wants to place an order. To do this, they will look at the menu with different options, tell the cashier their order, the cashier will communicate the order to the kitchen, the kitchen will perform the necessary procedures to meet the requirement, and return the completed order to the cashier for delivery to the buyer. If we think carefully about each entity in the example, we can find that the buyer is the main actor, the menu with different options is the port, the cashier is the adapter, and the kitchen is our service.

The consumer will order a product by looking at the menu and can only order what they see on it. At the same time, that same order, which may have a catchy name for the public, probably has a simpler name for the kitchen to increase process efficiency. The cashier knows this nomenclature and is responsible for managing proper communication between the consumer and the kitchen. What is a margarita pizza for one person is number 53 for another.

A part we don't see is that the kitchen itself needs resources to complete the order. This means that both the cheese, tomato, and other ingredients must be requested to manage orders. For this, there is probably someone in charge between the restaurant and a raw material supplier. Again, what is tomato for one person is product ABC for another. Here we see the clear example of a secondary actor, the supplier, communicating through an intermediary, the person in charge, to provide the necessary resources to our hexagon.

Recommended Steps for Working with Hexagonal Architecture with Application Example

- 1- Carefully Consider Requirements. An example illustrating this distinction is an application that manages user bank accounts and must allow the registration of users over 18 years old.
- 2- Identify the Business Logic You Need to Fulfill. In the example of the flight reservation system, business logic could include validating flight availability and assigning seats to passengers, as well as managing payments and issuing tickets.
- 3- Identify What Actions Your Hexagon Must Provide to Satisfy the Required Logic. In the example of the flight reservation system, the actions that the hexagon must provide could include: searching for available flights, reserving a flight, assigning seats to passengers, managing payments, and issuing tickets.

- 4- Identify the Resources Needed to Satisfy That Logic and Who Can Provide Them. In the example of the flight reservation system, the necessary resources could include: a database of flights and seats, an online payment provider, and a ticket issuance service. At this step, it is essential to identify who can provide these resources and how they will be integrated into the system.
- 5- Create the Necessary Ports for Drivers and Drivens. In the example of the flight reservation system, the necessary ports could include: a flight search port, a reservation port, a seat assignment port, a payment management port, and a ticket issuance port. It is important that these ports are designed to interact with drivers (user interfaces) and drivens (databases, payment providers, etc.) clearly and consistently.
- 6- Create Stub/Mocked Adapters to Immediately Satisfy Requests and Test Your Hexagon. In this step, adapters are created for drivers and drivens that mimic their real behavior but can be used to test the hexagon immediately without depending on external systems. For example, Stub/Mocked adapters could be created for the flights and seats database, the online payment provider, and the ticket issuance service.
- 7- Create the Necessary Tests That Must Pass Successfully for Your Hexagon to Satisfy the Request. In this step, tests are created to validate that the hexagon works as expected and satisfies the system requirements. For example, tests could be created to validate that available flights can be searched, a flight can be reserved, seats can be assigned to passengers, payments can be managed, and tickets can be issued successfully.
- 8- Create the Logic Inside Your Hexagon to Satisfy Use Cases. In this step, business logic is created inside the hexagon to satisfy the use cases identified in step 2. In the example of the flight reservation system, logic could be created to validate flight and seat availability, assign seats to passengers, and manage the payment and ticket issuance process. This logic should be designed in a way that is easily modifiable and scalable in the future if changes or improvements to the system are required. It is also important that this logic is separated from the specific logic of drivers and drivens to facilitate maintenance and evolution of the system in the future.

To work with hexagonal architecture, it is crucial to follow a methodical and careful process. First and foremost, it is essential to read the requirements carefully to think about the best way to solve them. It is crucial to recognize the business logic and necessary use cases to fulfill it, as you will remember from the previous explanation; this is essential to ensure that all needs are covered.

Once requirements and necessary resources have been identified, it is time to create our ports. To do this, we must recognize the essential methods that must be available for our primary and secondary actors. This will allow us to control access and exits to the service. By convention, port names should start with the word "For," followed by the action they must perform. For example, if we need a port to perform a registration action, we could call it "ForRegistering." We can also reduce the number of ports if we associate different related actions, such as "ForAuthenticating," which will provide registration and login actions.

Following these steps and paying careful attention to details, we can effectively work on hexagonal architecture and achieve optimal results in our projects.

Next, we need to create our driver and driven adapters. To do this, I recommend using the performer of the action as the adapter name, followed by the action itself. For example, in our case, we could call them Registerer or Authenticator, respectively.

The adapters, first and foremost, must be of the stub type and provide controlled information that can be used to satisfy business logic and tests. This way, we can close our hexagon and get it ready for implementation.

With our adapters complete, we proceed to use Test-Driven Development (TDD). We will create the necessary tests to meet the use cases, so we can verify the correct functioning of the logic when implementing it.

Our service must have the necessary entities for typing, satisfy the methods provided in the primary ports, and meet the use cases. The service is responsible for receiving a request, finding the necessary resources through secondary adapters, and using them to meet use cases and, therefore, business logic.

🕶 Code Example Following the Theme of a Banking Application:

```
// Class representing the hexagon
export class BankAccountService {
  private bankAccountPort: BankAccountPort;

  constructor(bankAccountPort: BankAccountPort) {
    this.bankAccountPort = bankAccountPort;
  }

  /**
   * Method to create a new bank account.
   * @param name - The name of the account holder.
   * @param age - The age of the account holder.
   * @throws AgeNotAllowedException if the age is not allowed.
   */
  public createBankAccount(name: string, age: number): void | AgeNotAllowedException {
    if (age >= 18) {
      const bankAccount = new BankAccount(name, age);
      this.bankAccountPort.saveBankAccount(bankAccount);
    } else {
      throw new AgeNotAllowedException("The minimum age to create a bank account is 18 years.");
    }
  }
}

// File bank-account-port.ts
// Interface defining the port to access the database of bank accounts
export interface BankAccountPort {
  saveBankAccount(bankAccount: BankAccount): void;
}

// File bank-account.ts
// Class representing the entity of a bank account
export class BankAccount {
  private name: string;
  private age: number;
```

```
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public getName(): string {
    return this.name;
  }

  public getAge(): number {
    return this.age;
  }
}

// File age-not-allowed-exception.ts
// Custom exception for when the age is not allowed
export class AgeNotAllowedException extends Error {
  constructor(message: string) {
    super(message);
    this.name = "AgeNotAllowedException";
  }
}

// File bank-account-repository.ts
// Class implementing the port to access the database of bank accounts
export class BankAccountRepository implements BankAccountPort {
  private bankAccounts: BankAccount[] = [];

  public saveBankAccount(bankAccount: BankAccount): void {
    this.bankAccounts.push(bankAccount);
  }
}

// File bank-account-controller.ts
// Class representing the driver for creating bank accounts
import { BankAccountService } from "./bank-account-service";
import { AgeNotAllowedException } from "./age-not-allowed-exception";

export class BankAccountController {
  private bankAccountService: BankAccountService;

  constructor(bankAccountService: BankAccountService) {
    this.bankAccountService = bankAccountService;
  }

  /**
   * Method to create a new bank account.
   * @param name - The name of the account holder.
   * @param age - The age of the account holder.
   */
  public createBankAccount(name: string, age: number): void {
    try {
```

```
      this.bankAccountService.createBankAccount(name, age);
      console.log("The bank account has been created successfully.");
    } catch (e) {
      if (e instanceof AgeNotAllowedException) {
        console.log(e.message);
      } else {
        console.log("An error occurred while creating the bank account.");
      }
    }
  }
}

// File main.ts
// Usage example
import { BankAccountRepository } from "./bank-account-repository";
import { BankAccountService } from "./bank-account-service";
import { BankAccountController } from "./bank-account-controller";

const bankAccountPort = new BankAccountRepository();
const bankAccountService = new BankAccountService(bankAccountPort);
const bankAccountController = new BankAccountController(bankAccountService);

bankAccountController.createBankAccount("John Doe", 20);
bankAccountController.createBank
```

How to GoLang

An AMAZING language created by Google in collaboration with Rob Pike, Ken Thomson, and Robert Griesemer.

Advantages:

- Fast, compiles directly into machine code without using an interpreter.
- Easy to learn, very good documentation, and many things are simplified.
- Scales very well, supports concurrent programming through "GoRoutines".
- Automatic garbage collector, automatic memory management.
- Included formatting engine, no need for third parties.
- No libraries required for testing or benchmarks because they are already included.
- Very little boilerplate for creating applications.
- Has an API for network programming, included as a standard library.
- VERY fast, in some benchmarks it is faster than backend applications made in Java and Rust.
- Built-in template system, GREAT for working with HTMX.

Recommended Structure:

- **ui** (frontend-related content in case of server-side rendering)
  - *html* (templates)
  - *static* (multimedia and style static content)
    - *assets*
    - *css*
- **internal** (content related to tools and reusable entities throughout the project)
  - *models*
  - *utils*
- **cmd**
  - *web* (contains the application logic)
    - *domain* (business logic)

- *routes* (available routes)

How Does GoLang Work?

GoLang uses a base file called go.mod, which will contain the main module that will be called the same as the project, and also the version of Go used. Then each file will have the extension ".go" to identify that it is a package belonging to the language.

But... what is a package? If you come from JavaScript you can think of it in the same way as an ES module since it is used to encapsulate related logic. But unlike ES modules, the package is identified by the lines of code "package packageName" in camel case the name of the package in question and it imports the location of the package. Different files containing logic belonging to the same package can be arranged separately BUT they must be under the same parent folder as this is of utmost importance to later import said package in different ones.

To import a different package is done through the word "import" plus the path to which the package belongs.

```
import "miProject/cmd/web/routes"
```

If you need more than one package at a time, it is not necessary to repeat the line of code since it can be grouped using "()" the various packages:

```
import (
    "miProject/cmd/web/routes"
    "miProject/internal/models"
)
```

It is worth mentioning that then Go will relate the final name of the path with the use of the package so in order to use logic contained in it will be done thinking of it as if it were an object, where each property represents a logical element of the package:

```
routes.MyRoute
```

Private and public package methods:

If the method starts with lowercase it is a private method, it cannot be accessed from outside the package itself.

```
func myFunction()
```

If the method starts with uppercase it is a public method, it can be accessed by importing the package from another.

```
func MyFunction()
```

Package scope

Let's see a Go file

```
package main - package name

import "fmt" - fmt package is imported, no path because it's Go's own

var number int = 2

func main() {
    i, j := 42, 2701 - local variables to the method, i with value 42 and j with value 2701

    fmt.Println(i) - using the "Println" method of the "fmt" package
}
```

You surely have noticed something, "number" has a type "int" preceding the value assignment, while "i" and "j" do not, this is because like Typescript, Go infers the type for those primitives. Let's see how to work with types.

Data Types

- **bool** = true / false
- **string** = string of characters
- **int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr** = integer numeric values with their limits, these are generally 32 bits on 32-bit systems and 64 for 64-bit systems. Integer should be used unless there is a specific reason to use a restricted value.
- **byte** === uint8
- **rune** === int32
- **float32, float64** = represents real numeric values
- **complex64, complex128** = complex numbers that have a real part and an imaginary part.

Structs

Represents a collection of properties, you can think of it as a Typescript interface, as it represents the contract that must be followed when creating a property. Important, if you want that property to be accessible outside the package, remember it should start with "uppercase".

```
type Person struct {
    Name string
    LastName string
    Age int
}

var person = Person {
    Name: "Gentleman",
    LastName: "Programming",
    Age: 31
}

fmt.Println(person.Name)
```

Another way:

```
var persona2 = Persona{"Gentleman", "Programming", 31}
```

Arrays

Now the fun begins, arrays are quite different from what we are used to as they MUST have the maximum number of elements they are going to contain inside:

```
var a [10]int - creates an array of 10 elements of type int

a[0] = "Gentleman"
```

Or also

```
var a = [2]int{2, 3}

fmt.Println(a) - [2 3]
```

If we need it to be dynamic we can talk about "slices". A "slice" is a portion of an existing array or a representation of a collection of elements of a certain type.

```
var primes = [6]int{2 ,3 ,5, 7, 11, 13}
var s []int = primes[1:4] - creates a slice using "primes" as a base from position 1 to 4

fmt.Println(s) - [3 5 7]

s = append(s , 14)
```

```
fmt.Println(s) - [3 5 7 14]

fmt.Println(primes) - [2 3 5 7 14 13]
```

You can also omit values for maximum and minimum ranges making them have default values:

```
var a [10]int

is the same as

a[0:10]
a[:10]
a[0:]
a[:]
```

Make Method

To create dynamic slices you can use the included "make" method, this will create an array filled with empty elements and return a slice referring to it. The "len" method can be used to see how many elements it currently contains and "cap" to see its capacity, that is, how many elements it can hold.

```
a := make([]int, 0, 5)  // len(a)=0 cap(a)=5
```

Pointers

If you come from Javascript...

this will take a bit, but let's see together the following example:

```
type ElementType struct {
    name string
}

var exampleElement = ElementType {
    name: "Gentleman",
}

func MyFunction(element ElementType) {
    ...
}

MyFunction(exampleElement)
```

```
Here you might think that we are working on the element "exampleElement", but it's quite the opposite.

So if we want to work with the same element passed as a parameter to the function, a pointer must be us

```go
var a = 1
```

It creates a memory space which inside it contains the value "1" and we create a reference to that memory space called "a". The difference with Javascript is that this reference is not passed to the method unless we have created a pointer to it!

```go
var p *int // pointer "p" that will reference a property of type "int"

i := 42
p = &i // create a direct pointer to the property "i"

// If we want to access the value referenced by the pointer "p", we use the pointer's name preceded by
fmt.Println(*p) // 42

*p = 21

fmt.Println(*p) // 21
```

Where this changes is if we point to a "struct", as it would be a bit cumbersome to do (*p).Property, it reduces to using it as if it were the struct itself:

```go
v := Person{"Gentleman"}
p := &v
p.Name = "Programming"
fmt.Println(v) // {Programming}
```

Default Values

In Go, when you declare a variable without explicitly assigning a value, it takes on a default value based on its type. Here's a table summarizing the defaults:

Default Values for Data Types:

- **bool**: `false`
- **string**: `""` (empty string)
- **Numeric Types**: `0`
- **array**: `[0...]` (zero-valued)
```

- **map**: `nil` (uninitialized)
- **slice**: `nil` (uninitialized)
- **pointer**: `nil` (uninitialized)
- **function**: `nil` (uninitialized)

Range Loop

The `range` loop is a powerful construct for iterating over sequences like slices, arrays, maps, and strings. It provides two components: the index (`i`) and the value (`v`) of each element. Here are three common variations:

- **Full Iteration:**

```
var arr = []int{5, 4, 3, 2, 1}

for i, v := range arr {
  fmt.Printf("index: %d, value: %d\n", i, v)
}
```

This approach iterates over both the index and value of each element in `arr`.

- **Ignoring Index:**

```
for _, v := range arr {
  fmt.Printf("value: %d\n", v)
}
```

The underscore (_) discards the index information, focusing only on the element values.

- **Ignoring Value:**

```
for i, _ := range arr {
  fmt.Printf("index: %d\n", i)
}
```

Similarly, you can use an underscore to skip the value and access only the indices.

Maps

Maps are unordered collections that associate unique keys (of any hashable type) with values. Go provides two ways to create and work with maps:

- **Using `make` Function:**

```
type Persona struct {
  DNI, Nombre string
}

var m map[string]Persona

func main() {
  m = make(map[string]Persona)
  m["123"] = Persona{"123", "pepe"}
  fmt.Println(m["123"])
}
```

- **Map Literal:**

```
type Persona struct {
  DNI, Nombre string
}

var m = map[string]Persona{
  "123": Persona{"123", "pepe"},
  "124": Persona{"124", "jorge"},
}

func main() {
  fmt.Println(m)
}
```

  Map literals offer a concise way to initialize maps with key-value pairs.

Mutating Maps

- **Insertion:**

```
m[key] = element
```

  Adds a new key-value pair to the map `m`.

- **Retrieval:**

```
element = m[key
```

Functions

Functions are reusable blocks of code that perform specific tasks. They are declared with the `func` keyword, followed by the function name, parameter list (if any), return type (if any), and the function body enclosed in curly braces.

Here's an example:

```
func greet(name string) string {
  return "Hello, " + name + "!"
}

func main() {
  message := greet("Golang")
  fmt.Println(message)
}
```

Function Values

Functions can be assigned to variables, allowing you to pass them around like any other value. This enables powerful techniques like higher-order functions.

Here's an example demonstrating how to pass a function as an argument and call it indirectly:

```
func CallCallback(callBack func(float64, float64) float64) float64 {
  return callBack(3, 4)
}

func hypot(x, y float64) float64 {
  return math.Sqrt(x*x + y*y)
}

func main() {
  fmt.Println(hypot(5, 12))
  fmt.Println(CallCallback(hypot))
}
```

Closures

Closures are a special type of function that captures variables from its enclosing environment. This allows the closure to access and manipulate these variables even after the enclosing function has returned.

Here's an example of a closure that creates an "adder" function with a persistent sum:

```
func adder() func(int) int {
  sum := 0

  return func(x int) int {
    sum += x
    return sum
  }
}

func main() {
  pos, neg := adder(), adder()

  for i := 0; i < 10; i++ {
    fmt.Println(
      pos(i),
      neg(-2*i),
    )
  }
}
```

Methods

Go doesn't have classes, but it allows defining methods on types (structs, interfaces). A method is a function associated with a type, taking a receiver argument (usually the type itself) that implicitly refers to the object the method is called on.

Here's an example of a `Persona` struct with a `Saludar` method:

```
type Persona struct {
  Nombre, Apellido string
}

func (p Persona) Saludar() string {
  return "Hola " + p.Nombre
}

func main() {
  p := Persona{"Pepe", "Perez"}
  fmt.Println(p.Saludar())
}
```

Methods can also be defined on non-struct types:

```
type Nombre string

func (n Nombre) Saludar() string {
  return "Hola " + string(n)
}

func main() {
  nombre := Nombre("Pepe")
  fmt.Println(nombre.Saludar())
}
```

Methods can accept pointers as receivers, enabling modifications to the original object:

```
type Persona struct {
  nombre, apellido string
}

func (p *Persona) cambiarNombre(n string) {
  p.nombre = n
}

func main() {
  p := Persona{"pepe", "perez"}
  p.cambiarNombre("juan")
  fmt.Println(p) // Output: {juan perez}

  pp := &Persona{"puntero", "persona"}
  pp.cambiarNombre("punteroNuevoNombre")
  fmt.Println(*pp) // Output: {punteroNuevoNombre persona}
}
```

Go automatically dereferences pointer receivers when necessary, so you don't always need to use the explicit * operator.

Interfaces

Interfaces define a set of methods that a type must implement. They provide a way to achieve polymorphism, allowing different types to be used interchangeably as long as they implement the required methods.

Here's an example of an `Interface` that defines two methods, `Saludar` and `Moverse`:

```
type Persona interface {
  Saludar() string
  Moverse() string
```

```
}

type Alumno struct {
  Nombre string
}

func (a Alumno) Saludar() string {
  return "Hola " + a.Nombre
}

func (a Alumno) Moverse() string {
  return "Estoy caminando"
}

func main() {
  var persona Persona = Alumno{
    "Pepe",
  }

  fmt.Println(persona.Saludar())
  fmt.Println(persona.Moverse())
}
```

Interface Values with Nil

Interface values can be `nil`, indicating that they don't hold a reference to any specific object. Here's an example demonstrating how to handle `nil` interface values:

```
type I interface {
  M()
}

type T struct {
  S string
}

func (t *T) M() {
  if t == nil {
    fmt.Println("<nil>")
    return
  }
  fmt.Println(t.S)
}

func main() {
  var i I

  var t *T
  i = t
```

```
  describe(i)
  i.M() // Output: <nil>

  i = &T{"hello"}
  describe(i)
  i.M() // Output: hello
}

func describe(i I) {
  fmt.Printf("(%v, %T)\n", i, i)
}
```

Empty Interfaces

If you don't know the specific methods an interface might require beforehand, you can create an empty interface using the `interface{}` type. This allows you to store any value in the interface, but you won't be able to call methods on it directly.

```
var i interface{}
```

Type Assertion

When we use an empty interface `go interface{}`, we may use any kind of type BUT, this also comes with problems. How do we know if the parameter of a method is of the expected type if if it's an empty interface ? Here's where Type Assertions come handy, as they provide the possiblity of testing if the empty interface is of the expected type.

```
t := i.(T)
```

This means that the interface value `i` holds the concrete type `T` and assigns the underlying `T` value to the variable `t`.

If `i` does not hold a `T`, this will trigger a panic.

You can test if the interfaces faclue holds a specific type by using a second parameter, just like we do with `err`:

```
t, ok := i.(T)
```

This will save true or false inside `ok`. If false, `t` will save a zero value inside and no panic will occur.

```
func main() {
    var i interface{} = "hello"

    s := i.(string)
    fmt.Println(s) // hello

    s, ok := i.(string)
    fmt.Println(s, ok) // hello true

    f, ok := i.(float64)
    fmt.Println(f, ok) // 0 false

    f = i.(float64) // panic: interface conversion: interface {} is string, not float64
    fmt.Println(f) // nothing, it will panic before
  }
```

Type Switches

It provides the possiblity of doing more than one Type Assertion in series.

Just like a regular switch statemenet, but we use types instead of values, and the later ones will be compared against the type of the value held by the given interface value.

```
switch v := i.(type) {
    case T:
    // if v has type T
    case S:
    // if v has type S
    default:
    // if v has neither type T or S, it will have the same type as "i"
  }
```

Acclaration: just like Type Assertions, we use a type as a parameter `go i.(T)`, but instead of using `T`, we need to use the keyword `type`.

This is great when executing different logics which depends on the type of the parameter:

```
type Greeter interface {
    SayHello()
}

type Person struct {
    Name string
}

func (p Person) SayHello() {
```

```
    fmt.Printf("Hello, my name is %s!\n", p.Name)
}

type Number int

func (n Number) SayHello() {
    if n%2 == 0 {
        fmt.Printf("Hello, I'm an even number: %d!\n", n)
    } else {
        fmt.Printf("Hello, I'm an odd number: %d!\n", n)
    }
}

func main() {
    greeters := []Greeter{
        Person{"Alice"},
        Person{"Bob"},
        Number(3),
        Number(4),
    }

    for _, greeter := range greeters {
        switch value := greeter.(type) {
        case Person:
            value.SayHello()
        case Number:
            value.SayHello()
        }
    }
}
```

Stringers

It's a type that defines itself as a `string`, it's defined by the `fmt` package and it's used to print values.

```
type Person struct {
  Name string
  Age int
}

func (p Person) String() string {
  return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
    a := Person{"Gentleman Programming", 32}
    z := Person{"Alan Buscaglia", 32}
```

```
    fmt.Println(a, z) // Gentleman Programming (32 years) Alan Buscaglia (32 years)
```

Example using Stringers to modify the way we show an IpAdress when using fmt.Println :

```
type IPAddr [4]byte

func (ip IPAddr) String() string {
  str := ""

    for i, ipValue := range ip {
    str += fmt.Sprint(ipValue)

        if i < len(ip)-1 {
            str += "."
        }
  }

    return fmt.Sprintf("%s", str)
}

// TODO: Add a "String() string" method to IPAddr.

func main() {
    hosts := map[string]IPAddr{
        "loopback":  {127, 0, 0, 1},
        "googleDNS": {8, 8, 8, 8},
    }

    for _, ip := range hosts {
        fmt.Println(ip)
    }
}
```

Errors

To show erros, Go uses `error` values to express `error states`, and for this, the `error` type exists and it's similar to the `fmt.Stringer` interface:

```
type error interface {
  Error() string
}
```

Exactly as with `fmt.Stringer` the `fmt` package looks for the `error` interface when printing values. Normally methods return an `error` value and we should use it to manage what to do in case it's different to `nil`:

```
i, err := strconv.Atoi("42")

if err != nil {
  fmt.Println("couldn't convert number: %v\n", err)
  return
}

fmt.Println("Converted integer: ", i)
```

Readers

Another great interface which represents the read end of a stream of data, this data may be streamed over files, network connections, compressors, ciphers, etc.

And it has a **Read** method:

```
func (T) Read(b []byte) (n int, err error)
```

This method will populate the byte array with data and returns the number of bytes populated and an error value. It returns an `io.EOF` error when the stream ends.

```
func main() {
  data := "Gentleman Programming"


  // create a new io.Reader reading from data
  reader := strings.NewReader(data)

  // create a buffer to store the copied data
  var buffer strings.Builder

  // copy data from the reader to a buffer. io.Copy reads from the reader and writes to the writer unti
  n, err := io.Copy(&buffer, reader)

  if err != nil {
    fmt.Println("Error:", err)
  } else {
      fmt.Println("\n%d bytes copied successfully. \n", n)

      // access the data copied into the buffer
      fmt.Println("Copied Data:", buffer.String())
    }
}
```

Example, let's get a ciphered string and decode it !

```go
package main

import (
    "io"
    "os"
    "strings"
)

type rot13Reader struct {
    r io.Reader
}

func (rr *rot13Reader) Read(p []byte) (n int, err error) {
    n, err = rr.r.Read(p)
    for i := 0; i < n; i++ {
        if (p[i] >= 'A' && p[i] <= 'Z') || (p[i] >= 'a' && p[i] <= 'z') {
            if p[i] <= 'Z' {
        // p[i] - 'A' calculates the position of the current character relative to 'A', then we add 13
        // then we apply '%26' to ensure that the result is within the range of the alphabet (26 letter
        // and at the end we add 'A' wich converts the result back to the ASCII value of a letter
                p[i] = (p[i]-'A'+13)%26 + 'A'
            } else {
                p[i] = (p[i]-'a'+13)%26 + 'a'
            }
        }
    }
    return
}

func main() {
    s := strings.NewReader("Lbh penpxrq gur pbqr!")
    r := rot13Reader{s}
    io.Copy(os.Stdout, &r)
}
```

Images

The package `iamges` defines the `Image` interface, which is a powerful tool to work with images as you can create, manipulate, and decode various types of images such as PNG, JPEG, GIF, BMP, and more:

```go
package image

type Image interface {
  ColorModel() color.Model
  Bounds() Rectangle
  At(x, y int) color.Color
}
```

Let's create a black small image with a red pixel in the center:

```
package main

import (
    "image"
    "image/color"
    "image/png"
    "os"
)

func main() {
    // Create a new RGBA image with dimensions 100x100
    img := image.NewRGBA(image.Rectangle(0, 0, 100, 100))

    // Set all pixels to black
    for x := 0; x < 100; x++ {
        for y := 0; y < 100; y++ {
            img.Set(x, y, color.Black)
        }
    }

    // Set the pixel at the center to red
    img.Set(50, 50, color.RGBA{255, 0, 0, 255})

    // Create a PNG file to save the image
    file, err := os.Create("simple_image.png")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    // Encode the image to PNG format and save it to the file
    err = png.Encode(file, img)
    if err != nil {
        panic(err)
    }

    println("Simple image generated successfully!")
}
```

GoRoutines

As we mentioned before, Go is a language that supports concurrent programming through "GoRoutines". A GoRoutine is a lightweight thread managed by the Go runtime, allowing you to run multiple functions concurrently.

BUT it's different than other languages, as it's a virtual thread that runs on a real thread, and it's managed by the Go runtime.

To execute a function as a GoRoutine, you just need to add the `go` keyword before the function call:

```
    go f(x, y, z)
```

This will run the function `f(x, y, z)` concurrently in a new GoRoutine. The parameters are evaluated at the time of the function call, so if they change later, the GoRoutine will use the updated values.

Let's see an example:

```
package main

import (
  "fmt"
)

func say(s string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
  }
}

func main() {
  // Launch a new goroutine to run the say function with "Hello"
  go say("Hello")

  // Print "World" 3 times in the main function
  for i := 0; i < 3; i++ {
    fmt.Println("Gentleman")
  }
}
```

When you run this code, the output won't necessarily be "Hello" followed by "Gentleman" three times each. This is because the goroutines are running concurrently. You might see "Hello" and "Gentleman" mixed together.

Goroutines are lightweight, so you can create thousands of them without any performance issues. They are managed by the Go runtime, which schedules them efficiently on real OS threads.

Another great feature is that the run in the same address space, so they can communicate with each other using channels sharing memory, but, this also needs to be managed and synchronized.

To do so we can use `channels`:

Channels

They will be our way to communicate between goroutines, they are typed and can be used to send and receive data with the channel operator <-:

```
ch <- v // Send v to channel ch.
v := <-ch // Receive from ch, and assign value to v.
```

The data will flow in the direction of the arrow, so if you want to send data to a channel, you should use the arrow pointing to the channel, and if you want to receive data from a channel, you should use the arrow pointing from the channel.

You can also create a channel with the use of the `make` function:

```
ch := make(chan int)
```

This will create a channel that will send and receive integers.

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without we having to manually manage that synchronization.

```
package main

import (
  "fmt"
)

func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Wait infinitely for messages on the channel (ensure all "Hello" are printed)
  for {
    msg := <-ch // Receive message from the channel
    fmt.Println("Received:", msg)
  }
```

```
  fmt.Println("Gentleman") // Print "Gentleman" after receiving all messages
}
```

Buffered Channels

All channels can be buffered, this means that they can hold a limited number of values without a corresponding receiver for those values.

When the channel is full, the sender will block until the receiver has received a value. This is extremely useful when you want to send multiple values and you don't want to loose them if the receiver is not ready.

```
ch := make(chan int, 100)
```

This will create a channel that can hold up to 100 integers.

If you send more than 100 values to the channel, the sender will block until the receiver has received some values.

```
func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    ch <- 3 // fatal error: all goroutines are asleep - deadlock!

    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Range and Close

You can close a channel at any time ! a recommended time to close a channel is when you want to signal that no more values will be sent on it and the one to do it should be the sender, never the receiver as sending on a closed channel will cause a panic:

```
v, ok := <-ch
ok will be false if there are no more values to receive and the channel is closed.
```

To receive values from a chanel until it's closed you can use `range`:

```
for i:= range ch {
    fmt.Println(i)
```

```
}
```

Do we need to close them ? Not necessarily, only if the receiver needs to know that no more values will be sent, or if the sender needs to tell the receiver that it's done sending values, this way we will terminate the `range` loop.

Example:

```
func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
  close(ch) // Close the channel after sending messages
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Loop to receive and print messages until channel is closed
  for {
    msg, ok := <-ch // Receive message and check channel open state
    if !ok {
      break // Exit loop if channel is closed
    }
    fmt.Println("Received:", msg)
  }

  fmt.Println("Messages received. Exiting.")
}
```

GoRoutines Select

The `select` statement lets a goroutine wait on multiple communication operations. It blocks until one of its cases can run, then it executes that case.

It's useful when you want to wait on multiple channels and perform different actions based on which channel is ready.

```
func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
```

```
  // Close the channel after the loop finishes sending messages
  close(ch)
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Use select to handle messages from the channel or a timeout
  for {
    select {
    case msg, ok := <-ch: // Receive message and check channel open state
      if !ok {
        fmt.Println("Channel closed. Exiting.")
        break
      }
      fmt.Println("Received:", msg)
    case <-time.After(1 * time.Second): // Timeout after 1 second if no message received
      fmt.Println("Timeout waiting for message.")
      break
    }
  }
}
```

You can also use a `default` case in a `select` statement, this will run if no other case is ready:

```
func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
  close(ch) // Close the channel after sending messages
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Use select with default case
  for {
    select {
    case msg, ok := <-ch:
      if !ok {
        fmt.Println("Channel closed. Exiting.")
```

```
            break
        }
        fmt.Println("Received:", msg)
    case <-time.After(1 * time.Second):
        fmt.Println("Timeout waiting for message.")
        break
    default:
        fmt.Println("Nothing to receive or timeout yet.")
    }
  }
}
```

Now let's do an exercise where we will check if two node trees have the same sequence of values:

```
package main

import (
    "fmt"
)

type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}

func isSameSequence(root1, root2 *TreeNode) bool {
    seq1 := make(map[int]bool)
    seq2 := make(map[int]bool)

    traverse(root1, seq1)
    traverse(root2, seq2)

    return equal(seq1, seq2)
}

func traverse(node *TreeNode, seq map[int]bool) {
    if node == nil {
        return
    }

    seq[node.Val] = true
    traverse(node.Left, seq)
    traverse(node.Right, seq)
}

func equal(seq1, seq2 map[int]bool) bool {
    if len(seq1) != len(seq2) {
        return false
    }
```

```
    for val := range seq1 {
        if !seq2[val] {
            return false
        }
    }

    return true
}

func main() {
    // Constructing the first binary tree
    root1 := &TreeNode{
        Val: 3,
        Left: &TreeNode{
            Val: 1,
            Left: &TreeNode{
                Val: 1,
            },
            Right: &TreeNode{
                Val: 2,
            },
        },
        Right: &TreeNode{
            Val: 8,
            Left: &TreeNode{
                Val: 5,
            },
            Right: &TreeNode{
                Val: 13,
            },
        },
    }

    // Constructing the second binary tree
    root2 := &TreeNode{
        Val: 8,
        Left: &TreeNode{
            Val: 3,
            Left: &TreeNode{
                Val: 1,
                Left: &TreeNode{
                    Val: 1,
                },
                Right: &TreeNode{
                    Val: 2,
                },
            },
            Right: &TreeNode{
                Val: 5,
            },
        },
        Right: &TreeNode{
```

```
                Val: 13,
                },
    }

    fmt.Println(isSameSequence(root1, root2)) // Output: true
}
```

Mutex

Something that we need to take care of when working with GoRoutines is the access to shared memory, were more than one GoRoutine can access the same memory space at the same time, this can lead to great conflicts.

This concept is called **`mutual exclusion`**, and it's solved by the use of **`mutexes`**, which are used to synchronize access to shared memory.

```
import ("sync")

var mu sync.Mutex
```

It has two methods, **`Lock`** and **`Unlock`**, which are used to protect the shared memory:

```
func safeIncrement() {
    mu.Lock() // lock the shared memory
    defer mu.Unlock() // unlock the shared memory when the function returns
    count++ // increment the shared memory
}
```

Here we are using the **`defer`** statement to ensure that the mutex is unlocked when the function returns, even if it panics.

```
package main

import (
    "fmt"
)

type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}

type SequenceCollector struct {
    sequence map[int]bool
}
```

```
func isSameSequence(root1, root2 *TreeNode) bool {
    seq1 := &SequenceCollector{sequence: make(map[int]bool)}
    seq2 := &SequenceCollector{sequence: make(map[int]bool)}

    traverse(root1, seq1)
    traverse(root2, seq2)

    return equal(seq1.sequence, seq2.sequence)
}

func traverse(node *TreeNode, seq *SequenceCollector) {
    if node == nil {
        return
    }

    seq.sequence[node.Val] = true

    traverse(node.Left, seq)
    traverse(node.Right, seq)
}

func equal(seq1, seq2 map[int]bool) bool {
    if len(seq1) != len(seq2) {
        return false
    }

    for val := range seq1 {
        if !seq2[val] {
            return false
        }
    }

    return true
}

func main() {
    // Construyendo el primer árbol binario
    root1 := &TreeNode{
        Val: 3,
        Left: &TreeNode{
            Val: 1,
            Left: &TreeNode{
                Val: 1,
            },
            Right: &TreeNode{
                Val: 2,
            },
        },
        Right: &TreeNode{
            Val: 8,
            Left: &TreeNode{
                Val: 5,
```

```
            },
            Right: &TreeNode{
                Val: 13,
            },
        },
    }

    // Construyendo el segundo árbol binario
    root2 := &TreeNode{
        Val: 8,
        Left: &TreeNode{
            Val: 3,
            Left: &TreeNode{
                Val: 1,
                Left: &TreeNode{
                    Val: 1,
                },
                Right: &TreeNode{
                    Val: 2,
                },
            },
            Right: &TreeNode{
                Val: 5,
            },
        },
        Right: &TreeNode{
            Val: 13,
        },
    }

    fmt.Println(isSameSequence(root1, root2)) // Salida: true
}
```

NVIM Gentleman Guide !

First of all...

What is NVIM ?

Vim was first invented on the 2nd of November 1991 by Bram Moolenaar. Vim is a highly configurable text editor built to enable efficient text editing.

It is an improved version of the vi editor distributed with most UNIX systems. Vim is often called a "programmer's editor," and so useful for programming that many consider it an entire IDE.

It's not just for programmers, though. Vim is perfect for all kinds of text editing, from composing email to editing configuration files.

So... what is NVIM then ?

NVIM is a fork of Vim, with a lot of new features, better performance, and a lot of new plugins and configurations.

It is a text editor that is highly configurable and can be used for programming, writing, and editing text files.

The most funny part is that it isn't even released yet ! as the time of writting this guide, NVIM is still in the beta phase 0.9.5. But it is stable enough to be used as a daily driver.

Why I love NVIM ?

EFficiency, Speed, and Customization.

Let's talk about me a little bit, Im a software engineer, and I spend most of my time writing code. I have tried a lot of text editors and IDEs, but I always come back to Vim. Why ? because it is fast, efficient, and highly customizable.

But Vim has its own problems, it is hard to configure, and it has a steep learning curve for beginners, but as a Dark Souls lover... I love the challenge. Once you master the beast, you will never go back.

Having the power of not leaving my keyboard, learn something new every day, play with new plugins, and create my own configurations is what makes me love Vim. It can run anywhere, on any platform, it's fast, it's light, you can share your config really easily, and it is open-source. And the best part... you look amazing while using it, a lot of people will ask you "what is that ?" and you will feel like a hacker in a movie, running commands and shortcuts and showing the power of your efficiency.

Ok, let's start this guide, I will show you how to install NVIM, configure it, and use it as a daily driver. I will show you how to install plugins, create your own configurations, and make it look amazing.

Let's start with the installation.

Installation

Previous Requirements

(Execute all commands using the system default terminal, we will change it later)

WINDOWS USERS:

First, let's install WSL (https://learn.microsoft.com/en-us/windows/wsl/install), this is a must-have for windows users, it will allow you to run a full Linux distribution on your windows machine and I recommend using the version 2 as it uses the whole machine resources.

```
wsl --install
wsl --set-default-version 2
```

As we are now running a full linux distribution on our windows machine, the next steps will be the same for all platforms, being windows, mac, or linux.

1- Install Homebrew, this is a package manager for macOS and Linux, it will allow you to install a lot of packages and tools easily and it's always up to date.

```
set install_script (curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)
```

Include HomeBrew Path in your shell profile

```
Change 'YourUserName' with the device username

(echo; echo 'eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"') >> /home/YourUserName/.bashrc
eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"
```

2- Install build-essential, this is a package that contains a list of essential packages for building software and we will need it to compile some plugins. This step is not needed for macOS users.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
```

3- Install NVIM, we will install NVIM using Homebrew, this will install the latest version of NVIM and all its dependencies.

```
brew install nvim
```

4- Install Node and NPM, needed for web development plugins and some language servers.

```
brew install node
brew install npm
```

5- Install GIT, we will need GIT to clone repositories.

```
brew install git
```

6- Install FISH, this is a shell that is highly customizable and has a lot of features, I recommend using it as your default shell. Some of it's amazing features are autocompletion and syntax highlighting.

```

```bash
brew install fish
// set as default:

which fish
// this will return a path, let‘s call it whichFishResultingPath

// add it as an available shell
echo whichFishResultingPath | sudo tee -a /etc/shells

// set it as default
sudo chsh -s wichFishResultingPath
```

7- Install Oh My Fish, this is a framework for fish shell, it will allow you to install themes, plugins, and configure your shell easily.

```
curl https://raw.githubusercontent.com/oh-my-fish/oh-my-fish/master/bin/install | fish
```

8- Install the following dependencies needed to execute LazyVim

```
brew install gcc
brew install fzf
brew install fd
brew install ripgrep
```
```

9- Install Zellij, this is a terminal multiplexer, it will allow you to split your terminal in multiple panes and execute multiple commands at the same time.

```
brew install zellij
```

10 - Install Wezterm, this is a terminal emulator, it is highly customizable and has a lot of features, I recommend using it as your default terminal. One of the strongest features is the GPU acceleration, it will make your terminal faster and more responsive and that it's written in lua, the same language that LAZYVIM uses.

```
https://wezfurlong.org/wezterm/index.html
```

11- Install Iosevka Term Nerd Font, this is a font that is highly customizable and has a lot of features, I recommend using it as your default font. It has a lot of ligatures and special characters that will make your terminal look amazing. A nerd font is a font that has a lot of special characters and ligatures that will make your terminal look amazing and it's needed to render icons.

```
https://github.com/ryanoasis/nerd-fonts/releases/download/v3.1.1/IosevkaTerm.zip
```

12- Now let me share my custom repository that contains all my configurations for NVIM, FISH, Wezterm, and Zellij.

```
https://github.com/Gentleman-Programming/Gentleman.Dots
```

Just follow the steps and you will have a fully customized and gentlemanized terminal and code editor. One last thing before continuin, we will use some plugins that are already configured inside the repository and are managed by LazyVim, an amazing package manager that will allow you to install and update plugins easily.

Now that we have everything... let's start learning how to configure NVIM !

Configuration

As we are using my custom repository, all the configurations are already done, but I will explain how to configure NVIM and how to install plugins and create your own configurations.

As previously said, we are using LazyVim, http://www.lazyvim.org/, an amazing package manager that will allow you to install and update plugins smoothly like butter, it also provides already configured plugins that will make your life easier.

But first, let's learn how to install plugins manually.

If you see the nvim folder structure you will find a "plugins" folder, it will contain a number of files representing each one of the installed plugins by our hand.

Each file will contain the plugin name and the repository URL. To install a plugin manually, you will need to create a new file inside the "plugins" folder and add the following content.

```
return {
    "repository-url",
}
```

And that's it, the next time you open NVIM, the plugin will be installed automatically.

To access a LazyVim management window, just open nvim using the command "nvim" at your terminal and then type the following command ":LazyVim", this will open a window with all the installed plugins and their status, you can install, update, and remove plugins using this window.

Now, to acess all extra already configured plugins provided by LazyVim, just type the following command ":LazyVimExtra", this will open a window with all the available plugins, you can install, update, and remove plugins using this window.

To install a new programming language, type ":MasonInstall" and select the language you want to install, this will install all the necessary plugins and configurations for that language and that's it, you are ready to go.

To stablish new keybindings, just open the "keymaps.lua" file inside the "lua" folder and add the following content.

```
vim.keymap.set('mode', 'whatDoYouWantToPress', 'WhatDoYouWantToDo')
```

The mode represents the mode you are going to use, it can be "n" for normal mode, "i" for insert mode, "v" for visual mode, and "c" for command mode. The second parameter represents the key you want to press, and the third parameter represents the action you want to do.

You shuld come back later to this part after we touch the basics of vim modes.

Basics

Nvim has 4 modes, Normal, Insert, Visual, and Command. Each one of them has its own purpose and shortcuts.

Normal Mode

At this mode, you can navigate through the text, delete, copy, paste, and execute commands. You can enter this mode by pressing the "ESC" key.

In resume, this is the mode you will spend most of your time and where we will move across our code.

Horizontal Movement

To navigate, we will NOT use the arrow keys, we will use the "h" key to move left, the "j" key to move down, the "k" key to move up, and the "l" key to move right. This is the most efficient way to navigate through the text and it will make you look like a pro.

I really recommend you to use the "hjkl" keys to navigate, it will make you more efficient and you will not need to move your hands from the home row. Efficiency is the name of the game.

To jump to the beginning of the line, use the "0" key, to jump to the end of the line, use the "$" key. To jump to the beginning of the file, use the "gg" keys, to jump to the end of the file, use the "G" key.

It's always the same, if you press a command you will do something, and if you press "Shift" while doing it, you will do the opposite.

To correctly move horizontally throug a line, you will need to use the "w" key to jump to the beginning of the next word, the "b" key to jump to the beginning of the previous word, the "e" key to jump to the end of the next word, and the "ge" key to jump to the end of the previous word.

You can also use the amazing "f" key to jump to a specific character in the line, just press "f" and then the character you want to jump to, and that's it, you are there. And I said before, if you use "Shift" while doing it, you will do the opposite, moving to the previous ocurrency.

You can also use the "s" key to search for a character, this is using a plugin called "Sneak", it will allow you to search for a character and jump to it after pressing the key that will appear next to all the ocurrences.

Vertical Movement

To navigate vertically, you can use the "Ctrl" key with the "u" key to move up half a page, the "Ctrl" key with the "d" key to move down half a page, the "Ctrl" key with the "b" key to move up a page, and the "Ctrl" key with the "f" key to move down a page. This is the most efficient way to navigate through the text as we don't know where that particular piece of logic is, so we can move pretty quickly this way and find what we are looking for.

Another great way of navigating vertically is using the "Shift" key with the " and " keys, this will allow you to jump to the next or previous paragraph, and this is one of the things that makes me love NVIM, if your code is clean and correctly indented, you will be able to jump through the code really quickly and find what you are looking for, IT'S TEACHING YOU TO WRITE CLEAN CODE !!

If you want to jump to a particular line, you can use the ":" key, this will open the command mode, and then you can type the line number you want to jump to, and that's it, you are there.

Visual Mode

This mode is used to select text, you can enter this mode by pressing the "v" key. You can use the same commands as the normal mode, but now you can select text. You can also use the "Shift" key with the "v" key to select the whole line.

Again we can use motions to select text, for example, if we want to select the next 10 lines, we can use the "10j" command, and if we want to select the next 10 words, we can use the "10w" command. And if we want to select the next 10 characters, we can use the "10l" command.

Block Visual Mode

This mode is used to select a block of text, you can enter this mode by pressing the "Ctrl" key with the "v" key. You can use the same commands as the normal mode, but now you can select a block of text.

Again we can use motions to select text, for example, if we want to select the next 10 lines, we can use the "10j" command, and if we want to select the next 10 words, we can use the "10w" command. And if we want to select the next 10 characters, we can use the "10l" command.

A block of text is a rectangle of text, and you can copy, paste, and delete it. You can also use the "Shift" key with the "I" key to insert text in a block, and the "Shift" key with the "A" key to append text in a block.

It's also useful to write a lot of lines at the same time, for example, if you want to write a comment in multiple lines, you can use the "Ctrl" key with the "v" key to select the lines you want to write the comment, and then use the "Shift" key with the "I" key to insert the comment, and that's it, you are done after you press the "ESC" key.

Visual Line Mode

This mode is used to select a line of text, you can enter this mode by pressing the "Shift" key with the "v" key. You can use the same commands as the normal mode, but now you can select a line of text.

Again we can use motions to select text, for example, if we want to select the next 10 lines, we can use the "10j" command, and if we want to select the next 10 words, we can use the "10w" command. And if we want to select the next 10 characters, we can use the "10l" command.

Insert Mode

This is the mode you will use to write text, you can enter this mode by pressing the "i" key. You can use the same commands as the normal mode, but now you can write text. You can also use the "Shift" key with the "i" key to insert text at the beginning of the line, and the "A" key to append text at the end of the line, and the same if you want to start writting at a specific character, you can use "i" to insert before the character and "a" to append after the character.

You can also use the "o" key to insert a new line below the current line, and the "O" key to insert a new line above the current line. Using the "Ctrl" key with the "w" key will delete the last word, and using the "Ctrl" key with the "u" key will delete the last line while being in insert mode.

Another useful command is the "Ctrl" key with the "n" key, this will autocomplete the text you are writing, and it's really useful when you are writing code. And if you want to exit the insert mode, you can use the "ESC" key.

Command Mode

This mode is used to execute commands, you can enter this mode by pressing the ":" key. Here is were we can exit NVIM !!! Just do ":q" and that's it ! if you have changes, first save them by using ":w" and if you want to force the exit ":q!".

Another cool thing is that you can do more than one command at once, for example, if you want to save and exit, you can use ":wq".

One cofiguration I recommend it's setting the number of lines to relative by doing ":set relativenumber", this will allow you to see the line number relative to the line you are in, and it's really helpful to know where you are in the file. You can do this by typing the following command ":set relativenumber", and we want this as we can move to a specific line really quickly by using a number and the direction we want to go to, for example, if we want to jump to the 10th line above us, we can use the "10k" command, and if we want to jump to the 10th line below us, we can use the "10j" command.

Nvim Motions

And this introduces the concept of "Motions" in NVIM, each command we type is a motion, and it's created by combining a number, a direction, and a command. For example, if we want to delete the next 10 lines, we can use the "10dd" command, and if we want to copy the next 10 lines, we can use the "10yy" command. This is the most efficient way to navigate through the text and and one of the strongest features of NVIM.

Now let's use what we have learned to delete, copy, and paste text.

To delete text, we can use the "d" key, and then the motion we want to use, for example, if we want to delete the next 10 lines, we can use the "10dd" command, and if we want to delete the next 10 words, we can use the "10dw" command. And if we want to delete the next 10 characters, we can use the "10dl" command, and if we want to delete the whole line we can use the "dd" command.

To copy text, we can use the "y" key, and then the motion we want to use, for example, if we want to copy the next 10 lines, we can use the "10yy" command, and if we want to copy the next 10 words, we can use the "10yw" command. And if we want to copy the next 10 characters, we can use the "10yl" command, and if we want to copy the whole line we can use the "yy" command.

To paste text, we can use the "p" key, this will paste the text after the cursor, and if we want to paste the text before the cursor, we can use the "P" key.

Registers

And now's the funny thing, have you seen what happens when we delete or copy text ? the text is saved in a register, and we can access it by using the "p" key, and we can access the last deleted text by using the "P" key. This is something a lot of beginners hate because they don't know what a register is or how to access it, so let me explain it to you.

A register is a place where the text is saved, and we can access it by using "Leader" (normally "Space") and the double quote key, sometimes we need to do "Leader" and double quote two times if you layout is International, and a panel will appear with all the registers, and you will see that the lates copied text is saved in the "0" register, so now that we know this, you can access it by using the "0p" command. And if you want to access the last deleted text, you can use the "1p" command.

Buffers

A buffer is a place where the text is saved, and you can access it by using the "Leader" key and "be", and a panel will appear with all the buffers, and you can navigate through them by using the "j" and "k" keys. You can also use the "d" key to delete a buffer.

One way of thinking of buffers is like tabs, you can have multiple buffers open at the same time, and you can navigate through them, each time you open a file a new buffer is created and saved into memory, and if you open the same buffer in two places at the same time you will see that if you change something in one buffer, it will change in the other buffer too.

There's a special command I created so you can clear all buffers but the current one for those special times where you have been coding for hours and the performance is a little bit slow, you can do "Leader" and "bq".

Marks

Marks are amazing, you can create a new mark by using the "m" key and then a letter, for example, if you want to create a new mark in the current line, you can use the "ma" command, and if you want to jump to that mark, you can use the "`a" command.

If you want to delete a mark, do ":delm keyOfTheMark", and to delete ALL the marks ":delm!" .Marks are saved in the current buffer, and you can use them to navigate through the text quickly.

Recordings

Now this is amazing and super useful, let's say we need to do an action multiples times and its super tedious to do so, what NVIM provides is a way to replicate a set of commands by creating a macro, you can start recording by using the "q" key and then a letter, for example, if you want to start recording a macro in the "a" register, you can use the "qa" command, and then you can do the actions you want to replicate, and then you can stop recording by using the "q" key.

To replay the macro, you can use the "@" key and then the letter, for example, if you want to replay the macro in the "a" register, you can use the "@a" command. This is super useful and will make you more efficient.

And again you can use motions with your recordings, for example, if you want to delete the next 10 lines and copy them, you can use the "qad10jyy" command, and then you can replay the macro by using the "@a"

command, and also you can replicate the macro multiple times by using the "10@a" command.

Algorithms the Gentleman Way

Big O Notation

"How code slows as data grows"

- Performance of an algorithm depends on the amount of data it is given.
- Number of steps needed to complete. Some machines run algorithms faster than others so we just take the number of steps needed.
- Ignore smaller operations, constants. O(N + 1) → O(N) where N represents the amount of data.

```
function sum(n: number): number {
  const sum = 0;

  for(let i = 0; i < n; i++) {
    sum += i;
  }

  return sum;
}

// if n equals 10, then O(N) is 10 steps, if in equals 100, then O(N) is 100 steps
```

Here we can see that O(N) is linear which means that the amount of steps depends on the number of data we are given.

```
function sayHi(n: string):  string{
  return `Hi ${n}`
}

// if n equals 10, then O(1) is 3 step... 3 ? YES 3 steps

// 1 - Create new string object to store the result (allocating memory for the new string)
// 2 - Concatenate the 'Hi' string with the result.
// 3 - Return the concatenated string.
```

But now we have O(1) as the amount of steps does not depend on the amount of data we are given, it will always be 1.

Needed previous knowledge

- The 'Log' of a number is the power to which the base must be raised to produce that number. For example, the log base 2 of 8 is 3 because 2^3 = 8.
- 'Linear' means that the number of steps grows linearly with the amount of data.
- The 'Quadratic' of a number is the square of that number. For example, the quadratic of 3 is 9 because 3^2 = 9.
- 'Exponential' of a number is the power of the base raised to that number. For example, the exponential of 2 to the power of 3 is 8 because 2^3 = 8.
- 'Factorial' of a number is the product of all positive integers less than or equal to that number. For example, the factorial of 3 is 6 because 3!!) - Factorial time - The number of steps grows factorially (brute force algorithms, those which try all possible solutions)

Example with N equal to 1000:

```bash
- O(1) - 1 step
- O(log N) - 10 steps
- O(N) - 1000 steps, a thousand steps
- O(N log N) - 10000 steps, a ten thousand steps
- O(N^2) - 1000000 steps, a million steps
- O(2^N) - 2^1000 steps
- O(N!) - 1000! steps, factorial of 1000
```

The main idea is that we want to avoid exponential and factorial time algorithms as they grow very fast and are not efficient at all, UNLESS we are sure that the amount of data we are given is very small as it can actually be faster than other algorithms.

Letter grade for Big O Notation, from best to worst, taking in consideration we are using a big dataset of data:

```
- O(1) - Constant time - A
- O(log N) - Logarithmic time - B
- O(N) - Linear time - C
- O(N log N) - Linearithmic time - D
- O(N^2) - Quadratic time - F
- O(2^N) - Exponential time - F
- O(N!) - Factorial time - F
```

Examples using code

O(1) - Constant time

```
function sayHi(n: string):  string{
  return `Hi ${n}`
}
```

Here's why it's O(1):

- The algorithm performs a constant amount of work, regardless of the size of the input.
- The number of steps needed to complete the algorithm does not depend on the input size.

Therefore, the time complexity of the algorithm is O(1) in all cases.

O(log N) - Logarithmic time

```
// having the following array that represents the numbers from 0 to 9 in order
const arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

// we want to find the index of a number in the sorted array
function binarySearch(arr: number[], target: number): number {
  // initialize left and right pointers
  let left = 0;
  let right = arr.length - 1;

  // while left is less or equal to right we keep searching for the target
  while (left <= right) {
    // get the middle of the array to compare with the target
      // we iterate using the middle of the array to find the target because we know the array is sor
    const mid = Math.floor((left + right) / 2); // middle index
    const midValue = arr[mid]; // middle value

    // if the middle value is the target, return the index
    if (midValue === target) {
      return mid;
    }

    // if the middle value is less than the target, we search the right side of the array by updating
    if (midValue < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1; // target not found
}
```

In binary search, the algorithm continually divides the search interval in half until the target element is found or the search interval becomes empty. With each iteration, the algorithm discards half of the search space based on a comparison with the middle element of the current interval.

Here's why it's O(log N):

- In each iteration of the while loop, the search space is halved.
- This halving process continues until the search space is reduced to a single element or the target is found.
- Since the search space is halved with each iteration, the number of iterations required to reach the target element grows logarithmically with the size of the input array.

Thus, the time complexity of binary search is O(log N) on average.

O(N) - Linear time

```
function sum(n: number): number {
  const sum = 0;
  for(let i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}
```

Here's why it's O(N):

- The algorithm iterates over the input array once, performing a constant amount of work for each element.
- The number of iterations is directly proportional to the size of the input array.
- As the input size increases, the number of steps needed to complete the algorithm grows linearly.

Therefore, the time complexity of the algorithm is O(N) in the worst-case scenario.

O(N log N) - Linearithmic time

```
// having the following array
const arr = [5, 3, 8, 4, 2, 1, 9, 7, 6];

// we want to sort the array using the quick sort algorithm
function quickSort(arr: number[]): number[] {
  // first we check if the array has only one element or no elements
  if (arr.length <= 1) {
    return arr;
  }

  // we get the pivot as the last element of the array, the pivot is the element we are going to comp
  const pivot = arr[arr.length - 1];
```

```
  // we create two arrays, one for the elements less than the pivot and another for the elements grea
  const left = [];
  const right = [];

  // we iterate through the array and compare each element with the pivot
  for (let i = 0; i < arr.length - 1; i++) {
    // if the element is less than the pivot, we add it to the left array
    if (arr[i] < pivot) {
      left.push(arr[i]);
    } else {
      // if the element is greater than the pivot, we add it to the right array
      right.push(arr[i]);
    }
  }

  // we recursively call the quickSort function on the left and right arrays and concatenate the resu
  return [...quickSort(left), pivot, ...quickSort(right)];
}
```

Here's why it's O(N log N):

- The algorithm partitions the array into two subarrays based on a pivot element and recursively sorts these subarrays.
- Each partitioning step involves iterating over the entire array once, which takes O(N) time. However, the array is typically divided in a way that the size of the subarrays reduces with each recursive call. This results in a time complexity of O(N log N) on average.

O(N^2) - Quadratic time

```
// having the following array
const arr = [5, 3, 8, 4, 2, 1, 9, 7, 6];

// we want to sort the array using the bubble sort algorithm
function bubbleSort(arr: number[]): number[] {

  // we iterate through the array
  for (let i = 0; i < arr.length; i++) {

    // we iterate through the array again
    for (let j = 0; j < arr.length - 1; j++) {

      // we compare adjacent elements and swap them if they are in the wrong order
      if (arr[j] > arr[j + 1]) {

        // we swap the elements
        const temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
```

```
      }
    }
  }
  return arr;
}
```

Here's why it's O(N^2):

- Bubble sort works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order.
- In the worst-case scenario, where the array is in reverse sorted order, bubble sort will need to make N passes through the array, each pass requiring N-1 comparisons and swaps.
- This results in a total of N * (N-1) comparisons and swaps, which simplifies to O(N^2) in terms of time complexity.

O(2^N) - Exponential time

```
// we want to calculate the nth Fibonacci number using a recursive algorithm
function fibonacci(n: number): number {

  // we check if n is 0 or 1 as the base case of the recursion because the Fibonacci sequence starts
  if (n <= 1) {
    return n;
  }

  // we recursively call the fibonacci function to calculate the nth Fibonacci number
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Here's why it's O(2^N):

- In each recursive call to the fibonacci function, two additional recursive calls are made with n - 1 and n - 2 as arguments.
- This leads to an exponential growth in the number of recursive calls as n increases.
- Each level of recursion branches into two recursive calls, resulting in a binary tree-like structure of recursive calls.
- The number of function calls doubles with each level of recursion, leading to a total of 2^N function calls when calculating the nth Fibonacci number.

Therefore, the time complexity of the algorithm is O(2^N) in the worst-case scenario.

O(N!) - Factorial time

```
// having the following array
const arr = [1, 2, 3];

// we want to generate all permutations of a given array using a recursive algorithm
function permute(arr: number[]): number[][] {
  // base case: if the array has only one element, return it as a single permutation
  if (arr.length <= 1) {
    return [arr];
  }

  // initialize an empty array to store permutations
  const result: number[][] = [];

  // iterate over each element in the array
  for (let i = 0; i < arr.length; i++) {
    // generate all permutations of the array excluding the current element
    const rest = arr.slice(0, i).concat(arr.slice(i + 1));
    const permutations = permute(rest);

    // add the current element to the beginning of each permutation
    for (const perm of permutations) {
      result.push([arr[i], ...perm]);
    }
  }
  return result;
}
```

Here's why it's O(N!):

- In each recursive call to the permute function, the algorithm generates permutations by selecting each element of the array as the first element and then recursively generating permutations of the remaining elements.
- The number of permutations grows factorially with the size of the input array.
- For each element in the array, there are (N-1)! permutations of the remaining elements, where N is the number of elements in the array.
- Therefore, the total number of permutations is N * (N-1) * (N-2) * ... * 1, which is N factorial (N!).

Hence, the time complexity of the algorithm is O(N!) in the worst-case scenario.

Worst-case, Best-case, and Average-case Complexity

- The worst-case time complexity represents the maximum number of steps an algorithm takes to complete for a given input size. It provides an upper bound on the algorithm's performance. It is the most commonly used measure of time complexity in job interviews.

- The best-case time complexity represents the minimum number of steps an algorithm takes to complete for a given input size. It provides a lower bound on the algorithm's performance. It is less informative than the worst-case complexity and is rarely used in practice.
- The average-case time complexity represents the expected number of steps an algorithm takes to complete for a given input size, averaged over all possible inputs. It provides a more realistic estimate of an algorithm's performance than the worst-case complexity. However, calculating the average-case complexity can be challenging and is often avoided in favor of the worst-case complexity.

Space complexity

The space complexity of an algorithm is a measure of the amount of memory it requires to run as a function of the input size. It is typically expressed in terms of the maximum amount of memory used by the algorithm at any point during its execution.

It is important to distinguish between time complexity and space complexity, as an algorithm with good time complexity may have poor space complexity, and vice versa. For example, a recursive algorithm with exponential time complexity may also have exponential space complexity due to the recursive calls consuming memory.

But something to have in mind is that space complexity is not as important as time complexity, as memory is usually cheaper than processing power and in real life scenarios, we usually skip the space complexity analysis and focus on time complexity.

Imagine you're at a traditional Argentine barbecue, known as an "asado." You've got limited space on the grill (similar to limited memory in computing), and you want to optimize how much meat you can cook at once.

Now, let's compare the meat (or "carne") to the data in an algorithm. When you're cooking, you have to consider how much space each cut of meat takes up on the grill. Similarly, in computing, algorithms have to consider how much memory space they need to store and process data.

But here's the thing: at an asado, the most important factor is usually how quickly you can cook the meat and serve it to your guests. Similarly, in computing, the time it takes for an algorithm to run (time complexity) is often the most critical factor for performance.

So, while it's essential to be mindful of how much space (or "espacio") your algorithm uses, it's usually more exciting to focus on how efficiently it can solve a problem in terms of time.

Of course, in some situations, like if you're grilling on a tiny balcony or cooking for a huge crowd, space becomes more of a concern. Similarly, in computing, if you're working with limited memory resources or on a device with strict memory constraints, you'll need to pay closer attention to space complexity.

But overall, just like at an Argentine barbecue, the balance between time and space complexity is key to creating a delicious (or efficient) outcome!

However, let's talk about how you calculate the space complexity, or "cuánto espacio ocupas" in the case of our barbecue analogy. Just as you'd assess how much space each cut of meat takes up on the grill, in

computing, you need to consider how much memory each data structure or variable in your algorithm consumes.

Here's a basic approach to calculate space complexity:

- **Identify the Variables and Data Structures**: Look at the algorithm and identify all the variables and data structures it uses. These could be arrays, objects, or other types of variables.
- **Determine the Space Used by Each Variable**: For each variable or data structure, determine how much space it occupies in memory. For example, an array of integers will take up space proportional to the number of elements multiplied by the size of each integer.
- **Add Up the Space**: Once you've determined the space used by each variable, add them all up to get the total space used by the algorithm.
- **Consider Auxiliary Space**: Don't forget to account for any additional space used by auxiliary data structures or function calls. For example, if your algorithm uses recursion, you'll need to consider the space used by the call stack.
- **Express Space Complexity**: Finally, express the space complexity using Big O notation, just like you do with time complexity. For example, if the space used grows linearly with the size of the input, you'd express it as O(N). If it grows quadratically, you'd express it as O(N^2), and so on.

So, just as you carefully manage the space on your grill to fit as much meat as possible without overcrowding, in computing, you want to optimize the use of memory to efficiently store and process data. And just like finding the perfect balance of meat and space at an Argentine barbecue, finding the right balance of space complexity in your algorithm is key to creating a delicious (or efficient) outcome!

Example Time !!!

Let's use a simple algorithm to find the sum of elements in an array as an example for calculating space complexity.

```
function sumArray(arr: number[]): number {
    let sum = 0;  // Space used by the sum variable: O(1)

    for (let num of arr) {  // Space used by the loop variable: O(1)
        sum += num;  // Space used by temporary variable: O(1)
    }

    return sum;  // Space used by the return value: O(1)
}
```

In this example:

- We have one variable `sum` to store the sum of elements, which occupies a constant amount of space, denoted as O(1).

- We have a loop variable `num` that iterates through each element of the array. It also occupies a constant amount of space, O(1).
- Within the loop, we have a temporary variable to store the sum of each element with `sum`, which again occupies a constant amount of space, O(1).
- The return value of the function is the sum, which also occupies a constant amount of space, O(1).

Since each variable and data structure in this algorithm occupies a constant amount of space, the overall space complexity of this algorithm is O(1).

In summary, the space complexity of this algorithm is constant, regardless of the size of the input array.

Now let's consider an example where we create a new array to store the cumulative sum of elements from the input array. Here's the algorithm:

```
function cumulativeSum(arr: number[]): number[] {
    const result = []; // Space used by the result array: O(N), where N is the size of the input array
    let sum = 0;  // Space used by the sum variable: O(1)
    for (let num of arr) {  // Space used by the loop variable: O(1)
        sum += num;  // Space used by temporary variable: O(1)
        result.push(sum);  // Space used by the new element in the result array: O(1), but executed N t
    }
    return result;  // Space used by the return value (the result array): O(N)
}
```

In this example:

- We have a variable `result` to store the cumulative sum of elements, which grows linearly with the size of the input array `arr`. Each element added to `result` contributes to the space complexity. Therefore, the space used by `result` is O(N), where N is the size of the input array.
- We have a loop variable `num` that iterates through each element of the input array `arr`, which occupies a constant amount of space, O(1).
- Within the loop, we have a temporary variable `sum` to store the cumulative sum of elements, which occupies a constant amount of space, O(1).
- Inside the loop, we add a new element to the `result` array for each element in the input array. Each push operation adds an element to the array, so it also contributes to the space complexity. However, since it's executed N times (where N is the size of the input array), the space used by the push operations is O(N).
- The return value of the function is the `result` array, which occupies O(N) space.

Overall, the space complexity of this algorithm is O(N), where N is the size of the input array. This is because the space used by the `result` array grows linearly with the size of the input.

Arrays

When we talk about arrays, we usually think of ordered collections of elements, right? But in JavaScript, arrays are actually objects. So, what's a real array, you might ask? Well, a true array is a contiguous block of memory where each element takes up the same amount of space.

In a real array, accessing an element is super quick—it's a constant time operation, meaning it takes the same amount of time no matter how big the array is. Why? Because you can calculate exactly where each element is in memory.

Now, let's contrast that with JavaScript arrays. They're implemented as objects, where the indexes are the keys. So, when you access an element in a JavaScript array, you're actually accessing a property of an object. This means accessing elements isn't as snappy—it's a linear time operation because the JavaScript engine has to search through the object keys to find the right one.

To find the memory location of an element in a real array, you use a simple formula:

```
const index = base_address + offset * size_of_element;
```

Here, the **`index`** is what we usually call the index, but it's more like an offset. The **`base_address`** is the starting point of the array in memory, and **`size_of_element`** is, well, the size of each element.

With this formula, every time you search for an element, you're performing a constant time operation because it doesn't matter how big the array is—the math stays the same.

Now, let's illustrate this with some code:

```
// Let's create a new array with 5 elements
const a = [1, 2, 3, 4, 5];

// Calculate the total space the array occupies in memory
const totalSpace = array.length * 4; // assuming each element occupies 4 bytes

// Choose an index
const index = 3;

// Calculate the memory location of the element
const sizeOfEachElement = 4; // each element occupies 4 bytes
const baseAddress = array; // the reference to the array itself
const offset = index * sizeOfEachElement;
const memoryLocation = baseAddress + offset;

// Access the element at the calculated memory location
const elementAtIndex = memoryLocation;

console.log("Value at index", index, "is:", elementAtIndex); // Value at index 3 is: 4
```

In this example, we're simulating how a real array works under the hood. We calculate the memory location of an element using the index, and then we access that memory location to get the element.

Now, let's visualize this with Node.js, where we can peek into what's happening in memory:

```
// Create a new array buffer in Node.js
const buffer = new ArrayBuffer(16); // 16 bytes of memory

// Check the size of the buffer (in bytes)
console.log("buffer.byteLength: " + buffer.byteLength); // Output: buffer.byteLength: 16

// Now let's create a new Int8Array, a typed array of 8-bit signed integers
const int8Array = new Int8Array(buffer);

// Log the contents of the Int8Array (all zeros initially)
console.log(int8Array); // Output: Int8Array(16) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

// Each element in the Int8Array represents a single byte in the buffer

// Now let's change the value of the first element of the array to 1
int8Array[0] = 1;

// Log the contents of the Int8Array and buffer again
console.log(int8Array); // Output: Int8Array(16) [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
console.log(buffer); // Notice how the underlying buffer is also modified (first byte becomes 1)
// Output: Buffer after change: ArrayBuffer { [Uint8Contents]: <01 00 00 00 00 00 00 00 00 00 00 00 00

// TypedArrays provide a different view of the same memory

// Now let's create a 16-bit signed integer array (uses 2 bytes per element)
const int16Array = new Int16Array(buffer);

// Log the contents of the Int16Array (initial values based on buffer)
console.log(int16Array); // Output: Int16Array(8) [1, 0, 0, 0, 0, 0, 0, 0]

// The Int16Array has fewer elements because it uses more bytes per element, 2 bytes

// Again, let's change a value (third element) and see the effects
int16Array[2] = 4000;

// Log the contents of all three arrays and the buffer
console.log(int16Array); // Output: Int16Array(8) [1, 0, 4000, 0, 0, 0, 0, 0]
console.log(buffer); // Notice how multiple bytes in the buffer are modified (5th and 6th bytes)
// Output: ArrayBuffer { [Uint8Contents]: <01 00 00 00 a0 0f 00 00 00 00 00 00 00 00 00 00> } (contents
console.log(int8Array); // The view of the Int8Array is also affected  (5th and 6th bytes change)
// Output: Int8Array(16) [1, 0, 0, 0, -96, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] (5th byte is -96, 6th byte

// Now let's create a 32-bit signed integer array (uses 4 bytes per element)
const int
```

```
32Array = new Int32Array(buffer);

// Log the contents of the Int32Array (initial values based on buffer)
console.log(int32Array); // Output: Int32Array(4) [1, 4000, 0, 0]

// Even fewer elements due to the larger size of each element, 4 bytes

// Let's change the value and observe the effects
int32Array[2] = 100000;

// Log the contents of all three arrays and the buffer
console.log(int32Array); // Output: Int32Array(4) [1, 4000, 100000, 0]
console.log(buffer); // ArrayBuffer { [Uint8Contents]: <01 00 00 00 a0 0f 00 00 10 27 00 00 00 00 00 00
console.log(int16Array); // Int16Array(8) [1, 0, 4000, 0, 10000, 0, 0, 0] (4th element is now 10000)
console.log(int8Array); // Int8Array(16) [1, 0, 0, 0, -96, 15, 0, 0, 16, 39, 0, 0, 0, 0, 0, 0] (9th, 10
```

In this Node.js example, we're creating a buffer of 16 bytes, which is like a block of memory. Then, we create different typed arrays to view this memory differently—as arrays of different types of integers. Modifying one of these arrays also changes the underlying buffer, showing how they're different views of the same memory.

Now, let's circle back to why we're seeing decimals. We'll need to understand two's complement representation first.

So, two's complement is like the magic trick we use in computing to handle both positive and negative numbers using the same binary system. Picture this: in our binary numbers, the first digit from the left (the big boss, you know?) decides if the number is positive or negative. If it's 0, it's positive, and if it's 1, it's negative.

Now, for the positive numbers, it's easy peasy. You just write them down in binary, as usual. For example, number 3 in 8-bit binary is 00000011. No problem there, right?

But when it comes to negative numbers, we need a trick. We take the positive number's binary representation, flip all the bits (0s become 1s and 1s become 0s), and then add 1 to the result. This gives us the two's complement of the negative number. Let's say we want to represent -3. First, we start with 3's binary representation, which is 00000011. Then, we flip all the bits to get 11111100, and finally, we add 1 to get 11111101. That's the two's complement of -3 in 8-bit binary.

Now, why do we do all this? Well, it's because in computing, we like things to be tidy and consistent. With two's complement, we can use the same rules for adding and subtracting both positive and negative numbers, without needing to worry about special cases. It keeps our math nice and clean, just like sipping on yerba mate on a sunny day in Buenos Aires.

Now we can see why we are seeing decimals !

When working with 8 bits, the values are being represented in decimal, as the range of an 8-bit signed integer is -128 to 127. If the value overflows, it wraps around the range:

When you're dealing with 8 bits, the values are shown in decimal because, you know, an 8-bit signed integer can only hold numbers from -128 to 127. It's like having a limited space in a crowded bus—you can only fit so many people!

Now, imagine you're trying to squeeze in the number 4000. In binary, that's `111110100000`. But here's the thing: our bus only goes up to 127. So, when you try to cram 4000 in there, it's like trying to fit a football team into a tiny car—it's just not gonna happen.

Now, when you try to jam 4000 into our 8-bit bus, it overflows. But instead of causing chaos, it does something pretty neat: it wraps around. You see, the bus route starts again from the beginning, like a never-ending loop.

This "wrapping around" is where things get interesting. The leftmost bit, the big boss of the number, flips its sign. So, instead of being positive, it becomes negative. It's like turning the bus around and going in the opposite direction!

Now, we use our previous trick called two's complement to figure out the new number. First, we flip all the bits of `111110100000` to get `000001011111`. Then, we add 1 to this flipped number, giving us `000001100000`.

And voilà! That binary number `000001100000` represents -96 in decimal. So, even though you tried to squeeze in 4000, our little bus handles it like a champ and tells you it's -96. That's the magic of overflow and wrap-around in an 8-bit world!

How to think?

When you're up to your eyeballs in code, understanding the concepts and applying them to crack those problems is the name of the game. My advice? Pour your heart and soul into commenting your code, explaining each step inside the method, and THEN dive into implementation.

That way, you'll know what needs doing and just need to figure out how to pull it off.

Example:

```
function sumArray(arr: number[]): number {
        // Initialize the sum variable to store the sum of elements
        let sum = 0;  // O(1)

        // Iterate over each element in the array
        for (let num of arr) {  // O(N)
                // Add the current element to the sum
                sum += num;  // O(1)
        }

        // Return the final sum
        return sum;  // O(1)
}
```

Linear Search

It's the bread and butter of algorithms, but do you really know how it struts its stuff?

Here's the scoop: we're sashaying through each element of a collection, asking if the element we're hunting down is the one staring us in the face. The JavaScript bigwig that uses this algorithm? The `indexOf` method, baby.

```
function indexOf(arr: number[], target: number): number {
    // Iterate over each element in the array
    for (let i = 0; i < arr.length; i++) {

        // Check if the current element is equal to the target
        if (arr[i] === target) {

            // If it is, return the index of the element
            return i;
        }
    }

    // If the target is nowhere to be found, return -1
    return -1;
}
```

So... what's the worst-case scenario here that'll give us the Big O Notation? That we come up empty-handed or that the element we're after is lounging at the end of the array, making this algorithm O(N). As N gets bigger, so does the complexity - it's like watching your waistline after devouring a mountain of alfajores.

Binary Search

Now, this bad boy right here is the cream of the crop, the bee's knees, and one of the heavy hitters in professional interviews and coding showdowns. Let's unpack when and how to unleash its power:

The name of the game? Divide and conquer, baby! Instead of playing tag with each element of the array, we slice that sucker in half and ask, "Hey, are you on the left or right?" Then, we keep slicing and dicing until we've nabbed our elusive target.

```
function binarySearch(arr: number[], target: number): number {
    // Initialize the left and right pointers
    // left starts at the beginning of the array
    // right starts at the end of the array
    let left = 0;
    let right = arr.length - 1;

    // Keep on truckin' as long as the left pointer is less than or equal to the right pointer
    while (left <= right) {
```

```
        // Calculate the middle index of the current search space
        const mid = Math.floor((left + right) / 2);

        // Check if the middle element is the target
        if (arr[mid] === target) {

            // If it is, return the index of the element
            return mid;
        } else if (arr[mid] < target) {

            // If the middle element is less than the target, swing right
            left = mid + 1;
        } else {

            // If the middle element is greater than the target, veer left
            right = mid - 1;
        }
    }
}
```

And whit this example you will see, clear as the light that guide us in the darkest nights that I'm not lying:

```
// we need to find the index of the target number inside an array of 1024 elements

1024 / 2 = 512 // we halve it and see that the target number is in the right half
512 / 2 = 256 // we halve it again and see that the target number is in the right half
256 / 2 = 128 // we halve it again and see that the target number is in the right half
128 / 2 = 64 // we halve it again and see that the target number is in the right half
64 / 2 = 32 // we halve it again and see that the target number is in the right half
32 / 2 = 16 // we halve it again and see that the target number is in the right half
16 / 2 = 8 // we halve it again and see that the target number is in the right half
8 / 2 = 4 // we halve it again and see that the target number is in the right half
4 / 2 = 2 // we halve it again and see that the target number is in the right half
2 / 2 = 1 // we halve it again and see that the target number is in the right half
1 / 2 = 0.5 // we can't halve it anymore, so we stop

// here comes the magic, if we count the number of steps we have a total of 10 steps to find the target
// do you know what is the logarithm of 1024 in base 2? it's 10 !!
log2(1024) = 10
```

Exercise of the Crystal Ball

The crystal ball problem is a classic mathematical issue that is often tackled in programming interviews. It involves finding the position where two crystal balls meet and shatter when dropped from a certain height. The key here isn't to know about the crystal balls, their falling from a height, or any other detail; we need to grasp the fundamentals and generalize to make it applicable in any scenario. If we can accomplish this, we can represent the height as an array of n elements, where 0 is the start of the fall, n is the maximum height, and the index is the place where the balls collide.

```
// f = not found, t = found or previously encountered
const height = [f, f, f, f, f, f, t, t, t, t, t];
//             start              collision        end of the fall
```

Solution

Let's review the tools we already have to solve this problem:

- **Arrays**: Arrays are data structures containing a set of elements of the same type. In this case, the heights of the balls.
- **Loop**: Loop is a control structure that allows repeating a section of code a number of times. In this case, the number of repetitions is the number of elements in the array.
- **If**: If is a control structure that allows executing a section of code if a condition is true. In this scenario, the condition is whether the element in the array is true or false.
- Linear Search: Linear Search is a searching technique in which an element is searched sequentially in an array by checking each position of the array consecutively.
- Binary Search: Binary Search is a searching technique in which an element is searched in an array by dividing the array into two parts and searching for the element in the part that most fits what we are looking for and so forth.

```
// Linear search
const height = [f, f, f, f, f, f, t, t, t, t, t];

function findCrystalBall(height: number[]): number {
  for (let i = 0; i < height.length; i++) {
    if (height[i] === true) {
      return i;
    }
  }

  return -1;
}
```

The issue with this solution is that it cannot be optimized to work in every case, as linear search checks each position of the array consecutively, meaning if the array has many elements, the execution time will be very long.

So, let's look for a more efficient solution.

```
// Binary search
function findCrystalBall(height: number[]): number {
  let start = 0;
  let end = height.length - 1;
```

```
  while (start <= end) {
    const index = Math.floor((start + end) / 2);

    if (height[index] === true) {
      return index;
    } else if (height[index] === false) {
      start = index + 1;
    } else {
      end = index - 1;
    }
  }

  return -1;
}
```

Does it work? The truth is, no! Because we're not finding the exact moment when the balls meet. Instead, we're returning the first true we find, rather than the very first one. Additionally, if we consider the condition that once a ball breaks, it cannot be used again, the execution time remains linear with the size of the array. In the worst case, we will search through half the height, and if we find a true, we will need to go back to the previous position and determine the exact breaking point, which still results in O(N) complexity.

So let's combine everything we've learned to implement a more incredible and much more efficient solution:

1- We will reduce the size of the section cut so that instead of half the array, it is the square root of N, which significantly reduces the execution time.

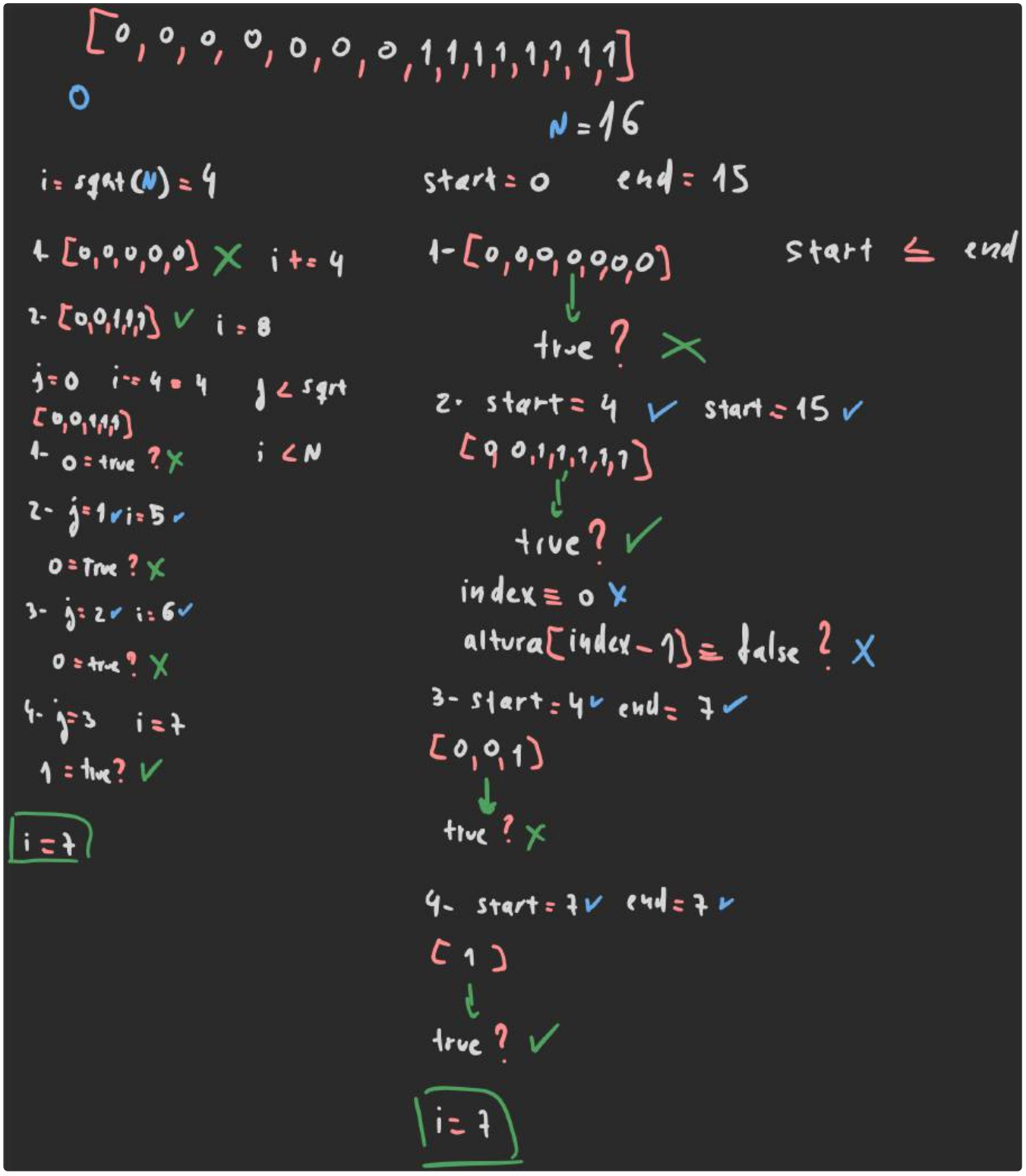
2- Once we find the first true, we will return to the point where the section cut begins (SQRT(N)) and search for the first true found linearly.

```
function findCrystalBall(height: number[]): number {
  const N = height.length;
  const SQRT = Math.floor(Math.sqrt(N));

  // Now we will reduce the size of the section cut so that instead of half the array, it is the square
  // and we traverse it linearly, the square root of N at a time
  let i = SQRT;

  for (; i < N; i += SQRT) {
    if (height[i] === true) {
      break;
    }
  }

  i -= SQRT;

  for (let j = 0; j < SQRT && i < N; i++, j++) {
    if (height[i] === true) {
      return i;
    }
```

```
  }

  return -1;
}

// Binary search
function findCrystalBall(height: number[]): number {
  let start = 0;
  let end = height.length - 1;

  while (start <= end

) {
    const index = Math.floor((start + end) / 2);

    // check if the value at the mid position is 'true' and
    // if it is the first 'true' in the array (the previous element must be 'false' if index is not 0)
    if (height[index] === true)
      if (index === 0 || height[index - 1] === false) return index;
      else end = index - 1;
    else start = index + 1;
  }

  return -1;
}
```

While both solutions work and seem effective, binary search is more efficient!

- N = 1000 steps
- SQRT(N) = 100 steps
- log2(N) = 10 steps



Bubble Sort

Sorting is one of the most common tasks we perform daily as programmers, BUT many people don't really understand what's happening behind the scenes and rely on the famous "magic" functions that are part of every programming language.

Believe it or not, sorting is one of the easiest things we can do! And the best part is that it's also one of the shortest algorithms to write.

The main idea of bubble sort is that it iterates over an array, comparing each element with the next one. If the current element is greater than the next, they are swapped. What's important to note is that at the end of the first iteration, the largest element will have bubbled up to the end of the array. This allows us to iterate through the array again to continue sorting the remaining elements more efficiently, as we can exclude the last element from the previous iteration and so reduce the number of elements to be considered with each pass until the array is sorted.

```
let unsortedArray = [3, 1, 4, 8, 2];

// first iteration
// compare the first with the second and swap them
unsortedArray = [1, 3, 4, 8, 2];
// compare the second with the third and leave them
unsortedArray = [1, 3, 4, 8, 2];
// compare the third with the fourth and leave them
unsortedArray = [1, 3, 4, 8, 2];
// compare the fourth with the fifth and swap them
unsortedArray = [1, 3, 4, 2, 8]; // the largest number, 8, ends up at the end

// second iteration excluding the 8 and iterate again

// compare the first with the second and leave them
unsortedArray = [1, 3, 4, 2];
// compare the second with the third and leave them
unsortedArray = [1, 3, 4, 2];
// compare the third with the fourth and swap them
unsortedArray = [1, 3, 2, 4]; // the next largest number, 4, ends up at the end

// third iteration excluding the 4 and iterate again

// compare the first with the second and leave them
unsortedArray = [1, 3, 2];
// compare the second with the third and swap them
unsortedary = [1, 2, 3]; // the next number, 3, ends up at the end

// fourth iteration excluding the 3 and iterate again

// compare the first with the second and leave them
unsortedArray = [1, 2]; // END
```

If we look closely, we can discern a hidden pattern in the code; after each iteration, the result is N - i, where N is the size of the array and i is the number of iterations we have performed so far.

```
const unsortedArray = [3, 1, 4, 8, 2];

// first iteration N - i = 4
// result of the first iteration = [1, 3, 4, 2]

// second iteration N - i = 3
```

```
// result of the second iteration = [1, 3, 2]

// third iteration N - i = 2
// result of the third iteration = [1, 2]
```

If we generalize the pattern, we can say that the final result would be N - N + 1 because the generalized way to calculate the sum of elements in an array is: ((N + 1) * N) / 2

```
Sum between 1 and 10 is 11
Sum between 2 and 9 is 11
Sum between 3 and 8 is 11
Sum between 4 and 7 is 11
.
.
.
Sum between 5 and 6 is 11
So, we can say that with N being 5 and N + 1 being 6, the final result would be
(6 + 1) * 5 / 2 = 11

Now, if we generalize the pattern within Big O notation, we can say that:

1- N (N + 1) / 2
2- N (N + 1) // removing constants elevating everything to 2
3- N^2 + N
// in Big O, we remove less significant values, if we're talking about VERY large numbers,
// + N is VERY small in comparison
4- N^2

Thus, it remains O(N^2) in Big O notation
```

Code

```
function bubbleSort(array: number[]): number[] {
  // define the size of the array
  const N = array.length;

  // iterate over the array


  for (let i = 0; i < N; i++) {
    // iterate over the array
    for (let j = 0; j < N - 1 - i; j++) {
      if (array[j] > array[j + 1]) {
        // if it is greater, swap the elements
        const temp = array[j];

        // swap the element with the next one
        array[j] = array[j + 1];
        array[j + 1] = temp;
```

```
      }
    }
  }

  return array;
}
```

Gentleman of Code: Mastering Clean Architecture

Introduction to the Book "Gentleman of Code: Mastering Clean Architecture"

Hello, code geniuses! I'm Gentleman, and I bring you an exciting journey through the universe of software architecture and agile methodologies, using the famous pattern of Clean Architecture. This book is a compilation based on my speeches and talks, specially designed to guide you from conceptualization to the practical implementation of scalable and easy-to-maintain systems. Get ready to become Gentlemen of code!

Chapter 1: "Discovering Clean Architecture!"

For those of you just peeking into this world, let me tell you that this is not just some fairy tale, it is one of the best things that could happen to software development! So, let's get started with everything.

What is Clean Architecture? First things first: What's up with Clean Architecture? Well, guys, this is not a Lego game, although it

does resemble one in that everything fits perfectly. Clean Architecture is an architectural pattern, and what makes it special is that it helps you organize your project in a way that, no matter how much time passes, you can maintain and expand it without wanting to pull your hair out. That is, it is not just a simple design pattern, it is a complete philosophy that helps you separate what is business code from infrastructure or UI, making everything more modular and testable.

Where did it come from and where are we going This gem did not come out of nowhere. It was popularized by Uncle Bob Martin, who is something like the Messi of software development. He thought of it as a way to prevent projects from turning into a monster that no one wants to touch. With Clean Architecture, each piece of the system has its place and its responsibilities clearly defined, making even the largest projects easy to manage.

Where are we going with this? To the future, of course! Implementing Clean Architecture in your projects means you are thinking big, in systems that not only work well today but can adapt and expand in the future without drama.

Who am I and where am I? Well, for those who don't know me, I'm Gentleman, passionate about clean code and good practices in software development. I'm here to guide you on this journey of discovery and to make sure that by the end of this book you not only understand what Clean Architecture is, but can implement it and defend it like true code gurus.

Chapter conclusion So, guys, what we're going to be doing throughout this book is applying this clinic of clean architecture. You'll see that it's a beautiful and super useful concept. Not only is it different from what many think, but it really is an architectural solution that will change the way you build software. Get ready, because this is just the beginning and I promise it will be an incredible journey.

In the next chapter, we'll bite into "Separation of Concerns," which is the heart of Clean Architecture. We'll break down how this practice is not only essential for keeping our code clean and tidy, but also the key to making our projects scalable and easy to maintain. So don't peel off, because this is getting good!

Chapter 2: "Separation of Concerns: The Key to an Efficient Architecture"

In this chapter, we'll delve into one of the most fundamental and powerful principles: the Separation of Concerns. So, fasten your seatbelts because we're going to break down how this concept can make your projects beautiful in terms of maintenance and scalability.

What is the Separation of Concerns? First, let's get serious and define what this is. Separation of Concerns is nothing more than a pro strategy for organizing our code. Imagine you're organizing a party and have to keep Boca and River fans separated... well, that's kind of what we do here, but with our code. This technique helps us keep each part of our project focused on a specific task, without meddling in the affairs of others. This, my friends, is the essence of building software that is not only easy to understand but also to modify and expand.

What is the goal of applying it in Clean Architecture? In Clean Architecture, the Separation of Concerns is like the ace of spades in a magician's sleeve. It allows us to structure the project in layers, where each one has a clear responsibility. For example, we have a layer for business logic, another for the user interface, and another for data access. This means that if tomorrow we want to change the database or the user interface, we can do it without the rest of the system suffering a nervous breakdown.

The Advantages: Maintainability and Scalability Now, let's talk about the advantages. Implementing the Separation of Concerns in our architectures not only makes the code more aesthetically pleasing (which also counts, eh!), but it works wonders for maintainability. When each part of the system only deals with its own, updating or fixing bugs becomes much simpler. And not only that, but it also prepares the ground for the system to grow without drama. As we add more functionalities, each one integrates into its respective layer without altering the others, making the system super scalable.

Practical Recommendations To close, I'll leave you with some golden recommendations. If you are implementing Clean Architecture, don't forget to define the interfaces between the layers well. This is like saying, "you, take care of this and don't get involved in that." Keep these interfaces clean and clear, and you'll save yourself a lot of headaches. Also, don't be afraid to reorganize the layers if over time you see that something could improve. Flexibility is key in this game.

Chapter conclusion Well, I hope you now have a clear idea of why the Separation of Concerns is so crucial in Clean Architecture. In the next chapter, we

'll compare design patterns with architectures, so you can see how everything fits into this big puzzle of software development. So don't miss the next episode, because we're going to continue dissecting these topics so you can become true masters of code. See you in the next chapter, cracks!

Chapter 3: "Design Patterns vs. Architectures: Understanding the Differences"

Let's continue to delve into this exciting world of programming. In this chapter, we're going to step on the gas on the difference between design patterns and software architectures. Many people confuse these two terms, but today we're going to make everything clearer than water. So let's get into the ring!

What are design patterns? Let's start with the basics: design patterns are proven solutions to common problems we encounter in software development. Think of them as grandma's cooking recipes, each with its secret to solving a specific problem in the kitchen. In programming, for example, we have the Factory pattern, which helps us create objects without specifying the exact class of the object that will be created. This is pure gold when we want flexibility in our system.

And what is a software architecture? On the other hand, when we talk about software architecture, we are talking about the master plan, the complete structure on which we build our applications. The architecture defines how the system is organized, how the components interact, and how data and flow control are managed in the application. It is, let's say, the blueprint of the house in which we are going to put those furniture (the design patterns).

The key differences The fundamental difference between a design pattern and an architecture is the scope. While a design pattern applies to a specific problem within part of the system, the architecture extends throughout the entire project. It's like comparing changing the upholstery of a chair (design pattern) with designing the entire living room (architecture).

Integration of patterns in the architecture In Clean Architecture, design patterns play a crucial role, but always within the framework that the architecture defines. We use patterns as tools that help us implement the separation of concerns and ensure that each part of the system can be maintained and scaled efficiently. Patterns give us the tactics, but architecture gives us the strategy.

Practical advice

My advice here is simple: learn and master various design patterns, but always keep in mind how these fit within the general architecture of your application. Don't be tempted to apply a pattern just because; each pattern has its place and its time, and knowing when and how to use it is what separates good programmers from true masters.

Chapter conclusion I hope you now have a better understanding of what separates design patterns from software architectures and how both complement each other to create robust and maintainable applications. In our next chapter, we will explore in more depth how Clean Architecture facilitates the maintainability and scalability of our projects. So don't miss it, because we will continue to unravel these concepts so that you become the next star architects of the software world! See you in the next chapter, code geniuses!

Chapter 4: "Maintainability and Scalability with Clean Architecture"

Let's dive into two of the most prized jewels when we talk about software architectures: maintainability and scalability. These are the pillars that make Clean Architecture shine on its own. Let's explore how this

architecture helps us achieve code that is not only a pleasure to maintain but also easy to scale. Here we go!

Maintainability in Clean Architecture

Imagine we have a car where each part is so entangled with the others that changing a spark plug requires dismantling half the engine. Sounds like a nightmare, right? Well, that's what Clean Architecture seeks to avoid in software development. By maintaining a clear and defined separation of responsibilities, each system component can be understood, tested, and modified independently. This is crucial because when it comes time to fix bugs or add improvements, we can get straight to the point without fear of breaking other parts of the system.

How does this translate into practice?

To put it into practice, let's think of a simple example. Suppose we have an application with a user interface, business logic, and a database. In a well-designed architecture, we could change the user interface without touching the business logic or the database. This not only reduces the risk of bugs but also makes development faster and less prone to errors.

Scalability with Clean Architecture

Now, let's talk about scalability. In the world of software, scaling not only means handling more data or users but also adding new functionalities without collapsing under the system's own weight. Clean Architecture makes this easier by allowing us to add new features as if they were independent modules or blocks. Each block communicates with the others through well-defined interfaces, which means we can expand our application without having to rewrite or excessively adjust what already works.

A concrete example of scalability

Let's consider a practical case. If we decide we want to add a payment system to our application

, in a clean architecture, this would be integrated through a service interface, without directly altering the existing business logic. This allows us not only to efficiently integrate the new module but also to test and modify it without impacting the rest of the application.

Chapter conclusion

Understanding and applying maintainability and scalability with Clean Architecture may seem challenging at first, but I assure you that the rewards are worth it. A well-designed system is not only easier to maintain and scale, but also more robust and adaptable to changes, which in the world of software, my friends, is pure gold.

In our next chapter, we will delve deeper into how plugin architecture can further increase the flexibility and adaptability of our systems. So don't miss this next chapter because we will continue to build on this solid foundation to turn you into true software architects. See you there, code masters!

🕶 Chapter 5: "Plugins in Architecture: Flexibility and Adaptability"

In this chapter, we will delve into a super key concept: plugin architecture. This approach gives us tremendous flexibility and adaptability in our projects. Ready, set, go!

What is Plugin Architecture?

Imagine that our software is a soccer team. In a team, each player has a specific position and function, but we

can make changes as needed by the game. A forward can come out and another can come in to refresh the attack. Well, something similar happens in plugin architecture: we treat parts of our system as components or modules that can be "plugged in" and "unplugged" easily, without altering the overall functioning of the system.

Flexibility and Adaptability

This approach is wonderful when the project needs to quickly adapt to new conditions or requirements. Suppose we want to add a new analytical function or modify the payment process. Instead of rewriting large portions of code, we simply "plug in" the new module that handles this function. If later we need to update or replace it, we can do it without affecting the rest of the application.

Practical Implementation of Plugins

To implement this architecture, each module or plugin must be clearly defined with standardized interfaces. This means that each part knows how to communicate with the others without needing to know exactly what is inside each one. It's like saying, "Hey, I need you to do this, I don't care how you do it, just do it." This separation helps keep our code tidy, testable, and, above all, easy to change.

Advantages of Using Plugins

One of the biggest advantages of using plugin architecture is the ability to test new features or services in isolation. We can develop, test, and deploy new components independently, which speeds up development cycles and reduces the risk associated with changes in complex systems.

A Real Example

Let's think of an application that manages reservations at a hotel. If we want to add functionality to manage special events, we can develop a specific plugin for that. This module will handle everything related to events, from booking to guest management, and will integrate with the existing system through a defined interface.

Chapter conclusion

Plugin architecture is a brilliant example of how Clean Architecture promotes adaptability and scalability. It allows us to maintain complex and constantly evolving systems without losing our minds in the process. In the next chapter, we will talk about the disadvantages and temporal considerations of Clean Architecture. So, don't miss it because we're going to lay our cards on the table and talk clearly about when and how to use this architecture to our advantage!

See you in the next chapter, let's keep moving forward, team!

Chapter 6: "Disadvantages and Temporal Considerations of Clean Architecture"

Like everything in life, even the best practices have their disadvantages and limitations. In this chapter, let's be brutal and honest about the downsides of Clean Architecture and the temporal considerations involved in its implementation. So, sharpen your pencils, because we're going to jot down the pros and cons of this powerful tool.

The Disadvantages of Rigorous Structuring

While the clear and defined separation of responsibilities in Clean Architecture offers many benefits, it can also be its Achilles' heel. This structure requires rigorous planning and discipline, which can result in increased

initial development time. Yes, mates, not everything that glitters is gold. This rigor can be seen as a disadvantage, especially in projects with tight deadlines or where delivery speed is crucial.

Verbosity and Complexity

Another possible downside is verbosity. Clean Architecture can lead to the creation of many files and directories, which can overcomplicate the project more than necessary, especially for those who are new to this approach. Sometimes, one can end up feeling like they're writing more 'scaffolding' code than code that actually contributes to the business.

Temporal Considerations: When to Use Clean Architecture?

Not every project needs a bazooka to kill a fly. Clean Architecture is ideal for large and complex applications that will require long-term maintenance and where scalability is a critical factor. However, if you're working on a rapid prototype or a small project with a short lifecycle, implementing Clean Architecture might be overkill. In these cases, simplicity should be the priority.

The Cost of Time

Time is money, and this has never been more true than in software development. Adopting Clean Architecture means investing time in learning and applying its principles correctly. This can be a challenge for teams under pressure to deliver quickly. Additionally, any major architectural change, such as moving from a monolithic architecture to one based on Clean, will require significant effort and possibly a steep learning curve.

Chapter Conclusion

So, is it worth it? Absolutely, but only if the shoe fits. Clean Architecture is not a one-size-fits-all solution, but when used in the right context, it can transform a chaotic project into a well-oiled system that is a pleasure to maintain and scale.

That wraps up our book, my dear developers. I hope you've enjoyed this journey as much as I have and that you're ready to apply this knowledge to your own projects. Remember, the best architecture is one that serves the needs of the project and the team. Keep coding, keep learning, and above all, keep enjoying the process!

See you in the next code, Gentleman out!

🕶 Chapter 7: "The Structure of Clean Architecture: A Layered View"

In this additional chapter, we're going to delve deeper into the structure of Clean Architecture and how it breaks down into distinct layers. This architecture, developed by Robert C. Martin, also known as Uncle Bob, is based on design principles aimed at promoting separation of concerns and independence from infrastructure. Let's break down each layer so you understand how each contributes to creating a robust, maintainable, and scalable software system.

1. Entities Layer: The entities layer represents the core of the application, where the business logic and rules fundamental to the system's operation are defined. These entities are agnostic to the rest of the system and must be independent of any framework or database used. They include domain objects whose responsibility is to contain data and execute business rules critical and central to the application.

2. Use Cases Layer: The use cases layer encapsulates and implements all the actions a user can perform with the system. This is where the specific business processes of the application are modeled. Use cases orchestrate the flow of data to and from the entities and direct that data towards the outer layers. This layer should know nothing about how data is presented to the end user or how data is stored; its sole purpose is to handle the application logic required to execute each use case.

3. Interface Adapters Layer: The interface adapters layer, also known as "Interface Adapters," acts as a bridge between internal use cases and the external world. This layer converts data from the most convenient form for use cases and entities to the most convenient form for some external agent like a database or a web browser. For example, an MVC controller in a web application would be located here, taking user data, converting it to a format that use cases can understand, and then passing this data to an appropriate use case.

4. Frameworks & Drivers Layer: The outer layer, or "Frameworks & Drivers," is where all the details about implementation with specific frameworks or databases are handled. This layer is completely independent of the business and takes care of things like the database, the user interface, and any framework used. The goal of this layer is to minimize the amount of code you have to change if you decide to implement another database, another framework, or even a different user interface.

Chapter Conclusion: Understanding the layered structure of Clean Architecture is crucial for designing an application that is easy to maintain, test, and expand. By keeping these layers well-defined and separated, developers can ensure that changes in one area of the software (such as the user interface or the database) have the least possible impact on the other areas, especially on the core business logic.

With this structure, we are better equipped to face the challenges of modern software development, providing systems that not only meet current needs but are also capable of adapting to future changes with ease.

Chapter 8: "Use Cases and Domain in Clean Architecture: The Difference Between Business Logic and Application"

In this fresh chapter, we're diving into the distinctions between use cases and domain within Clean Architecture. It's crucial to grasp these concepts to craft robust, well-structured applications. So, gear up your minds, because we're kicking off with a bang!

What is Domain and Why is it Central?

The domain in Clean Architecture is the core, the heart of our application. Why? Because it houses all the business logic that defines how our project functions. Everything else in our system may change, but the domain is sacred, it's the essence that cannot be altered by external layers. It's untouchable, and all other layers depend on it.

The Domain Rule

There's a golden rule in Clean Architecture known as "the domain rule," which dictates that the domain is autonomous and should not depend on anything beyond itself. Everything in our system revolves around the domain; it doesn't conform to the other layers, but rather, the other layers must adapt and serve the domain.

Use Cases: Bridging the Gap between External and Domain

While the domain focuses on business rules, use cases are responsible for implementing that logic in specific application scenarios. Use cases know what happens in the domain, but not vice versa. They act as intermediaries between the external world and the pure business rules contained in the domain.

How is Information Communicated?

In Clean Architecture, information flows from the core (the domain) outward. External layers, such as adapters or the user interface, can interact with use cases, but always respecting the domain's autonomy. This structure ensures that dependencies are well-organized and that the core business logic remains protected and clear.

Common Mistake in Responsibility Division

A critical point that often confuses even experienced developers is how to correctly divide responsibilities between the domain and use cases. A classic example is business rule validation: these should reside in the domain if they are fundamental to the essence of the application, regardless of the technology or platform used.

Practical Example: Banking Application

Let's consider a banking application where it's established that to open an account, the client must be over 18 years old. This is a business rule and, therefore, belongs to the domain because it defines a fundamental requirement of the application, uninfluenced by external technical considerations.

Chapter Conclusion

Mastering the distinction between the domain and use cases is essential for any developer aiming to build robust and maintainable systems using Clean Architecture. By maintaining this clear separation, we ensure that our application is not only functional but also adaptable in the long run.

In the next chapter, we'll delve even deeper into how to implement these theories in practical examples and code, ensuring that these theoretical structures translate into real and efficient applications. Don't miss out, it's only getting better!

Chapter 9: "Practical Implementation of Use Cases and Domain in Clean Architecture"

It's time to roll up our sleeves and put all that theory into practice! Let's see how to effectively implement use cases and the domain in our projects. Get ready to dive into the code!

Basic Structure of a Use Case

Use cases define how a specific operation is executed in the system, following the rules of the domain. For example, in a banking application, a use case could be "Create Account," implementing the age validation we mentioned earlier. Let's see how this could look in code:

```
class CreateAccountUseCase {
    constructor(private accountRepository: AccountRepository) {}

    execute(accountData: AccountData) {
        if (accountData.age < 18) {
```

```
            throw new Error("The client must be over 18 years old to open an account.");
        }
        const account = new Account(accountData);
        this.accountRepository.save(account);
    }
}
```

Integration of the Domain

The domain handles entities and core business rules. In our example, the entity would be the `Account`, and a business rule would be that the client must be of legal age. This could be implemented in the `Account` class like this:

```
class Account {
    constructor(private data: AccountData) {
        if (data.age < 18) {
            throw new Error("The client must be over 18 years old to open an account.");
        }
    }
}
```

Differentiation Between Layers

It's vital to clearly differentiate between the domain and use case layers. The domain contains the fundamental logic for the business and does not depend on the application or infrastructure layer. Use cases interact with these domain entities and orchestrate how operations are executed in response to user actions.

Handling the User Interface

The user interface must be completely agnostic regarding the domain and use cases. Its only role is to present data to the user and send user actions to the application layer. For example, a button in a graphical interface that allows the user to open a new account would simply invoke the `CreateAccountUseCase`.

Chapter Conclusion

The key to properly implementing Clean Architecture is to ensure that each part of the system does only what it's supposed to, and nothing more. This not only simplifies system maintenance and expansion but also facilitates testing and debugging of each component in isolation.

In the next chapter, we'll explore how business rules and use cases influence the choice of technologies and tools for developing our application. So stay tuned, as we continue to explore how to make our applications cleaner, more efficient, and, above all, magnificent!

Chapter 10: "Technologies and Tools: Alignment with Use Cases and Domain in Clean Architecture"

Now that we have a solid understanding of use cases and the domain, it's time to discuss how to choose the technologies and tools that best align with our architecture. Buckle up, because we're diving right in.

The Role of Technologies in Clean Architecture

In Clean Architecture, the choice of technologies must be careful and strategic. The key here is for technology to serve the architecture, not the other way around. This means that technological decisions should support and not complicate our domain and use case structure.

Technological Independence of the Domain

The domain must be completely agnostic regarding technology. This is crucial because business rules should not be contaminated by technological limitations or specifics. For example, if we decide to switch our database from SQL to NoSQL, this should not affect the domain logic of our application.

Selecting Technologies for Use Cases

Use cases, although closer to technology than the domain, should still remain flexible enough to adapt to changes. When selecting tools to implement use cases, let's consider those that offer the greatest flexibility and ease of integration. For example, in a backend, we might opt for frameworks like Express.js or Spring Boot, which are robust but offer the necessary flexibility to adapt to different use cases without imposing severe restrictions.

Practical Example: Implementing the User Interface

Consider the user interface in a system that uses Clean Architecture. While the interface can be built with any frontend framework like React, Angular, or Vue, it's essential that these technologies are used in a way that respects the separation of domain and use cases. This is achieved through adapters or bridges that communicate the UI with the use cases, maintaining domain independence.

```
// Example adapter in React for the CreateAccount use case
function CreateAccountComponent({ createAccountUseCase }) {
    const [age, setAge] = useState('');

    const handleCreateAccount = async () => {
        try {
            await createAccountUseCase.execute({ age });
            alert('Account created successfully!');
        } catch (error) {
            alert(error.message);
        }
    };

    return (
        <div>
            <input
                type="number"
                value={age}
                onChange={e => setAge(e.target.value)}
                placeholder="Enter your age"
            />
            <button onClick={handleCreateAccount}>Create Account</button>
        </div>
    );
}
```

Chapter Conclusion

Choosing technologies in Clean Architecture is not just a matter of personal preference or market trend, but a strategic decision that must align with the structure and principles of our architecture. Let's ensure that our tools and technologies strengthen our use cases and domain, not complicate them.

In the next chapter, we'll address the management and maintenance of these systems in the long term, considering how Clean Architecture facilitates these processes. So stay tuned, because the learning continues, and the best is yet to come!

Chapter 11: "Long-Term Maintenance and Management in Clean Architecture"

Hello again, fellow code enthusiasts! Gentleman here, and today we're delving into one of the most crucial topics in software development: maintenance and long-term management of our applications built with Clean Architecture. Let's get ready to understand how this methodology not only helps build robust systems but also efficiently maintains them over time.

Why Maintenance is Important in Clean Architecture?

Clean Architecture greatly facilitates software maintenance by design. Thanks to the clear separation between the domain and use cases, and between these and the external layers, each part of the system can be updated, improved, or even replaced without causing a domino effect on other parts. This independence is vital for the long-term sustainability of any application.

Strategies for Effective Maintenance

- **Automated Testing**: Implementing a robust set of automated tests is fundamental. These tests should cover both use cases and domain entities. By keeping the domain isolated from external and technological changes, we can ensure that tests remain stable and reliable over time.
- **Clear Documentation**: Maintaining detailed and up-to-date documentation of each layer, especially the domain and use cases, helps new developers and teams quickly understand the logic and structure of the system, facilitating maintenance and scalability.
- **Continuous Refactoring**: Refactoring is not just for fixing bugs but for continuously improving the code structure as the software evolves and grows. Clean Architecture makes refactoring more manageable and less risky since dependencies are controlled and clear.

Dependency Management and Updates

One of the great benefits of Clean Architecture is how it handles dependencies. By having well-defined and separated layers, updating a library or changing a tool in a specific layer (such as the infrastructure layer or the user interface) does not impact the domain or use cases. This significantly reduces the risk during updates and minimizes system downtime.

Practical Example: Updating the Persistence Layer

Let's say we decide to switch our database from MySQL to PostgreSQL. In a clean architecture, this transition would mainly affect the infrastructure layer, where the database adapters reside. The domain and use cases,

containing the business logic, would remain intact, ensuring that the core functionality of the application is not compromised.

```
// Database adapter before the update
class MySQLAccountRepository implements AccountRepository {
    save(account: Account) {
        // Logic for saving the account in MySQL
    }
}

// Database adapter after the update
class PostgreSQLAccountRepository implements AccountRepository {
    save(account: Account) {
        // Logic for saving the account in PostgreSQL
    }
}
```

Chapter Conclusion

Maintainability and effective long-term management of applications are perhaps the greatest benefits offered by Clean Architecture. By adhering to its principles, we can ensure that our systems are not only robust at launch but also continue to be so as they scale and evolve.

In our next chapter, we'll discuss how to lead teams and projects using Clean Architecture, ensuring that the design integrity is maintained from conception to final implementation. Stay tuned, as there's still much more to learn!

Chapter 12: "Differences Between Application Logic, Domain Logic, and Business Logic in Clean Architecture"

In this chapter, we'll delve into the key differences between application logic, domain logic, and business logic, three fundamental concepts in software architecture that can easily be confused but are critical for designing and maintaining robust and efficient systems.

Domain Logic: The Core of the Application

Domain logic is the central core of any system in Clean Architecture. It contains the essential business rules and processes that define how the enterprise operates at the most fundamental level. This logic is independent of the platform and technology used to implement the application. Its main characteristic is that it is purely about the "what" and "why" of operations, not the "how".

For example, in a banking system, the rule that "customers must be over 18 years old to open an account" is part of the domain logic.

Application Logic: Orchestrating the Workflow

Application logic refers to the "how" operations defined in domain logic are carried out within a specific application. It acts as the mediator between the user interface and domain logic, managing the flow of data and ensuring that operations are executed in the correct order. This layer also handles the logic necessary to interact with the database and other external services.

Continuing with the example of the banking system, application logic would decide when and how the customer's age is verified and what to do if the customer does not meet this rule.

Business Logic: Organizational-Level Rules

Business logic encompasses rules and policies that are not limited to a specific application but are common to several applications or the entire organization. These rules may include compliance policies, security regulations, and other operational standards that affect multiple systems within the enterprise.

For example, in our analogy of the banking system, a business rule could be that all financial transactions must be audited annually, which is a policy that would affect all financial management applications within the organization.

Implementation of Logic Layers in Clean Architecture

In practice, separating these three logics allows developers and system designers to ensure that each part of the software deals with specific and well-defined aspects of the business. This not only clarifies development and maintenance but also facilitates scalability and adaptability of the system in the long term.

```
// Example of how these logics might interact in code

// Domain Logic
class Account {
    constructor(private age: number) {
        if (!this.isOfLegalAge()) {
            throw new Error("Must be over 18 years old to open an account.");
        }
    }

    private isOfLegalAge() {
        return this.age >= 18;
    }
}

// Application Logic
class CreateAccountService {
    constructor(private accountRepo: AccountRepository) {}

    createAccount(customerData: { age: number }) {
        const account = new Account(customerData.age);
        this.accountRepo.save(account);
    }
}
```

In summary, clear differentiation between application logic, domain logic, and business logic is essential for building computer systems that are efficient both in execution and maintenance. By understanding and properly applying these separations, teams can develop software that not only meets current requirements but is also robust against future changes.

🕶 Chapter 13: "Adapters: Breaking Schemes in Clean Architecture"

In this chapter, we dive deep into one of the most revolutionary and dynamic layers of Clean Architecture: adapters. We'll break down how this layer, which may seem complicated due to its dual nature, is actually the vital bridge between the inner core of our application and the chaotic external world. Join me on this journey to better understand its disruptive role!

Adapters: A Vital Dual Function Adapters in Clean Architecture have a complex and critical task: they communicate the application with the external world and vice versa. They are comparable to cell membranes in biology, which control what substances can enter and exit the cell, ensuring its survival and optimal functioning.

- **Data Input**: When information arrives from the outside, such as data from a frontend endpoint, the adapter transforms it to fit the needs of internal use cases. For example, it may receive information in a non-ideal format and must map it to a format that the business logic can efficiently process.
- **Data Output**: Similarly, when use cases generate data that needs to be sent outside, the adapter also handles this transformation, ensuring that the data is understandable and usable for other systems or end users.

```
// Example of adapter in action
class UserInfoAdapter {
    convertToInternalFormat(externalData: { name: string; lastName: string }) {
        // Transform external data to internal format
        return {
            firstName: externalData.name,
            lastName: externalData.lastName
        };
    }

    convertToExternalFormat(internalData: { firstName: string; lastName: string }) {
        // Prepare internal data to send externally
        return {
            name: internalData.firstName,
            lastName: internalData.lastName
        };
    }
}
```

Why Adapters Are Key These components not only facilitate interaction between different systems and formats but also protect the integrity of the internal logic. By isolating changes and peculiarities from the external world in a specific layer, adapters allow the core of the application to remain clean and focused on business logic, without being affected by external variables.

Conclusion of the Chapter Understanding and effectively implementing adapters in Clean Architecture is crucial for developing robust and maintainable systems that interact with a complex and constantly changing environment. In the next chapter, we'll explore real examples of how adapters have allowed applications to

quickly adapt to new demands and technologies without major disruptions, keeping the business logic intact and running smoothly. Follow along with me on this adventure to see how this theory is applied in practice!

Chapter 14: "The Outer Layer in Clean Architecture: Connecting Your Application with the External World"

In this chapter, we'll dissect the outer layer in Clean Architecture, a crucial layer that connects us with the external world. Often underestimated, today we'll give it the spotlight it deserves!

Importance of the Outer Layer The outer layer in Clean Architecture includes everything that interacts with the external environment of our application, such as user interfaces, databases, and external APIs. It's essential to understand that, although it's the layer closest to the user or external service, its design and operation must be carefully managed to maintain the integrity of our internal architecture.

Components of the Outer Layer

- **Frontend**: Whether you're working with React, Vue, Angular, or any other framework, the frontend is considered part of the outer layer. Its main task is to present information to the user and capture their interactions, which are then communicated to the internal layers through adapters.
- **Backend**: Services that process business logic and handle client requests also reside here, acting as the bridge between the frontend and the application's domain.
- **Database**: While many might think that databases are the core of an application, in Clean Architecture, they are simply another implementation detail, managed through adapters to maintain domain independence.

Implementation Details While we spend days and nights learning and discussing which technology to use, at the end of the day, these are all just implementation details. What really matters is how these details integrate and contribute to the core business rules without directly affecting them.

```
// Example of an adapter for a MongoDB database
class MongoDBUserAdapter implements UserDatabaseAdapter {
    constructor(private db: DatabaseConnection) {}

    async fetchUser(userId: string) {
        const userDocument = await this.db.collection('users').findOne({ id: userId });
        return new User(userDocument.name, userDocument.email);
    }

    async saveUser(user: User) {
        await this.db.collection('users').insertOne({
            id: user.id,
            name: user.name,
            email: user.email
        });
```

```
    }
```

The Power of Adapters The real magic happens with adapters, which allow elements of the outer layer like databases and APIs to change without having a devastating impact on the business logic. Switching from MongoDB to DynamoDB, for example, would simply require adjustments to the corresponding adapter, not a complete rewrite of business logic or use cases.

Conclusion of the Chapter Understanding and correctly applying the outer layer in Clean Architecture allows us to make the most of the technologies we choose without compromising the integrity of our application. Through adapters, we can ensure that our application is as flexible as we need it to be, without being tied to any specific technology.

In the next chapter, we'll explore how these integrations affect the performance and security of our applications, ensuring that they are not only functional but also robust and secure. So, don't miss out on this next journey, as we continue enriching our knowledge together!

Chapter 15: "Performance and Security in the Outer Layer: Optimizing Interaction with the External World"

Now that we've explored the structure and function of the outer layer in Clean Architecture, it's crucial to address how these interactions affect the performance and security of our applications. This chapter focuses on optimizing these critical aspects to ensure that not only our architecture is solid but also fast and secure.

Performance Optimization in the Outer Layer Performance is key when it comes to user experience and operational efficiency. In the outer layer, where our application meets the world, every millisecond counts. This is where adapters play a crucial role, not only in data transformation but also in ensuring that these transformations are efficient.

- **Caching**: Implementing caching strategies in adapters can significantly reduce latency and load on our servers. For example, caching common query results from databases can prevent expensive operations from being repeated.
- **Asynchrony**: Using asynchronous operations to handle requests can greatly improve performance by not blocking processing while waiting for responses from the server or database.

Ensuring Security in the Outer Layer Security is another vital aspect, especially when our application interacts with the external world. The outer layer is often the first point of contact for attacks, so it's imperative to strengthen it.

- **Data Validation and Sanitization**: It's crucial to validate and sanitize all data entering through the outer layer to protect our application from malicious inputs. Adapters should ensure that all incoming data meets expectations before passing it to the internal layers.
- **Error Handling**: Implementing robust error handling in adapters can prevent the propagation of errors that could expose vulnerabilities or sensitive information.

Practical Example: Improving Performance and Security Consider an adapter that interacts with an external API. This adapter not only needs to map the data for internal use cases but also optimize access and ensure that the data is secure.

```
class ExternalAPIAdapter {
    constructor(private apiClient: APIClient) {}

    async fetchSecureData(userId: string) {
        const userData = await this.apiClient.get(`/users/${userId}`);
        if (!userData) {
            throw new Error('User not found');
        }
        return this.sanitize(userData);
    }

    private sanitize(data: any) {
        // Remove any unwanted or dangerous fields
        delete data.internalId;
        return data;
    }
}
```

Conclusion of the Chapter Optimizing performance and ensuring security in the outer layer are essential aspects that require careful and continuous attention. By applying the right strategies and using adapters effectively, we can ensure that our application not only runs smoothly but is also secure against external threats.

Book Conclusions: "Code Gentleman: Mastering Clean Architecture"

As we reach the end of this journey through Clean Architecture, we've explored each layer, dissected every component, and discovered how each part contributes to the success of a modern and scalable application. But what have we learned, and how can we apply this knowledge to our projects? Here are the key takeaways for you to take away and start applying from today.

1. Clarity in Separation of Concerns: Clean Architecture teaches us the importance of maintaining a clear separation between the different concerns of our applications. This not only facilitates maintainability and scalability but also allows teams from different disciplines to collaborate effectively without stepping on each other's toes.

2. Independence of the Domain: The heart of our application, the domain, must remain pure and free from external influences. This independence ensures that business rules remain consistent and protected, regardless of changes in peripheral technologies or user interfaces.

3. Adapters and External Layers: Adapters and the external layer act as the guardians of our system, protecting the business logic from impurities and variations from the external world. Learning to implement

these components correctly is essential for building systems that can evolve without constant internal refactoring.

4. Flexibility and Adaptability: One of the great lessons of Clean Architecture is its focus on flexibility. By isolating dependencies and allowing implementation details to reside on the periphery, our applications can quickly adapt to new technologies or business requirements without major revisions to the core of the system.

5. Maintenance and Evolution: Finally, we've seen how Clean Architecture facilitates the maintenance and evolution of applications over time. Automated testing, clear documentation, and continuous refactoring strategies are just some of the practices that help systems built with this architecture endure and adapt without crumbling under the weight of their own success.

Overall Conclusion: As leaders and developers, our goal is to build software that not only meets current requirements but also adapts and responds to future challenges. Clean Architecture offers a robust and proven framework for achieving this. By understanding and applying its principles, we position ourselves to lead projects that not only solve technical problems but also drive business success and innovation.

This book does not mark the end of your learning, but rather, I hope, the beginning of a new way of thinking and building software. With these tools in your arsenal, you are more than equipped to face the challenges of modern development. Go ahead, Gentleman, and transform the world of software with every line of code you write!

Applying Clean Architecture in the Front End

Hey folks! Today we are going to talk about how to bring Clean Architecture to the front end, but with a special twist. We will see how we can do it in a more organic and flexible way, without overly rigid folder structures, and how the scope rule plays a fundamental role in all of this. Join me on this journey through a more natural and adaptable approach!

Key Concepts of Clean Architecture

The idea behind Clean Architecture is to decouple software from UI technologies, databases, and any other external elements, focusing on pure business logic. This is achieved through layers that clearly separate responsibilities:

- **Domain:** Here we define our entities and business rules that are completely independent of the user interface and external technologies. These are the fundamental concepts on which the application is built.
- **Use Cases:** Responsible for implementing the business logic necessary to meet the functional requirements of the system. They operate on the domain model and use adapters to communicate with the infrastructure layer.
- **Adapters:** Connect use cases with the outside world, either by presenting data to the user or communicating with a database.
- **Frameworks and Drivers:** This is the outermost layer, where we interact directly with specific frameworks and libraries.

Example

For the example of a banking application where the business requirement is that only people over 18 years old can register, we will design a structure following the principles of Clean Architecture. This structure will ensure that business rules, such as the age restriction, are clearly defined and decoupled from the user interface and other external components.

Proposed Structure for the Banking Application

Let's imagine how we could organize our project into the layers suggested by Clean Architecture:

Domain, Entities Models and Business Rules

Here we define the `User` model, which includes attributes such as name, date of birth, address, etc. Additionally, this layer will contain business rules, such as verifying the age of majority.

- User.js

```
class User {
  constructor(name, dateOfBirth) {
    this.name = name;
    this.dateOfBirth = dateOfBirth;
  }

  isAdult() {
    const today = new Date();
    const age = today.getFullYear() - this.dateOfBirth.getFullYear();
    return age >= 18;
  }
}
```

- UserBusinessRules.js

```
function validateUser(user) {
  return user.isAdult();
}
```

Use Cases

This layer contains the specific logic that implements the functional requirements, using the domain entities. In our case, a use case would be "Register User".

- RegisterUser.js

```
class RegisterUser {
  constructor(userRepository, user) {
    this.userRepository = userRepository;
    this.user = user;
  }

  execute() {
    if (validateUser(this.user)) {
      this.userRepository.add(this.user);
      return true;
    } else {
      throw new Error("User must be at least 18 years old");
    }
  }
}
```

Interface Adapters

These adapters will include controllers and presenters that interact with the domain layer and transform data for the UI or external services.

- UserAdapter.js

```
class UserAdapter {
  static toDTO(user) {
    return {
      name: user.name,
      dateOfBirth: user.dateOfBirth.toISOString().split("T")[0],
    };
  }
}
```

Frameworks and Drivers

Here is where specific technology details such as React components for the UI, route configuration, and services that interact with databases or external APIs are implemented.

- UserComponent.js (React Component)

```
import React, { useState } from "react";

function UserComponent({ onSubmit }) {
  const [name, setName] = useState("");
  const [dateOfBirth, setDateOfBirth] = useState("");

  const handleSubmit = () => {
    const user = new User(name, new Date(dateOfBirth));
    onSubmit(user);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <input
        type="date"
        value={dateOfBirth}
        onChange={(e) => setDateOfBirth(e.target.value)}
      />
      <button type="submit">Register</button>
    </form>
```

```
  );
}
```

- App.js (Setting up the application)

```
import React from "react";
import UserComponent from "./UserComponent";
import UserRepository from "./UserRepository";

function App() {
  const userRepository = new UserRepository();

  const handleUserSubmit = (user) => {
    try {
      const registerUser = new RegisterUser(userRepository, user);
      registerUser.execute();
      alert("User registered successfully!");
    } catch (error) {
      alert(error.message);
    }
  };

  return <UserComponent onSubmit={handleUserSubmit} />;
}
```

Organic Approach to Folder Structure

Unlike traditional approaches that structure folders very rigidly from the start, I prefer a more organic approach. The idea is to allow the project structure to evolve naturally as requirements grow and change. I don't like over-structuring folders because I believe the structure should emerge from the needs of the project and the team, not the other way around.

Introduction to the Scope Rule

The scope rule is essential in this organic approach. It defines how we organize and reuse components based on their visibility and use within the application:

- **Components in Root:** These are components and services that are accessible and reusable throughout the entire application. For example, generic UI components or authentication services that are needed in multiple parts of the system.
- **Components in Specific Functionalities:** Located in specific containers or modules, these components are only used within a particular context or functionality. They are perfect for applying lazy loading, as they are only loaded when the corresponding functionality is needed.

Applying Clean Architecture with the Scope Rule

Here's how we can apply these principles together to build a robust and maintainable front-end application:

- **Clearly Define the Domain Model:** We start by defining our domain model in an agnostic way, without worrying about the UI or infrastructure.
- **Develop Use Cases:** We implement the business logic in the form of use cases, which manipulate the domain model and communicate with adapters.
- **Implement Adapters Flexibly:** We use adapters to connect our use cases with specific user interface components and external services. This is where we apply the scope rule to decide whether a component is global or module-specific.
- **Use Lazy Loading to Improve Performance:** We lazily load specific functionality modules or containers as needed, which is easily managed through routing in modern frameworks like React, Angular, or Vue.

Let's expand on how each module or functionality in the front end can be organized to clearly reflect its purpose and internal structure, following the

philosophy of Clean Architecture and the scope rule we discussed.

Modular Structure by Functionality

In a front-end development approach based on Clean Architecture, each feature or functionality of the application is organized into its own folder, named exactly after the feature it represents. This structure not only facilitates code navigation and understanding but also effectively encapsulates functionality.

Container Component

Within each functionality folder, a main component is created that carries the same name as the folder. This component acts as a "container" and has two main responsibilities:

- **Presentation Structure:** Defines how components are structured and displayed on the screen. This container determines the layout and visual composition of child components that form the specific functionality's interface.
- **Business Logic and Data Retrieval:** Integrates the business logic relevant to the layout, managing the necessary state, and performing operations to obtain data from the domain entities. This includes interacting with services to retrieve or send data to external sources.

Specific Functionality Components

Each component within the functionality folder handles its specific functionality. These components are designed to be as autonomous as possible, interacting with the domain entities and applying the relevant business rules. Their modular and well-defined design facilitates reuse and maintenance.

Services and Adapters

Services are used by components to communicate with external entities, such as backends or APIs. These services generally reside in their own subfolder within the functionality module and are responsible for sending and receiving data to and from the outside.

Adapters, on the other hand, are also found in a folder called `adapters` within the module. Their function is to map data between the format expected by external services and the format used by domain entities. This bidirectional mapping ensures that data can be exchanged smoothly and coherently, respecting the abstractions imposed by the architecture.

Implementation Conclusion

Organizing each functionality into its own module, with a container component managing presentation and associated logic, along with specific components handling more granular details, creates a highly modular and scalable system. This structure not only improves code readability and maintainability but also optimizes performance through techniques like lazy loading, loading only the necessary modules when required by the user.

Implementing Clean Architecture in the front end with this detailed and organized approach prepares the ground for robust, maintainable, and adaptable applications, capable of evolving and expanding with business and market needs.

What is the Container Pattern?

The container pattern, also known in some circles as the "Container Pattern," is a software architecture technique that involves encapsulating or grouping several related elements within the same module or container. This container acts as a logical boundary that defines the autonomy and responsibility over a specific portion of the application's functionality.

Container Pattern Structure

Let's imagine we are working on a web application. In a traditional approach, you might have all your components, business logic, and service calls scattered throughout the project. In contrast, with the container pattern, you organize these elements into logical groups that reflect their functions within the application.

For example, if we have a section of the application dedicated to user management, we might have a folder structure like this under a `UserManagement` directory:

```
UserManagement/
├── components/
│   ├── UserProfile.js
│   ├── UserList.js
├── hooks/
│   ├── useUserSearch.js
│   ├── useUserProfile.js
├── models/
```

```
|   ├── UserModel.js
├── services/
|   ├── UserService.js
└── UserContainer.js  // This is the main container
```

How the Container Pattern Works

- **Encapsulation:** Each container handles its own state and dependencies. This means that the user management container handles everything specific to that function, from business logic to interacting with the corresponding API.
- **Independence:** Containers are independent of each other, making development, testing, and maintenance easier. If you need to work on user management, everything you need is in one place, without having to touch other parts of the application.
- **Reusability:** By encapsulating logic and components coherently, you can reuse these containers in different parts of the application or even in different projects, as long as the functionality is required.
- **Integration with Lazy Loading:** When using modern frameworks like React, Angular, or Vue, we can lazily load these containers. This means that the code related to user management is only loaded when the user accesses that specific part of the application, improving performance and initial load speed.

Benefits of the Container Pattern

- **Better Organization:** By having a clear boundary of what code does what, readability is improved, and the project structure is simplified.
- **Decoupling:** Reduces cross-dependencies between different parts of the application, minimizing the risk of unwanted side effects during updates or changes.
- **Scalability:** Facilitates managing the growth of the application. As the application grows, you can keep adding containers without significantly affecting existing ones.
- **Easier Maintenance:** Updating, testing, or fixing bugs becomes easier because everything affecting a functional area is contained within its own module.

Implementing the container pattern is like organizing a set of tools into specific boxes in your workshop; each tool has its place, and you know exactly where to find what you need for each job. This not only makes your life easier but also makes the work more efficient and less prone to errors. In software development, especially in complex applications, adopting this pattern can make the difference between a manageable project and a constant headache.

So go ahead, try it out and see how it transforms your workflow!

Adopting Clean Architecture in our projects not only improves code quality but also prepares us to grow and adapt easily to new demands and technologies. Implementing techniques like lazy loading and the container pattern ensures a robust, maintainable, and efficient application.

Example with Everything Learned

Sure! Let's dive into how we could structure a practical example using Clean Architecture in the front end, applying the concept of modules and containers we discussed. Suppose we are building an e-commerce application that includes functionalities like listing products, adding products to the cart, and making payments.

Example Folder Structure for an E-commerce Application

The following folder structure reflects how we might organize the application's code:

```
src/
└── products/
    ├── ProductsContainer.js     // Container for the product listing
    ├── components/
    │   ├── ProductList.js       // Displays the product list
    │   └── ProductItem.js       // Displays an individual product
    ├── services/
    │   ├── ProductService.js    // Communicates with the API to fetch products
    └── adapters/
        └── ProductAdapter.js    // Adapts product data for the view
└── cart/
    ├── CartContainer.js         // Container for the shopping cart
    ├── components/
    │   ├── CartView.js          // Main view of the cart
    │   └── CartItem.js          // Component for an individual item in the cart
    ├── services/
    │   ├── CartService.js       // Manages the logic for adding/removing items from the cart
    └── adapters/
        └── CartAdapter.js       // Adapts cart data for the view
└── checkout/
    ├── CheckoutContainer.js     // Container for the checkout process
    ├── components/
    │   ├── PaymentForm.js       // Form for payment details
    │   └── OrderSummary.js      // Order summary before purchase
    ├── services/
    │   ├── PaymentService.js    // Processes payments
    └── adapters/
        └── PaymentAdapter.js    // Adapts payment data to be sent to the API
```

Implementation Details

Containers:

- **ProductsContainer.js:** This file would be responsible for loading the products from the service, managing any state related to displaying the products, and passing the necessary data to the components for display.

- **CartContainer.js:** Manages the shopping cart state, including added products, selected quantities, and communication with services to update the cart.
- **CheckoutContainer.js:** Controls the checkout process flow, including collecting payment information and completing the purchase.

Components:

- Each component, such as `ProductList.js`, `CartItem.js`, and `PaymentForm.js`, handles the application logic and use cases, bridging the entities with the business rules.

Services and Adapters:

- **ProductService.js, CartService.js, and PaymentService.js** interact with external APIs to send and receive data.
- Adapters like **ProductAdapter.js and PaymentAdapter.js** transform the data received from the API into a format that the components can use more efficiently and vice versa.

Benefits of this Structure

- **Modularity:** Each application functionality is self-contained with its own set of logic and presentation, facilitating maintainability and scalability.
- **Reusability:** Components within each module can be reused in different parts of the application if needed.
- **Maintenance:** Updating or fixing bugs in a specific part of the application is easier because the affected code is isolated.
- **Performance:** With techniques like lazy loading, only the necessary resources are loaded when they are actually needed, which can significantly improve the application's load time.

Mastering React, the Frameworkless Gem

React without a Full Framework

Dear readers and future front-end masters, in this chapter, we will dive into the fascinating world of React, a library that, although sometimes mistaken for a framework, is actually an essential and flexible tool for building user interfaces.

React, developed by Facebook (now Meta), has earned a privileged place in developers' hearts due to its simplicity and efficiency. But when is it appropriate to use React alone and not opt for a more robust framework like Angular or even Vue?

Ideal Conditions for Using React Alone

- **Small to Medium Scale Projects**: React is incredibly efficient for projects that do not require a large number of integrated backend features or additional complexities that a complete framework might handle better.
- **Teams Experienced with Modern JavaScript**: If your team has solid knowledge of modern JavaScript and does not want to deal with the learning curve of TypeScript (although React also works wonderfully with TS), React offers an excellent and flexible base for building without much predefined structure.
- **Applications Needing High Customization**: Without a framework dictating the structure, React allows for extreme customization and flexibility. This is ideal for applications that need a unique or specific architecture that a framework might not support as easily.
- **Heavy Use of Reusable Components**: React focuses on component composition, which makes it easy to reuse them. If your project benefits from a high degree of component reuse, React might be your best option.
- **No Need for a Framework**: If your project does not require SEO because it is private, opting for Vanilla React can be very beneficial as we can exclusively choose which technologies and tools to use. For example, if your application is private and will not benefit from the advantages of Server Side Rendering, then NextJs might be much more than you actually need.
- **A Reference in the Team with Experience**: Following what we mentioned earlier, the incredible flexibility of React is also a double-edged sword. Faced with a problem, there are MANY solutions, so you need someone with experience to know which option is best according to the context in which the team and the project are.

Integrating React into Your Development Team

Integrating React into a development team requires considering both technical skills and team culture. Here are some tips to ensure successful adoption:

- **Continuous Training**: Ensure your team understands the basics of React and its most common design patterns. As Gentleman Programming, I recommend holding pair programming sessions and code

reviews focused on React best practices.

- **Establish Code Standards**: React is very flexible, but that flexibility can lead to code inconsistencies if clear guidelines are not set. Define style and architecture guides from the start.
- **Leverage the Community**: The React community is vast and always willing to help. Encourage your team to participate in forums, webinars, and conferences to stay updated with the latest trends and best practices.
- **Prioritize Quality Over Speed**: While React can enable rapid development, it is essential not to sacrifice code quality. Implement unit and integration tests from the beginning to ensure applications are robust and maintainable.

Conclusion

Using React without an additional framework is perfectly viable and, in many situations, the best decision you could make. Encourage your team to experiment with this library, adapting it to project needs, always with a critical eye towards code quality and sustainability.

Building Our First Component

Understanding JSX

Before getting hands-on with our components, it is crucial to understand what JSX is. JSX is a syntax extension for JavaScript that allows us to write React elements in a way that looks very much like HTML but with the power of JavaScript. This makes writing our interfaces intuitive and efficient.

For example, if we want to display a simple "Hello, world!" in React, we would do it like this:

```
function Greeting() {
 return <h1>Hello, world!</h1>;
}
```

Although it looks like HTML, JSX is actually converted by Babel (a JavaScript compiler) into React function calls like `React.createElement`. So, the above example is essentially the same as writing:

```
function Greeting() {
 return React.createElement('h1', null, 'Hello, world!');
}
```

Functional Components: Like a Cooking Recipe

A functional component in React is simply a JavaScript function that returns a React element, which can be a simple UI description or can include more logic and other components. It is the most direct and modern way

to define components in React, especially since the introduction of Hooks, which allow using state and other features of React without writing a class.

Let's see the basic structure of a functional component:

```
// Defining a functional component that accepts props
function Welcome(props) {
 // We can access the properties passed to the component through `props`
 return <h1>Welcome, {props.name}</h1>;
}

// Using the component with a prop 'name'
<Welcome name='Gentleman' />;
```

In this example, `Welcome` is a component that receives `props`, an object containing all the properties we pass to the component. We use `{}` inside JSX to insert JavaScript values, in this case, to dynamically display the name we receive.

Stateful vs Stateless Components

Let's revisit the distinction between stateful and stateless components:

- **Stateful Components**: Manage some internal state or changing data. Although we have not yet talked about Hooks, it is important to know that these components can use things like `useState` in the future to manage their internal state.
- **Stateless Components**: Simply accept data through `props` and display something on the screen. They do not maintain any internal state and are typically used to display UI:

  ```
  function Message({ text }) {
   return <p>{text}</p>;
  }
  ```

Using UseState in a Component

In React, managing the state of our components is crucial to controlling their behavior and data in real-time. The `useState` function is an essential tool in this process, similar to how we manage daily decisions in our lives.

What is `useState`?

Imagine `useState` as a safe in your house where you keep a valuable object that can change over time, like money that you decide to spend or save. This safe has a special way of showing you how much money is inside (`get`) and a way to update this amount (`set`).

Structure of `useState` as a Class:

If we think of **`useState`** as if it were a class, it might look like this:

```
class State {
 constructor(initialValue) {
  this.value = initialValue; // The initial value is stored here
 }

 getValue() {
  return this.value; // Method to get the current value
 }

 setValue(newValue) {
  this.value = newValue; // Method to update the value
 }
}
```

In this structure, **`value`** represents the current state, and we have **`getValue()`** and **`setValue(newValue)`** methods to interact with this state.

Using `useState` in a Component

To understand it better, let's compare it to something everyday: adjusting the temperature of an air conditioner in your house.

Suppose you want to maintain a comfortable temperature while at home. You would use a control (like **`useState`**) to adjust this temperature. Here's how you might do it:

```
import React, { useState } from 'react';

function AirConditioner() {
 const [temperature, setTemperature] = useState(24); // 24 degrees is the initial temperature

 const increaseTemperature = () => setTemperature(temperature + 1);
 const decreaseTemperature = () => setTemperature(temperature - 1);

 return (
  <div>
   <h1>Current Temperature: {temperature}°C</h1>
   <button onClick={increaseTemperature}>Increase Temperature</button>
   <button onClick={decreaseTemperature}>Decrease Temperature</button>
  </div>
 );
}
```

In this example, `temperature` is the state we are managing. We start with a temperature of 24 degrees. The `increaseTemperature` and `decreaseTemperature` methods act like the buttons to increase or decrease the temperature on the air conditioner control.

🕶 Functional Components: Like a Recipe

Imagine a functional component in React as following a recipe. Every time you decide to cook something, you follow the steps to prepare your dish. Similarly, a functional component "follows the steps" every time React decides it needs to update what is displayed on the screen.

Preparing Ingredients (Props)

When you cook, you first gather all the ingredients you need. In a functional component, these ingredients are the `props`. `Props` are data or information that you pass to the component to do its job, like ingredients in your recipe that determine how the dish turns out.

```
function Sandwich({ filling, bread }) {
 return (
  <div>
   A {filling} sandwich on {bread} bread.
  </div>
 );
}
```

In this example, `filling` and `bread` are the props, the ingredients you need to make your sandwich.

Executing the Recipe (Component Function)

Every time you make the recipe, you follow the steps to combine the ingredients and cook them. Each execution might vary slightly, for example, you might decide to add more spices or less salt. In a functional component, the "execution" is when React calls the component function to generate the JSX based on the current props.

Each time the props change, it's like deciding to adjust the recipe. React "cooks" the component again, that is, executes the component function to see how it should look now with the new "ingredients".

Presenting the Dish (Rendering)

The final presentation is when you put the cooked dish on the table. In React, this is what you see on the screen after the component is executed. The JSX returned by the component function determines how the component is "presented" in the user interface.

Usage Example

Using the component is like serving your cooked dish to someone to enjoy.

```
<Sandwich filling='ham and cheese' bread='whole grain' />
```

Each time the details of the sandwich change (say, changing `filling` to "chicken and tomato"), React performs the process again to ensure the presentation on the screen matches the given ingredients.

This approach helps you see each component as an individual recipe, where the props are your ingredients and the component function is the guide on how to combine them to get the final result on the screen, always fresh and updated according to the ingredients you provide.

Virtual DOM

Imagine that the DOM (Document Object Model) is a stage where each HTML element is an actor. Every time something changes on your web page (for example, a user interacts with it or data received from a server changes the state of the page), you might have to rearrange the actors on this stage to reflect those changes. However, rearranging these actors (DOM elements) directly and frequently is very costly in terms of performance.

This is where the Virtual DOM comes into play. React maintains a lightweight copy of this stage in memory, a kind of sketch or script of how the stage is organized at any given time. This sketch is what we call the Virtual DOM.

How the Virtual DOM Works

- **State or Props Update**: Every time there is a change in the state or props of your application, React updates this Virtual DOM. No changes are made to the real stage yet, only to the sketch.
- **Comparison with the Real DOM**: React compares this updated Virtual DOM with a previous version of the Virtual DOM (the last time the state or props were updated).
- **Detecting Differences**: This comparison process is known as "reconciliation". React identifies which parts of the Virtual DOM have changed (for example, an actor needs to move from one side of the stage to the other).
- **Efficiently Updating the Real DOM**: Once React knows what changes are necessary, it updates the real DOM in the most efficient way possible. This is like giving specific instructions to the actors on how to relocate on the stage without having to rebuild the entire scene from scratch.

Advantages of the Virtual DOM

- **Efficiency**: By working with the Virtual DOM, React can minimize the number of costly manipulations of the real DOM. It only makes the necessary changes and does so in a way that affects the performance of the page as little as possible.

- **Speed**: Since operations with the Virtual DOM are much faster than direct operations with the real DOM, React can handle changes at high speed without degrading the user experience.
- **Development Simplicity**: As developers, we do not have to worry about how and when to update the DOM. We focus on the application's state, and React takes care of the rest.

Change Detection: Understanding the Flow

In the universe of React, change detection is like radar in a soccer game; it is constantly monitoring and ensuring that everything happening on the field is handled appropriately. Let's break down how this process works, focusing on the concept of triggers and how they influence component rendering.

What is a Trigger?

A trigger in React is any event that initiates the rendering process. This can be as simple as a button click, a change in the component's state, or even a response to an API call that arrives asynchronously.

Imagine you are in a kitchen: every action you take, from turning on the stove to chopping vegetables, can be seen as a trigger. In React, each of these triggers has the potential to update the user interface, depending on how the logic is configured in your components.

Types of Triggers

There are two fundamental types of triggers in React:

- **Initial**: It's like the initial whistle of a game. It occurs when the component is first mounted in the DOM. In technical terms, this refers to when the root of your React application is created, and the initial component is loaded.
- **Re-renders**: These occur after the initial mounting. Each time there is an update in the state or props, React decides if it needs to re-render the component to reflect those changes. It's like making real-time adjustments to your game strategy.

The Rendering Process

Rendering in React is like preparing and presenting a dish. When a render is invoked, React prepares the UI based on the current state and props, and then serves it on the screen. This process repeats every time a trigger activates a change.

The "render" is nothing more than the function that composes your component. Every time this function is executed, React evaluates the returned JSX and updates the DOM accordingly, as long as it detects differences between the current DOM and the render output.

Commit: Updating the DOM

Once React has prepared the new view in memory (via the Virtual DOM), it performs a "commit". This is the process of applying any detected changes to the real DOM. It's like, after preparing the dish in the kitchen, finally bringing it to the table. React compares the new Virtual DOM with the previous one and performs the necessary updates to ensure the real DOM reflects these changes.

This process ensures that only the parts of the DOM that actually need changes are updated, optimizing performance and avoiding unnecessary re-renders.

Recap

Each component in React acts like a small chef in the kitchen of a large restaurant, preparing their part of the dish. React, as the head chef, ensures that each component does its part only when necessary, based on the received triggers. This approach ensures that the kitchen (your application) works efficiently and effectively, responding appropriately to user actions and other events.

Mastering Custom Hooks

Introduction to Custom Hooks

In the React universe, custom hooks are like personalized recipes in the kitchen: they allow us to mix common ingredients in new and exciting ways to create unique and reusable dishes (or components). Throughout this chapter, we will explore why custom hooks are essential for simplifying logic and improving reusability in our applications.

The Technical Talk: Lifecycles and Custom Hooks

To better understand custom hooks, let's think of them as if they were individual chefs in a large kitchen. Each chef follows a specific recipe and performs their task at the right moment, similar to how custom hooks execute logic at precise points in a component's lifecycle.

Suppose you want something to happen automatically in your application, like turning on a light when it gets dark. Here's how a custom hook might handle this "automatic turning on":

```
function useAutoLight() {
 const [isDark, setDark] = useState(false);

 useEffect(() => {
  const handleDarkness = () => {
   const hour = new Date().getHours();
   setDark(hour > 18 || hour < 6);
  };

  window.addEventListener('timeChange', handleDarkness);
  return () => window.removeEventListener('timeChange', handleDarkness);
 }, []); // Executes once on mount and unmount
```

```
  return isDark;
}
```

This hook encapsulates the logic for detecting if it is dark and acting accordingly, similar to an automatic light sensor in your home.

Practical Applications of Custom Hooks

Exploring how custom hooks are implemented in everyday situations, we can think of them as shortcuts on a mobile device, allowing us to perform common tasks more quickly and efficiently.

Custom Hook for Form Management:

Consider the process of keeping a personal diary. You want it to be easy to record your thoughts without distractions. See how this custom hook simplifies managing a "digital diary":

```
function useDiary(initialEntries) {
 const [entries, setEntries] = useState(initialEntries);

 const addEntry = newEntry => {
  setEntries([...entries, newEntry]);
 };

 return {
  entries,
  addEntry,
 };
}
```

This hook acts as a personal assistant for your diary, helping you add new thoughts in an organized and efficient manner.

🕶 Correct Usage of useEffect: Avoiding Common Mistakes

In this chapter, we will delve into the correct use of the `useEffect` hook in React, a fundamental tool for managing side effects in your components. While `useEffect` is very powerful, it is crucial to know when and how to use it to avoid performance issues and keep the code clean and manageable.

Introduction to `useEffect`

`useEffect` is used to manage side effects in your React components. But what is a side effect? Imagine `useEffect` as a timer in your kitchen that goes off when you put a pizza in the oven. No matter what you are doing, when the timer rings, you know the pizza is ready, and you need to take it out of the oven.

Basic Structure of `useEffect`

`useEffect` has a simple yet powerful structure:

```
import React, { useEffect, useState } from 'react';

function App() {
 const [data, setData] = useState(null);

 useEffect(() => {
  console.log('Component mounted or updated.');

  return () => {
   console.log('Component will unmount.');
  };
 }, []);

 return <h1>Hello React!</h1>;
}
```

In this example, `useEffect` executes after the component is first rendered, similar to setting a timer when you put the pizza in the oven. The cleanup function (return) is like taking the pizza out and turning off the timer when you finish.

Avoiding Incorrect Use of `useEffect`

`useEffect` should not be used for everything. Using it incorrectly can lead to performance issues and unnecessary complex logic.

Let's see some common mistakes and how to avoid them.

1. Avoiding Infinite Loops

A common mistake is causing an infinite loop by updating the state within `useEffect` without properly managing dependencies:

```
const [count, setCount] = useState(0);

useEffect(() => {
 setCount(count + 1);
}, [count]);
```

In this example, every time `count` changes, `useEffect` executes again, updating `count` again, causing an infinite loop. It's like every time you take the pizza out of the oven, you put it back in and restart the timer.

Solution: Adjust dependencies correctly and avoid updating the state within the same `useEffect` that depends on that state.

```
const [count, setCount] = useState(0);

useEffect(() => {
 const interval = setInterval(() => {
  setCount(c => c + 1);
 }, 1000);

 return () => clearInterval(interval);
}, []);
```

Here, `useEffect` executes only once when the component mounts, and the state updates every second without causing an infinite loop.

2. Avoid Running Logic on State Changes with `useEffect`

An incorrect use case of `useEffect` is running logic when a variable in the component changes. For this, it is better to run the logic at the moment of the action, such as a click.

Incorrect:

```
const [value, setValue] = useState('');

useEffect(() => {
 console.log('The value has changed:', value);
}, [value]);
```

Instead of using `useEffect` to detect changes in `value`, it is more efficient and clear to run the logic directly in the event handler.

Correct:

```
const handleChange = newValue => {
 setValue(newValue);
 console.log('The value has changed:', newValue);
};
```

This is like, instead of using a timer to take the pizza out of the oven, you simply pay attention to the oven and take the pizza out when the alarm rings.

Correct Cases for `useEffect`

1. API Calls

A correct use of `useEffect` is for API calls, where you need to perform an asynchronous action when the component mounts:

```
useEffect(() => {
 const fetchData = async () => {
  const response = await fetch('https://api.example.com/data');
  const result = await response.json();
  setData(result);
 };

 fetchData();
}, []);
```

Here, `useEffect` acts as your reminder to take the pizza out just when the timer rings, ensuring the data is correctly fetched when the component mounts.

2. Subscriptions and Cleanup

`useEffect` is useful for subscribing to external services and cleaning up those subscriptions when the component unmounts:

```
useEffect(() => {
 const subscription = someService.subscribe(data => {
  setData(data);
 });

 return () => {
  subscription.unsubscribe();
 };
}, []);
```

It's like subscribing to a newsletter and unsubscribing when you are no longer interested.

3. Data Synchronization

Another useful application of `useEffect` is synchronizing data between components or with external services. For example, synchronizing local state with the browser's local storage:

```
useEffect(() => {
 const savedData = localStorage.getItem('data');
 if (savedData) {
  setData(JSON.parse(savedData));
 }
}, []);

useEffect(() => {
 localStorage.setItem('data', JSON.stringify(data));
}, [data]);
```

Here, the first `useEffect` acts as an assistant checking if there is pizza in the fridge when you open the door, and the second ensures that any changes in the ingredients are automatically saved.

Component Communication with children Using the Composition Pattern

In this chapter, we will explore how to handle communication between components in React using the composition pattern and children. These concepts allow for creating flexible and reusable components, facilitating data transfer and state management between parent and child components.

Introduction to Component Composition

The composition pattern in React allows us to build complex components from smaller, more specific components. It's like building a car from various parts: engine, wheels, chassis, etc. Each of these parts has a specific function, but together they form a functional car.

Understanding `children`

We can use `props.children` to dynamically pass content from a parent component to a child component.

Basic Example of `props.children`

```
const Container = ({ children }) => {
 return <div className='container'>{children}</div>;
};

const App = () => {
 return (
  <Container>
   <h1>Hello, World!</h1>
   <p>This is a paragraph inside the container.</p>
  </Container>
 );
};
```

In this example, `Container` is a component that wraps its children, which are passed dynamically from the `App` component.

Composition Pattern

The composition pattern is based on the idea of composing smaller components to create complex user interfaces. Imagine you are building a web page like a lego set. Each block is a component that you can combine to create something larger and functional.

Example of Composition with Multiple `slots`

```
const Layout = ({ header, main, footer }) => {
 return (
  <div className='layout'>
   <header className='layout-header'>{header}</header>
   <main className='layout-main'>{main}</main>
   <footer className='layout-footer'>{footer}</footer>
  </div>
 );
};

const App = () => {
 return (
  <Layout
   header={<h1>Welcome</h1>}
   main={<p>This is the main content.</p>}
   footer={<small>© 2024 My Website</small>}
  />
 );
};
```

In this example, the `Layout` component acts as a container that organizes its children into different sections (`header`, `main`, `footer`). Each section is passed as a specific prop.

Communication Between Parent and Child Components

Communication between components in React is mainly handled through props. Parent components can pass data and functions to child components, allowing them to control the behavior and state of the children from the parent.

Example of Communication Between Components

```
const Button = ({ onClick, children }) => {
 return <button onClick={onClick}>{children}</button>;
};

const App = () => {
 const handleClick = () => {
  alert('Button clicked!');
 };

 return (
  <div>
   <Button onClick={handleClick}>Click Me</Button>
  </div>
 );
};
```

In this example, the `Button` component receives an `onClick` function as a prop from the parent `App` component. When the button is clicked, the `handleClick` function defined in the parent is executed.

Complete Example with Composition Pattern and Slots

To illustrate everything we've learned, let's see a more complete example that uses the composition pattern and effectively manages communication between parent and child components.

Complete Example

```
const Panel = ({ title, content, actions }) => {
 return (
  <div className='panel'>
   <div className='panel-header'>
    <h2>{title}</h2>
   </div>
   <div className='panel-content'>{content}</div>
   <div className='panel-actions'>{actions}</div>
  </div>
 );
};

const App = () => {
 const handleSave = () => {
  alert('Saved!');
 };

 const handleCancel = () => {
  alert('Cancelled');
 };

 return (
  <Panel
   title='Settings'
   content={<p>Here you can configure your preferences.</p>}
   actions={
    <div>
     <button onClick={handleSave}>Save</button>
     <button onClick={handleCancel}>Cancel</button>
    </div>
   }
  />
 );
};
```

In this example, the `Panel` component is composed of three sections (`title`, `content`, `actions`) that are passed from the `App` component. This allows for great flexibility and component reuse.

Practical Cases and Benefits

- **Flexibility**: You can design highly configurable and reusable components.
- **Separation of Concerns**: Keeps the logic of each component separate, making maintenance easier.
- **Reusability**: Well-designed components can be reused in multiple parts of the application.

Comparison to Everyday Life

Imagine you are organizing a party and decide to delegate tasks to different people (components). You have someone in charge of drinks, another for music, and another for food. Each works independently, but in the end, they all coordinate to make the party a success. This is the power of the composition pattern in React.

Communication Between Components: Composition vs Context vs Inheritance

When working with React, one of the most common challenges is sharing information between components that do not have a direct relationship. In this chapter, we will explore three main approaches to solving this problem: Prop Drilling, Context, and Composition. Each approach has its advantages and disadvantages, and it is important to understand when and how to use them to create efficient and maintainable applications.

Basic Component Structure

First, let's visualize a basic component structure to better understand the examples:

```
<Father>
  <Child1 />
</Father>

**Child1:**
<Child1>
  <Child2 />
</Child1>
```

Sharing Information Between Unrelated Components

Prop Drilling

Prop Drilling is the most direct method of passing information from a parent component to a child component, and then to a grandchild. However, it can become problematic as the application grows.

Example of Prop Drilling:

```
const Father = () => {
 const sharedProp = 'Shared Information';
 return <Child1 sharedProp={sharedProp} />;
};

const Child1 = ({ sharedProp }) => {
 return <Child2 sharedProp={sharedProp} />;
};

const Child2 = ({ sharedProp }) => {
 return <div>{sharedProp}</div>;
};
```

Problems with Prop Drilling:

- **High Coupling**: Components are tightly coupled, making them difficult to reuse.
- **Difficult Maintenance**: As the number of components increases, maintaining the code becomes complicated.
- **Complicated Understanding**: It is difficult to follow the prop trajectory through many levels of components.

State Management with Context

The React Context API provides a cleaner and more scalable way to share information between components that do not have a direct relationship. It uses a provider that maintains the shared state and consumers that can access that state.

Example of Context API:

```
const MyContext = React.createContext();

const Father = () => {
 const [sharedState, setSharedState] = React.useState('Shared State');

 return (
  <MyContext.Provider value={sharedState}>
   <Child1 />
  </MyContext.Provider>
 );
};

const Child1 = () => {
 return <Child2 />;
};

const Child2 = () => {
```

```
 const sharedState = React.useContext(MyContext);
 return <div>{sharedState}</div>;
};
```

Advantages of the Context API:

- **Understandable**: It is easy to follow the data flow.
- **Scalable**: It can handle applications of any size.
- **Direct**: Components can access the shared state directly without passing unnecessary props.

Disadvantages of the Context API:

- **Provider Location Dependence**: Components can only access the context if they are within the provider.

```
<MyContext.Provider>
  <Father>
    <Child1 />
  </Father>
</MyContext.Provider>

<FatherOutsideContextProvider>
  <Child2 />  // Cannot access the context state
</FatherOutsideContextProvider>
```

Composition for the Win

Composition in React allows for building more complex components from simpler components. Instead of passing props through many levels, we can use the `children` property to pass elements directly.

Example of Composition:

```
const Father = () => {
 const sharedProp = 'Shared Information';

 return (
  <Child1>
   <Child2 sharedProp={sharedProp} />
  </Child1>
 );
};

const Child1 = ({ children }) => {
```

```
  return <div>{children}</div>;
};

const Child2 = ({ sharedProp }) => {
  return <div>{sharedProp}</div>;
};
```

Advantages of Composition

- **Simple**: Less repetitive code and clearer.
- **Scalable**: Easy to scale as the application grows.
- **No Dependencies**: Does not depend on the provider's location like the Context API.
- **Reusable Code**: Components can be easily reused in different parts of the application.
- **Individual Logic**: Each component handles its own logic independently.

Using Context

When developing applications with React, we often need to share information between components that do not have a direct relationship. The React Context API provides a more robust and scalable way to handle this type of situation.

What is Context?

The React Context API allows sharing values between components without having to pass props explicitly through each level of the component tree. It is particularly useful for themes, language settings, or any other type of data that needs to be accessible in many parts of the application.

Creating a Context

First, we need to create a context. This is done with the `createContext` function from React:

```
import { createContext, useContext, useState } from 'react';

export const GentlemanContext = createContext();
```

The `GentlemanContext` can now be used to provide and consume values in our application.

Context Provider

The context provider is the component that wraps those components that need to access the context. It defines the context value that will be shared.

```
export const GentlemanProvider = ({ children }) => {
 const [gentlemanContextValue, setGentlemanContextValue] = useState('');
 return (
  <GentlemanContext.Provider
   value={{ gentlemanContextValue, setGentlemanContextValue }}
  >
   {children}
  </GentlemanContext.Provider>
 );
};
```

In this example, `GentlemanProvider` uses the `useState` hook to maintain and update the context value. This value can be any type of data, such as a string, number, object, or function.

Consuming the Context

To access the context value in a child component, we use the `useContext` hook:

```
export const useGentlemanContext = () => {
 const context = useContext(GentlemanContext);
 if (context === undefined) {
  throw new Error('GentlemanContext must be used within a GentlemanProvider');
 }
 return context;
};
```

The `useGentlemanContext` hook encapsulates the use of `useContext` and ensures that the context is used correctly within a `GentlemanProvider`.

Complete Example

Let's see a complete example of how to use this context in an application:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { GentlemanProvider, useGentlemanContext } from './GentlemanContext';

const Father = () => {
 const { gentlemanContextValue, setGentlemanContextValue } =
  useGentlemanContext();

 return (
  <div>
   <h1>Father Component</h1>
   <button onClick={() => setGentlemanContextValue('Shared Value')}>
    Set Context Value
   </button>
```

```
    <Child1 />
   </div>
  );
};

const Child1 = () => {
 return (
  <div>
   <h2>Child1 Component</h2>
   <Child2 />
  </div>
 );
};

const Child2 = () => {
 const { gentlemanContextValue } = useGentlemanContext();

 return (
  <div>
   <h3>Child2 Component</h3>
   <p>Context Value: {gentlemanContextValue}</p>
  </div>
 );
};

const App = () => (
 <GentlemanProvider>
  <Father />
 </GentlemanProvider>
);

ReactDOM.render(<App />, document.getElementById('root'));
```

Advantages of the Context API

1. Understandable: The data flow is clear and easy to follow. Components can access the context directly without the need to pass unnecessary props.

2. Scalable: The Context API can handle applications of any size without the maintenance issues associated with Prop Drilling.

3. Centralized: The shared state is centralized, making it easy to manage and update.

Disadvantages of the Context API

1. Provider Location: Components can only access the context if they are within the provider. If the provider is misplaced, some components might not have access to the context.

```
<MyContext.Provider>
  <Father>
    <Child1 />
  </Father>
</MyContext.Provider>

<FatherOutsideContextProvider>
  <Child2 />  // Cannot access the context state
</FatherOutsideContextProvider>
```

👄 Understanding useRef, useMemo, and useCallback

In this chapter, we will demystify the use of some of the most powerful and often misunderstood hooks in React: **`useRef`**, **`useMemo`**, and **`useCallback`**. We will see how and when to use them correctly, and compare them to the well-known **`useState`** to understand their differences and similarities.

`useRef`: The Constant Reference

The **`useRef`** hook allows us to create a mutable reference that can persist throughout the entire lifecycle of the component without causing re-renders.

Practical Example with `useRef`:

Suppose we have a component with a button that, when clicked, automatically activates another button:

```
import { useRef } from 'react';

const Clicker = () => {
 const buttonRef = useRef(null);

 const handleClick = () => {
  buttonRef.current.click();
 };

 return (
  <div>
   <button ref={buttonRef} onClick={() => alert('Button clicked!')}>
    Hidden Button
   </button>
   <button onClick={handleClick}>Click to trigger Hidden Button</button>
  </div>
 );
};

export default Clicker;
```

In this example, `useRef` is used to directly access the hidden button and programmatically simulate a click. `useRef` does not cause component re-renders when its value changes, making it ideal for storing references to DOM elements or any value that needs to persist without causing additional renders.

`useMemo`: Memoization of Calculations

`useMemo` is used to memoize costly calculated values and only recalculate them when one of the dependencies has changed. This is especially useful for improving the performance of components that perform intensive calculations.

Practical Example with `useMemo`:

```
import { useMemo, useState } from 'react';

const ExpensiveCalculationComponent = ({ number }) => {
 const expensiveCalculation = num => {
  console.log('Calculating...');
  return num * 2; // Simulates a costly operation
 };

 const memoizedValue = useMemo(() => expensiveCalculation(number), [number]);

 return (
  <div>
   <h2>Result: {memoizedValue}</h2>
  </div>
 );
};

const ParentComponent = () => {
 const [number, setNumber] = useState(1);

 return (
  <div>
   <ExpensiveCalculationComponent number={number} />
   <button onClick={() => setNumber(number + 1)}>Increment</button>
  </div>
 );
};

export default ParentComponent;
```

In this example, `useMemo` memorizes the result of `expensiveCalculation` and recalculates the value only when `number` changes, avoiding unnecessary executions of a costly function.

`useCallback`: Memoization of Functions

`useCallback` is similar to `useMemo`, but instead of memorizing the result of a function, it memorizes the function itself. This is useful to avoid unnecessary recreations of functions, especially when passed as props to child components that could cause additional renders.

Practical Example with `useCallback`:

```
import { useCallback, useState } from 'react';

const ChildComponent = ({ handleClick }) => {
 console.log('Child rendered');
 return <button onClick={handleClick}>Click me</button>;
};

const ParentComponent = () => {
 const [count, setCount] = useState(0);

 const handleClick = useCallback(() => {
  setCount(prevCount => prevCount + 1);
 }, []);

 return (
  <div>
   <h2>Count: {count}</h2>
   <ChildComponent handleClick={handleClick} />
  </div>
 );
};

export default ParentComponent;
```

In this example, `useCallback` ensures that `handleClick` maintains the same reference across renders, avoiding unnecessary renders of the `ChildComponent`.

Comparison between `useState`, `useRef`, `useMemo`, and `useCallback`

useState:

- **Purpose**: Manage internal state of the component.
- **Re-render**: Causes re-render of the component when state changes.
- **Example**: Counters, toggles.

useRef:

- **Purpose**: Create mutable references that persist throughout the component lifecycle.

- **Re-render**: Does not cause re-render when value changes.
- **Example**: References to DOM elements, storing persistent values.

useMemo:

- **Purpose**: Memoize costly calculated values to avoid unnecessary recalculations.
- **Re-render**: Does not cause re-render, memorizes the result of the calculation.
- **Example**: Calculating derived values from complex states.

useCallback:

- **Purpose**: Memoize functions to avoid unnecessary recreations.
- **Re-render**: Does not cause re-render, memorizes the function reference.
- **Example**: Passing callbacks to child components that depend on memoized functions.

Making API Requests and Handling Asynchronous Logic

In this chapter, we'll explore how to make API requests properly in React, manage asynchronous logic, and cache results in Local Storage to enhance application performance.

Making API Requests

Let's break down how to make API requests using `fetch`, explain its components, and understand why we use asynchronous functions (`async`) instead of making requests directly inside `useEffect`.

The Fetch Function

`fetch` is a native JavaScript function used to make HTTP requests to an API. Its basic usage is as follows:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

In this example, `fetch` sends a request to the specified URL. The response is converted to JSON using the `.json()` method, and then the data (`data`) is processed in the subsequent `.then` block. If an error occurs during the request, it's caught and handled in the `.catch` block.

Components of `fetch`

- **URL**: The address to which the request is made.
- **HTTP Method**: By default, `fetch` uses the GET method. We can specify other methods (POST, PUT, DELETE, etc.) by passing a configuration object as the second argument.
- **Headers**: Additional information sent with the request, such as content type (Content-Type) or authentication tokens.
- **Body**: Data sent with the request, especially in POST or PUT methods.

Example with additional configuration:

```
fetch('https://api.example.com/data', {
 method: 'POST',
 headers: {
  'Content-Type': 'application/json',
 },
 body: JSON.stringify({ key: 'value' }),
})
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error('Error:', error));
```

Asynchronous Functions (`async/await`)

`async/await` functions provide a modern and readable way to work with promises. A basic example of an asynchronous function is:

```
const fetchData = async () => {
 try {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
 } catch (error) {
  console.error('Error:', error);
 }
};
```

In this example, `await` pauses the execution of the `fetchData` function until the `fetch` promise resolves. If the promise is rejected, the error is caught in the `catch` block.

Using `async` in `useEffect`

We cannot directly declare an `async` function in the `useEffect` argument because `useEffect` expects the return to be a cleanup function or `undefined`. `async` functions implicitly return a promise, which is not compatible with `useEffect`. To resolve this, we encapsulate the `async` function inside `useEffect`.

Example of `fetchApi` in `useEffect`:

```
import React, { useState, useEffect } from 'react';

const fetchApi = async (setData, setLoading, setError) => {
 try {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) {
   throw new Error('Network response was not ok');
  }
  const data = await response.json();
  localStorage.setItem('apiData', JSON.stringify(data));
  setData(data);
  setLoading(false);
 } catch (error) {
  setError(error);
  setLoading(false);
 }
};

const ApiComponent = () => {
 const [data, setData] = useState(() => {
  const cachedData = localStorage.getItem('apiData');
  return cachedData ? JSON.parse(cachedData) : null;
 });
 const [loading, setLoading] = useState(!data);
 const [error, setError] = useState(null);

 useEffect(() => {
  const getData = async () => {
   if (!data) {
    await fetchApi(setData, setLoading, setError);
   }
  };
  getData();
 }, [data]);

 const clearCache = () => {
  localStorage.removeItem('apiData');
  setData(null);
  setLoading(true);
  setError(null);
 };

 if (loading) {
  return <div>Loading...</div>;
 }
```

```
 if (error) {
  return (
   <div>
    Error: {error.message}
    <button onClick={clearCache}>Retry</button>
   </div>
  );
 }

 return (
  <div>
   <h1>Data from API:</h1>
   <pre>{JSON.stringify(data, null, 2)}</pre>
   <button onClick={clearCache}>Clear Cache and Retry</button>
  </div>
 );
};

export default ApiComponent;
```

Creating a Custom Hook for Managing Cache

To improve code reusability and organization, we can create a custom hook responsible for storing state in context, utilizing Local Storage cache when available. To better encapsulate the context, we'll create a **`DataProvider.js`** file.

DataProvider.js

```
import React, { createContext, useContext, useState, useEffect } from 'react';

const DataContext = createContext();

const DataProvider = ({ children }) => {
 const [data, setData] = useState(() => {
  const cachedData = localStorage.getItem('apiData');
  return cachedData ? JSON.parse(cachedData) : null;
 });
 const [loading, setLoading] = useState(!data);
 const [error, setError] = useState(null);

 useEffect(() => {
  const getData = async () => {
   if (!data) {
    await fetchApi(setData, setLoading, setError);
   }
  };
  getData();
 }, [data]);
```

```
  return (
    <DataContext.Provider value={{ data, loading, error, setData }}>
      {children}
    </DataContext.Provider>
  );
};

const useData = () => {
  const context = useContext(DataContext);
  if (context === undefined) {
    throw new Error('useData must be used within a DataProvider');
  }
  return context;
};

const fetchApi = async (setData, setLoading, setError) => {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    localStorage.setItem('apiData', JSON.stringify(data));
    setData(data);
    setLoading(false);
  } catch (error) {
    setError(error);
    setLoading(false);
  }
};

export { DataProvider, useData };
```

Using the Custom Hook in a Component

Now we can use the custom hook **`useData`** within our components to efficiently access API data.

```
import React from 'react';
import { DataProvider, useData } from './DataProvider';

const ApiComponent = () => {
  const { data, loading, error } = useData();

  const clearCache = () => {
    localStorage.removeItem('apiData');
    window.location.reload();
  };

  if (loading) {
    return <div>Loading...</div>;
```

```
  }

  if (error) {
   return (
    <div>
     Error: {error.message}
     <button onClick={clearCache}>Retry</button>
    </div>
   );
  }

  return (
   <div>
    <h1>Data from API:</h1>
    <pre>{JSON.stringify(data, null, 2)}</pre>
    <button onClick={clearCache}>Clear Cache and Retry</button>
   </div>
  );
};

const App = () => (
 <DataProvider>
  <ApiComponent />
 </DataProvider>
);

export default App;
```

Importance of Security in Local Storage

It's crucial to emphasize that Local Storage is not a secure place to store sensitive information such as authentication tokens, passwords, or any personal data. Local Storage is accessible by any script running on the same page, making it vulnerable to Cross-Site Scripting (XSS) attacks.

Therefore, it's recommended to store in Local Storage only non-critical data for the security of both the application and the user.

🕶 Concept of Portals in React

In this chapter, we will explore the concept of Portals in React, understand what they are, how to use them, and provide practical examples to illustrate their usage.

What are Portals?

Portals in React offer an elegant way to render a child component into a DOM node that is outside the hierarchy of its parent component. This allows mounting a component in a completely separate place from the main DOM tree.

Concept of Portals

Portals are particularly useful in situations where you need a component to render outside the normal flow of the DOM, such as:

- **Modals and Dialogs**: Ensuring that modals appear correctly on top of other content.
- **Tooltips and Popovers**: Facilitating the management of overlays and dynamic positioning.
- **Context Menus**: Avoiding issues with z-index and overflow of parent elements.

How to Use a Portal

To create a Portal in React, you use the `createPortal` function from the `react-dom` module. This function takes two arguments:

- The content you want to render.
- The DOM node where you want to mount that content.

Basic Example of Usage

```
import React from 'react';
import ReactDOM from 'react-dom';

const portalRoot = document.getElementById('portal-root');

const Modal = ({ children }) => {
 return ReactDOM.createPortal(
  <div className='modal'>{children}</div>,
  portalRoot,
 );
};

const App = () => {
 const [showModal, setShowModal] = React.useState(false);

 const toggleModal = () => {
  setShowModal(!showModal);
 };

 return (
  <div>
   <button onClick={toggleModal}>Open Modal</button>
   {showModal && (
    <Modal>
     <h1>Modal Content</h1>
```

```
      <button onClick={toggleModal}>Close</button>
     </Modal>
    )}
   </div>
  );
};

export default App;
```

In this example, the `Modal` component is rendered inside the `portal-root` node, which is outside the `App` component's node.

Creating a Portal Node in the DOM

To ensure the above example works correctly, make sure you have a node in your HTML where the Portal will be mounted. Typically, this is added in the `index.html` file.

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <title>React App</title>
 </head>
 <body>
  <div id="root"></div>
  <div id="portal-root"></div>
  <script src="bundle.js"></script>
 </body>
</html>
```

This setup ensures that Portals function as expected within your React application.

Adding Styles to Components

In this chapter, we'll explore various techniques for adding styles to React components, including traditional CSS, CSS-in-JS, and SCSS modules. Each approach has its own advantages and disadvantages, and choosing the right one depends on your project's specific needs.

Traditional CSS Styles

Step 1: Creating the `styles` Object

Instead of using external CSS files, we define our styles directly within the component using a `styles` object. This approach is useful for local styles and integrates well with JSX.

```
// Button.js
import React from 'react';

const styles = {
 button: {
  backgroundColor: '#007bff',
  color: 'white',
  border: 'none',
  padding: '10px 20px',
  borderRadius: '4px',
  cursor: 'pointer',
  transition: 'background-color 0.3s ease',
 },
 buttonHover: {
  backgroundColor: '#0056b3',
 },
};

const Button = ({ children, onClick }) => {
 return (
  <button
   style={styles.button}
   onMouseOver={e =>
    (e.currentTarget.style.backgroundColor =
     styles.buttonHover.backgroundColor)
   }
   onMouseOut={e =>
    (e.currentTarget.style.backgroundColor = styles.button.backgroundColor)
   }
   onClick={onClick}
  >
   {children}
  </button>
 );
};

export default Button;
```

Pros and Cons

- **Pros**: Easy to understand and use, excellent for small projects.
- **Cons**: Can become hard to maintain in large projects due to lack of encapsulation and potential for naming conflicts.

CSS-in-JS with Styled Components

Step 1: Installing Styled Components

First, install the `styled-components` library.

```
npm install styled-components
```

Step 2: Creating Styled Components

Create components with integrated styles using `styled-components`.

```
// Button.js
import React from 'react';
import styled from 'styled-components';

const StyledButton = styled.button`
 background-color: #007bff;
 color: white;
 border: none;
 padding: 10px 20px;
 border-radius: 4px;
 cursor: pointer;
 transition: background-color 0.3s ease;

 &:hover {
  background-color: #0056b3;
 }
`;

const Button = ({ children, onClick }) => {
 return <StyledButton onClick={onClick}>{children}</StyledButton>;
};

export default Button;
```

Pros and Cons

- **Pros**: Encapsulated styles, support for themes, excellent for reusable components.
- **Cons**: May increase bundle size, syntax may be unfamiliar to some developers, and runtime processing can impact performance. Alternatives like `panda css` perform build-time processing to mitigate this.

SCSS Modules

Step 1: Installing SASS

Install `sass` to use SCSS in your project.

```
npm install sass
```

Step 2: Creating the SCSS Module File

Create a SCSS file for your component. With SCSS modules, you ensure styles are scoped locally to the component.

```
/* Button.module.scss */
.button {
 background-color: #007bff;
 color: white;
 border: none;
 padding: 10px 20px;
 border-radius: 4px;
 cursor: pointer;
 transition: background-color 0.3s ease;

 &:hover {
  background-color: #0056b3;
 }
}
```

Step 3: Importing the SCSS Module in the Component

Import the SCSS module into your component and apply the styles.

```
// Button.js
import React from 'react';
import styles from './Button.module.scss';

const Button = ({ children, onClick }) => {
 return (
  <button className={styles.button} onClick={onClick}>
   {children}
  </button>
 );
};

export default Button;
```

Pros and Cons

- **Pros**: Local and encapsulated styles, full support for SCSS.
- **Cons**: Additional setup required, can be complex for very large projects.

Comparison of Approaches

Traditional CSS

- **Use**: Ideal for small and quick projects.
- **Simplicity**: High.
- **Maintenance**: Can be challenging in large projects due to lack of encapsulation.

CSS-in-JS (Styled Components)

- **Use**: Ideal for highly reusable components and projects heavily using JavaScript.
- **Simplicity**: Medium, syntax may be a hurdle for some.
- **Maintenance**: High, thanks to encapsulation and theme support.
- **Performance**: May be less efficient due to runtime processing. Alternatives like `panda css` can mitigate this by performing build-time processing.

SCSS Modules

- **Use**: Ideal for projects requiring strong SCSS usage and style encapsulation.
- **Simplicity**: Medium-High, familiarity with SCSS is a plus.
- **Maintenance**: High, with local styles and powerful SCSS features.

🕶 Routing with react-router-dom

Introduction to React Router

In this chapter, we'll delve into managing routing in a React application using `react-router-dom`. Routing involves defining and handling different paths within a web application, allowing users to navigate between various views or pages without reloading the entire app.

`react-router-dom` is a popular library in the React ecosystem for implementing routing. It simplifies route creation, navigation between routes, and securing routes based on authentication and other factors.

Basic Concepts of React Router

Before diving into examples, let's review some key concepts you need to know about `react-router-dom`:

- **Router**: It's the container that wraps the application and manages the navigation history. We use `BrowserRouter` for standard web applications.
- **Routes**: Route definitions that specify which component should render for a particular URL.
- **Route**: A component used to define a route. Each `Route` is associated with a URL and a specific component.
- **Navigate**: A component used for programmatically redirecting the user to a new route.
- **Private Routes**: Routes that can only be accessed if certain conditions are met, such as being authenticated.

Installing react-router-dom

Before we begin, we need to install `react-router-dom` in our application. You can do this using npm or yarn:

```
npm install react-router-dom
```

or

```
yarn add react-router-dom
```

Basic Router Configuration

To set up routing in our application, we first wrap our app in a `BrowserRouter` and define routes using the `Routes` and `Route` components.

```
import React from 'react';
import {
 BrowserRouter as Router,
 Routes,
 Route,
 Navigate,
} from 'react-router-dom';
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';

const App = () => {
 return (
  <Router>
   <Routes>
    <Route path='/login' element={<Login />} />
    <Route path='/dashboard' element={<Dashboard />} />
    <Route path='*' element={<Navigate to='/login' replace />} />
   </Routes>
```

```
  </Router>
 );
};

export default App;
```

In this example, we've defined three routes:

- `/login` renders the `Login` component.
- `/dashboard` renders the `Dashboard` component.
- * redirects any undefined route to `/login`.

Private Routes

Private routes are routes that can only be accessed if the user is authenticated. To handle this, we can create a `PrivateRoute` component that checks if the user is authenticated before allowing access to the route.

```
// PrivateRoute.js
import React from 'react';
import { Navigate } from 'react-router-dom';
import { useUserContext } from '../UserContext';

const PrivateRoute = ({ children }) => {
 const { user } = useUserContext();
 return user.id ? children : <Navigate to='/login' replace />;
};

export default PrivateRoute;
```

We use the `UserContext` to check if the user is authenticated. If not, we redirect the user to the login page.

Using Context to Manage Authentication

Instead of Redux, we'll use React's Context API to manage the user's authentication state.

```
// UserContext.js
import { createContext, useContext, useState, useEffect } from 'react';

export const UserContext = createContext();

export const UserProvider = ({ children }) => {
 const [user, setUser] = useState(() => {
  const storedUser = localStorage.getItem('user');
  return storedUser
   ? JSON.parse(storedUser)
```

```
    : { id: null, name: '', email: '', role: 'user' };
  });

  const login = userData => {
   setUser(userData);
   localStorage.setItem('user', JSON.stringify(userData));
  };

  const logout = () => {
   setUser({ id: null, name: '', email: '', role: 'user' });
   localStorage.removeItem('user');
  };

  return (
   <UserContext.Provider value={{ user, login, logout }}>
    {children}
   </UserContext.Provider>
  );
};

export const useUserContext = () => {
  const context = useContext(UserContext);
  if (context === undefined) {
   throw new Error('useUserContext must be used within a UserProvider');
  }
  return context;
};
```

Complete Implementation Example

Let's integrate all these components into our main application and handle authentication and protected routes using `react-router-dom` and the Context API.

```
// App.js
import React from 'react';
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from 'react-router-dom';
import { UserProvider, useUserContext } from './UserContext';
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';
import PrivateRoute from './components/PrivateRoute';
import RoleRoute from './components/RoleRoute';

const App = () => {
  return (
   <UserProvider>
```

```
    <Router>
     <Routes>
      <Route path='/login' element={<Login />} />
      <Route
       path='/dashboard'
       element={
        <PrivateRoute>
         <Dashboard />
        </PrivateRoute>
       }
      />
      <Route
       path='/admin'
       element={
        <RoleRoute requiredRole='admin'>
         <AdminPage />
        </RoleRoute>
       }
      />
      <Route path='*' element={<Navigate to='/login' replace />} />
     </Routes>
    </Router>
   </UserProvider>
  );
};

export default App;
```

Nested Routing and Lazy Loading

Advantages of Nested Routes

Nested routes help organize application sections better, making maintenance and scalability easier. With nested routes, we can define routes within other routes, allowing a section of the app to have its own set of routes.

Basic Setup of Nested Routes

Suppose we have an application with a dashboard that has multiple sections like Home, Profile, and Settings. Let's define these nested routes in our main component.

```
import React from 'react';
import {
 BrowserRouter as Router,
 Routes,
 Route,
 Navigate,
 Outlet,
} from 'react-router-dom';
```

```
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';
import Home from './pages/Home';
import Profile from './pages/Profile';
import Settings from './pages/Settings';
import PrivateRoute from './components/PrivateRoute';
import RoleRoute from './components/RoleRoute';

const App = () => {
 return (
  <Router>
   <Routes>
    <Route path='/login' element={<Login />} />
    <Route
     path='/dashboard'
     element={
      <PrivateRoute>
       <Dashboard />
      </PrivateRoute>
     }
    >
     <Route path='home' element={<Home />} />
     <Route path='profile' element={<Profile />} />
     <Route path='settings' element={<Settings />} />
    </Route>
    <Route path='*' element={<Navigate to='/login' replace />} />
   </Routes>
  </Router>
 );
};

export default App;
```

In this example, we've defined nested routes within `/dashboard`. The `home`, `profile`, and `settings` routes are accessible only when the user is on `/dashboard`.

Using Lazy Loading

Lazy loading is a technique to load components only when they are needed, instead of loading them all upfront. This improves app performance, especially if you have many routes or heavy components.

To implement lazy loading, we use `React.lazy()` function and React's `Suspense` component.

```
import React, { Suspense, lazy } from 'react';
import {
 BrowserRouter as Router,
 Routes,
 Route,
 Navigate,
} from 'react-router-dom';
```

```
import Login from './pages/Login';
import PrivateRoute from './components/PrivateRoute';

const Dashboard = lazy(() => import('./pages/Dashboard'));
const Home = lazy(() => import('./pages/Home'));
const Profile = lazy(() => import('./pages/Profile'));
const Settings = lazy(() => import('./pages/Settings'));

const App = () => {
 return (
  <Router>
   <Suspense fallback={<div>Loading...</div>}>
    <Routes>
     <Route path='/login' element={<Login />} />
     <Route
      path='/dashboard'
      element={
       <PrivateRoute>
        <Dashboard />
       </PrivateRoute>
      }
     >
      <Route path='home' element={<Home />} />
      <Route path='profile' element={<Profile />} />
      <Route path='settings' element={<Settings />} />
     </Route>
     <Route path='*' element={<Navigate to='/login' replace />} />
    </Routes>
   </Suspense>
  </Router>
 );
};

export default App;
```

Here, we've wrapped our routes with the `Suspense` component and specified a `fallback` component to show while nested components are loading.

Logical Scope of Elements

The structure of nested routes allows us to define the logical scope of elements clearly. Each section of the application can have its own set of routes, making it easier to understand how components are structured and loaded.

- **Primary Routes**: Main routes like `/login` and `/dashboard` are defined at the top level.
- **Secondary Routes**: Within `Dashboard`, we define secondary routes like `home`, `profile`, and `settings`.

This hierarchical organization helps maintain clean and modular code. Each section of the application is encapsulated within its own logical scope, making it easier to understand how components are structured and loaded.

Complete Example

Let's show a complete example that combines nested routes, lazy loading, and authentication handling using Context API.

```
// App.js
import React, { Suspense, lazy } from 'react';
import {
 BrowserRouter as Router,
 Routes,
 Route,
 Navigate,
} from 'react-router-dom';
import { UserProvider } from './UserContext';
import Login from './pages/Login';
import PrivateRoute from './components/PrivateRoute';

const Dashboard = lazy(() => import('./pages/Dashboard'));
const Home = lazy(() => import('./pages/Home'));
const Profile = lazy(() => import('./pages/Profile'));
const Settings = lazy(() => import('./pages/Settings'));

const App = () => {
 return (
  <UserProvider>
   <Router>
    <Suspense fallback={<div>Loading...</div>}>
     <Routes>
      <Route path='/login' element={<Login />} />
      <Route
       path='/dashboard'
       element={
        <PrivateRoute>
         <Dashboard />
        </PrivateRoute>
       }
      >
       <Route path='home' element={<Home />} />
       <Route path='profile' element={<Profile />} />
       <Route path='settings' element={<Settings />} />
      </Route>
      <Route path='*' element={<Navigate to='/login' replace />} />
     </Routes>
    </Suspense>
   </Router>
  </UserProvider>
```

```
  );
};

export default App;
```

In this example:

- We use `React.lazy()` to lazily load components.
- `Suspense` wraps our routes to show a fallback while nested components are loading.
- `PrivateRoute` ensures only authenticated users can access nested routes under `/dashboard`.

This structure allows for efficient loading and clear code organization, enhancing navigation and user experience in the application.

Error Handling with Error Boundaries

Introduction

In this chapter, we'll learn how to efficiently handle errors in our React applications using Error Boundaries. This technique allows us to capture errors in specific components without affecting the rest of the application. We'll explore how to implement Error Boundaries and manage them to ensure our application continues to function smoothly even when errors occur.

What is an Error Boundary?

An Error Boundary in React is a component that catches errors in any of its child components during rendering. Similar to how a Provider wraps other components to provide them context, an Error Boundary wraps other components to handle errors that may arise.

The idea is that if a component within the Error Boundary fails, only that specific part of the application is affected. The Error Boundary can display an error message or a fallback UI without crashing the entire application.

Basic Implementation of an Error Boundary

Let's start by implementing a basic Error Boundary. We'll use class components since Error Boundaries currently only work with class components.

```
import React, { Component } from 'react';

class ErrorBoundary extends Component {
 constructor(props) {
  super(props);
```

```
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.log(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Oops! Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

In this code:

- `getDerivedStateFromError`: This method is called when an error is thrown in a child component. It updates the state to indicate that an error has occurred.
- `componentDidCatch`: This method is used to log errors or perform additional tasks.
- `render`: If there's an error, an error message is shown; otherwise, the child components are rendered normally.

Using the Error Boundary

To use the Error Boundary, simply wrap the components you want to monitor with your Error Boundary.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import ErrorBoundary from './ErrorBoundary';

ReactDOM.render(
  <ErrorBoundary>
    <App />
  </ErrorBoundary>,
  document.getElementById('root'),
);
```

This way, any errors occurring within `App` will be caught by `ErrorBoundary`, and instead of crashing the whole application, it will display the error message.

Handling Errors in Fetch Requests

Error Boundaries do not catch asynchronous errors such as those in fetch requests. To handle these errors, we need to use a different approach.

```
import React, { useState, useEffect } from 'react';

const DataFetcher = () => {
 const [data, setData] = useState(null);
 const [error, setError] = useState(null);

 useEffect(() => {
  fetch('https://rickandmortyapi.com/api/character')
   .then(response => {
    if (!response.ok) {
     throw new Error('Network response was not ok');
    }
    return response.json();
   })
   .then(data => setData(data))
   .catch(error => setError(error));
 }, []);

 if (error) {
  return <div>Oops! Something went wrong: {error.message}</div>;
 }

 return (
  <div>
   {data ? (
    data.results.map(character => (
     <div key={character.id}>{character.name}</div>
    ))
   ) : (
    <div>Loading...</div>
   )}
  </div>
 );
};

export default DataFetcher;
```

In this component:

- `useEffect` is used to make the fetch request.

- If an error occurs during the request, the error state is set, and an error message is displayed.

Integrating Error Boundaries with Fetch Requests

We can combine Error Boundaries with fetch error handling for a more robust solution.

```
import React, { Component } from 'react';

class ErrorBoundary extends Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }

 static getDerivedStateFromError(error) {
  return { hasError: true };
 }

 componentDidCatch(error, errorInfo) {
  console.log(error, errorInfo);
 }

 render() {
  if (this.state.hasError) {
   return <h1>Oops! Something went wrong.</h1>;
  }

  return this.props.children;
 }
}

const DataFetcher = () => {
 const [data, setData] = useState(null);
 const [error, setError] = useState(null);

 useEffect(() => {
  fetch('https://rickandmortyapi.com/api/character')
   .then(response => {
    if (!response.ok) {
     throw new Error('Network response was not ok');
    }
    return response.json();
   })
   .then(data => setData(data))
   .catch(error => setError(error));
 }, []);

 if (error) {
  throw error;
 }
```

```
  return (
   <div>
    {data ? (
     data.results.map(character => (
      <div key={character.id}>{character.name}</div>
     ))
    ) : (
     <div>Loading...</div>
    )}
   </div>
  );
};

const App = () => (
 <ErrorBoundary>
  <DataFetcher />
 </ErrorBoundary>
);

export default App;
```

In this example, any fetch errors will throw an exception caught by the Error Boundary, ensuring the fallback UI is displayed when necessary.

Benefits of Lazy Loading

Lazy loading allows us to load components only when needed, which can improve our application's performance by reducing initial load time and memory usage.

```
import React, { Suspense, lazy } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

const App = () => (
 <Suspense fallback={<div>Loading...</div>}>
  <LazyComponent />
 </Suspense>
);

export default App;
```

In this example, `LazyComponent` is loaded only when necessary, while displaying "Loading..." in the meantime.

Integrating Lazy Loading and Error Boundaries

Combining lazy loading with Error Boundaries can offer a very efficient and robust solution.

```
import React, { Suspense, lazy } from 'react';
import ErrorBoundary from './ErrorBoundary';

const LazyComponent = lazy(() => import('./LazyComponent'));

const App = () => (
 <ErrorBoundary>
  <Suspense fallback={<div>Loading...</div>}>
   <LazyComponent />
  </Suspense>
 </ErrorBoundary>
);

export default App;
```

This way, we handle both errors and lazy loading of components, ensuring a smooth and uninterrupted user experience.

Improving Your Skills with Axios Interceptor

Introduction

In this chapter, we'll delve into the world of Axios interceptors in React. We'll learn how to use them to efficiently handle HTTP requests and responses, giving us the flexibility and control needed to build robust and secure applications. We'll create a project from scratch, set up Axios, and implement interceptors to handle authentication, errors, and more.

What is Axios and Why Use It?

Axios is a popular library for making HTTP requests in JavaScript. It simplifies communication with APIs, offering a clean syntax and numerous additional features that the native Fetch API doesn't provide as directly. One of these key features is the use of interceptors, which allow us to intercept and modify requests and responses before they reach the server or client.

Creating a Project with Vite

Let's start by creating a new React project using Vite, a fast and lightweight bundler.

- **Create the Project**:

  ```
  npm create vite@latest
  ```

  Choose a name for your project and select "React" and "TypeScript" as the desired options.

- **Install Axios**:

```
npm install axios
```

Configuring Axios Interceptor

Now that we have our project set up, let's create an interceptor to handle our requests and responses.

- **Create the Interceptor**: Create an `axios.interceptor.ts` file in a `services` folder.

```
import axios from 'axios';

const axiosInstance = axios.create();

axiosInstance.interceptors.request.use(
 config => {
  // Modify the request before sending it
  const token = localStorage.getItem('token');
  if (token) {
   config.headers['Authorization'] = `Bearer ${token}`;
  }
  return config;
 },
 error => {
  return Promise.reject(error);
 },
);

axiosInstance.interceptors.response.use(
 response => {
  // Modify the response before sending it to the client
  return response;
 },
 error => {
  if (error.response.status === 401) {
   // Handle authentication errors
   console.log('Unauthorized, redirecting to login...');
  }
  return Promise.reject(error);
 },
);

export default axiosInstance;
```

  In this code, we intercept all requests to add an authentication token if available and handle specific response errors, such as authentication failure.

- **Use the Interceptor**: In your main component (`App.tsx`), import and use the interceptor to make a request.

```
import React, { useEffect, useState } from 'react';
import axiosInstance from './services/axios.interceptor';

const App = () => {
 const [data, setData] = useState(null);
 const [error, setError] = useState(null);

 useEffect(() => {
  const fetchData = async () => {
   try {
    const response = await axiosInstance.get(
     '<https://rickandmortyapi.com/api/character/1>',
    );
    setData(response.data);
   } catch (error) {
    setError(error);
   }
  };

  fetchData();
 }, []);

 if (error) return <div>Error: {error.message}</div>;
 if (!data) return <div>Loading...</div>;

 return (
  <div>
   <h1>{data.name}</h1>
   <p>Status: {data.status}</p>
  </div>
 );
};
export default App;
```

Advantages of Using Interceptors

- **Centralized Authentication**: Interceptors allow us to add authentication headers to all requests from a single place.
- **Error Handling**: We can globally catch and handle errors, showing custom messages or redirecting the user as needed.
- **Logging and Debugging**: We can log requests and responses to monitor network activity and debug issues.

Advanced Error Handling

Let's enhance our interceptor to handle different types of errors and refresh the authentication token when necessary.

```
axiosInstance.interceptors.response.use(
 response => {
  return response;
 },
 async error => {
  if (error.response.status === 401) {
   // Attempt to refresh the token
   try {
    const refreshToken = localStorage.getItem('refreshToken');
    const response = await axios.post('/auth/refresh-token', {
     token: refreshToken,
    });
    localStorage.setItem('token', response.data.token);
    error.config.headers['Authorization'] = `Bearer ${response.data.token}`;
    return axiosInstance(error.config);
   } catch (e) {
    // Redirect to login if token refresh fails
    console.log('Redirecting to login...');
   }
  }
  return Promise.reject(error);
 },
);
```

With this setup, if a request receives a 401 error (Unauthorized), we attempt to refresh the token. If the refresh fails, we redirect the user to the login page.

TypeScript Con De Tuti

Introduction

Hello everyone! This is Gentleman at the keyboard, bringing you an in-depth analysis of TypeScript and how this language can revolutionize the way we work in development teams. This book is an expansion of a video I uploaded on YouTube, where we talked about the basics of TypeScript, the advantages, and how it can help in a team. We will dive not only into the video content but also expand with practical examples, code, and key reflections so that you, my dear developers, can take your code to the next level.

Why TypeScript?

What is TypeScript?

TypeScript is a "superset" of JavaScript, which means it has everything JavaScript offers but adds more functionalities that are especially useful in large projects or teams. As I mentioned in the video, if JavaScript is good, TypeScript is JavaScript on steroids.

```
// Basic TypeScript example
let message: string = "Hello, TypeScript!";
console.log(message);
```

Advantages of Using TypeScript in Teams

- **Type Safety**: Reduces common JavaScript errors by allowing variable type specification.
- **Maintainability**: The code is easier to understand and maintain.
- **Refactoring**: Safe and easy to perform thanks to the type system.

Fundamental Concepts of TypeScript

Variables and Types

One of the pillars of TypeScript is its typing capability. This avoids many common errors in JavaScript.

```
// Typing example in TypeScript
let isActive: boolean = true;
let quantity: number = 123;
```

Interfaces and Classes

TypeScript allows defining interfaces and classes, which facilitates the implementation of advanced design patterns and code organization.

```
interface User {
  name: string;
  age: number;
}

class Employee implements User {
  constructor(
    public name: string,
    public age: number,
  ) {}
}
```

TypeScript in Practice

Practical Case Analysis

Let's analyze the video segment where we discussed variable mutability and how TypeScript can help control it.

```
// Immutability example in TypeScript
let x: number = 10;
// x = "Type change"; // This will generate an error in TypeScript.
```

Creating Effective Methods

We also discussed how the lack of clarity in types can lead to errors in seemingly simple methods.

```
function sum(a: number, b: number): number {
  return a + b;
}
```

Best Practices and Patterns

Working with Teams

- **Clarity**: Always use types.

- **Documentation**: Take advantage of TypeScript features to document the code.
- **Code Review**: Encourage code reviews focused on improving typing.

Tools and Extensions

We'll talk about tools that can be integrated with TypeScript to further improve workflow, such as linters, code formatters, and more.

What is Transpiling?

Transpiling, in the programming world, is the process of converting code written in one language (or version of a language) to another language (or version of that language). In our case, we often talk about converting TypeScript to JavaScript. Basically, transpiling is like translating.

Why Do We Need to Transpile?

TypeScript is amazing because it gives us superpowers: types, interfaces, and many aids to avoid errors. But browsers and Node.js don't understand TypeScript, they only speak JavaScript. So we need a translator, and that translator is the TypeScript compiler (tsc).

Practical Example

Let's see this in action with a simple example. Imagine we have a TypeScript file `script.ts` with the following code:

```
// script.ts
let message: string = "Hello, world";
console.log(message);
```

This file contains a `message` variable of type string and a `console.log` to display it. Now, for our browser to understand this code, we need to transpile it to JavaScript. We do this with the `tsc` (TypeScript Compiler) command:

```
tsc script.ts
```

After running this command, we get a `script.js` file with the following content:

```
// script.js
var message = "Hello, world";
console.log(message);
```

As you can see, the TypeScript compiler has converted (or "transpiled") the TypeScript code to JavaScript.

A Little More Magic: Compiler Configuration

We can do much more with our TypeScript compiler configuration. For example, we can define how we want the transpiling process to behave through a `tsconfig.json` file. Here's a basic example:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "es6", // Indicates which version of JavaScript we want to transpile to
    "outDir": "./dist", // Folder where the transpiled files will be saved
    "strict": true // Enables all strict checks
  },
  "include": ["src/**/*.ts"], // Files to include in the transpilation
  "exclude": ["node_modules"] // Files or folders to exclude
}
```

Making Magic with `tsc --watch`

If you want to take your workflow to the next level, you can use the `tsc --watch` command, which will watch for any changes in your TypeScript files and automatically transpile them to JavaScript. It's like having a personal assistant always ready to help.

```
tsc --watch
```

In Summary

Transpiling is an essential process that allows us to write in modern languages with better features and then convert that code to a language that browsers and Node.js can understand. It's like having a translator that converts our words into something everyone can understand.

So next time you hear "transpile," you know it's just a translation process, ensuring our TypeScript brilliance reaches any JavaScript environment intact and clear.

Understanding Typing in JavaScript and TypeScript

JavaScript: A Dynamically Typed Language

Although it may not seem like it at first glance, JavaScript does have a type system, but it is dynamic. This means that a variable's type can change during the program's execution, introducing flexibility but also a propensity for hard-to-track errors. This is where JavaScript shows both its flexibility and limitations, as this feature can lead to confusion in large projects or when working in teams.

The V8 JavaScript engine, used by most modern browsers like Chrome and Node.js, handles dynamic typing in a particular way. When a variable or method is defined, V8 assigns an initial type based on the assigned value. This information is stored in a memory buffer.

```
// Example of dynamic typing in JavaScript
let value = "Hello";

 // Initially a string
value = 100; // Now a number
```

In this example, `value` changes from a string to a number. It's like your dog suddenly deciding to be a cat! While useful at times, this can cause problems.

When a variable's type changes, V8 has to restructure how this variable is stored in memory. This involves recalculating and reallocating memory for the new data shape, which can be costly in terms of performance.

```
let a = 1; // 'a' is a number
a = "one"; // 'a' is now a string
```

The V8 engine detects the type change and adjusts memory and internal references to accommodate the new type. This process can slow down execution if it occurs frequently.

TypeScript: Stability and Security Through Static Typing

In contrast, TypeScript introduces a static typing system, which forces the definition of data types for variables and functions from the beginning. This helps avoid many common JavaScript errors by making the code more predictable and easier to debug. By using TypeScript, you can have much stricter control over how data is handled in applications, resulting in more robust and secure code.

```
let value: number = 100;
// value = "Hello"; // This will cause an error in TypeScript
```

And that's it! Now `value` can't change types and you ensure it will always be a number. It's like having a dog that will always be a dog.

TypeScript: The Superhero of Development

First, let's imagine that TypeScript is a superhero. Its mission: to save us from JavaScript code errors and nightmares. But, like any good superhero, it has its limits and areas of operation. In this case, TypeScript only uses its superpowers during development.

What Does TypeScript Do?

As we said earlier, TypeScript is JavaScript on steroids. It allows you to add types to your variables and functions, which helps avoid common errors. But here's the trick: when your application runs in the browser or Node.js, all that TypeScript code has been transformed (or transpiled) into JavaScript. It's like our superhero takes off the suit and puts on a regular uniform.

Linters: The Battle Companions

Now, let's talk about linters. They are like the superhero's teammates. Linters, like ESLint, work hand-in-hand with TypeScript to keep your code clean and error-free. While TypeScript focuses on types and code structure, linters handle style rules and best practices.

Practical Example

Let's look at an example to make this clearer. Suppose we are working on a super cool application:

```
// TypeScript
function greet(name: string): string {
  return `Hello, ${name}`;
}

const greeting = greet("Gentleman");
console.log(greeting);
```

Here we have a `greet` function that takes a `name` of type string and returns a greeting. TypeScript ensures that we always pass a string to this function. But, when the time comes, this is what runs in your browser:

```
// Transpiled JavaScript
function greet(name) {
  return "Hello, " + name;
}

var greeting = greet("Gentleman");
console.log(greeting);
```

Voilá! All the TypeScript code has been transformed into JavaScript.

Integration with ESLint

Now, let's add our linter to the team. Imagine we have a rule that says we must always use single quotes. Here's the `.eslintrc.json` file:

```
{
  "rules": {
    "quotes": ["error", "single"]
```

If someone rebels and uses double quotes instead of single quotes, ESLint will scold us and remind us to follow the rules:

```
// Incorrect code according to ESLint
function greet(name: string): string {
  return "Hello, " + name; // ESLint will notify us of this error
}
```

With ESLint integrated, our code will stay clean and consistent, teaming up with TypeScript to ensure everything is in order.

In Summary

TypeScript is our superhero during development, helping us write safer and more predictable code. But once everything is ready for action, it transforms into JavaScript. And with linters as battle companions, we keep our code in line.

So, the next time someone tells you that TypeScript "only works during development," you'll know that while true, that's precisely where it makes the difference.

Practical Example in TypeScript: The Use of any and the Importance of Typing

Let's see why, although TypeScript gives us powerful tools, using them incorrectly can lead us to the same problems we might have in JavaScript. Get ready for a little mental challenge!

Step 1: Declare a variable with a specific type

Let's start with something simple. In TypeScript, we can specify a variable's type to ensure it always contains the correct type of value. Look at this example:

```
let number: number = 5;
```

Now, if we try to assign a different type of value, like a `string`, TypeScript will show us an error. This is great because it prevents errors at compile time. Let's see:

```
number = "this should fail"; // Error: Type 'string' is not assignable to type 'number'.
```

Step 2: Use of the any type

The `any` type in TypeScript allows us to assign any type of value to a variable, similar to what happens by default in JavaScript. Let's see:

```
let flexibleVariable: any = 5;
flexibleVariable = "I can be a string too";
flexibleVariable = true; // And now a boolean!
```

With **any**, there are no compile-time errors, regardless of the type of data we assign. This gives us a lot of flexibility, but also eliminates the type safety guarantees that TypeScript offers.

Provocative question

Now, here comes the key question: If we are using **any**, do you think it's okay? If you think not... then you don't like JavaScript! It would be the same as using 'any' in all our variables.

Exactly! Most would say that it is not a good practice to use **any**, as we lose all the benefits of static typing that TypeScript offers. It's like going back to JavaScript, where we can easily make type errors.

Using **any** takes us back to JavaScript's flexibility (and dangers). While **any** can be useful in situations where we need a temporary solution or when working with third-party libraries for which we don't have types, we should avoid it in our main code. In TypeScript, the goal is to leverage the type system to write safer and more maintainable code.

Primitive Types in TypeScript

TypeScript enriches the set of primitive types of JavaScript, providing more robust control and options for variable declaration. Here are the main primitive types:

- **Boolean**: True or false value.

  ```
  let isActive: boolean = true;
  ```

- **Number**: Any number, including decimals, hexadecimal, binary, and octal.

  ```
  let amount: number = 56;
  let hexadecimal: number = 0xf00d;
  let binary: number = 0b1010;
  let octal: number = 0o744;
  ```

- **String**: Text strings.

  ```
  let name: string = "Gentleman";
  ```

- **Array**: Arrays that can be typed.

```
let numberList: number[] = [1, 2, 3];
let stringList: Array<string> = ["one", "two", "three"];
```

- **Tuple**: Allow expressing an array with a fixed number of elements and known types, but not necessarily the same type.

```
let tuple: [string, number] = ["hello", 10];
```

- **Enum**: A way to give more friendly names to sets of numeric values.

```
enum Color {
  Red,
  Green,
  Blue,
}
let c: Color = Color.Green;
```

- **Any**: For values that can change type over time, it is a way to tell TypeScript to handle the variable like pure JavaScript.

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // now it's a boolean
```

- **Void**: Absence of any type, commonly used as a return type in functions that do not return anything.

```
function warnUser(): void {
  console.log("This is a warning!");
}
```

- **Null and Undefined**: Subtypes of all other types.

```
let u: undefined = undefined;
let n: null = null;
```

Type Inference in TypeScript

TypeScript is smart when it comes to inferring variable types based on available information, such as the initial value of variables. However, this inference can be tricky.

```
let message = "Hello, world";
// `message` is automatically inferred as `string`
```

Traps of Inference in TypeScript: A Practical Example

Hello, community! Today we will delve into a fascinating and sometimes tricky topic of TypeScript: the traps of type inference, using a practical example that will show us how TypeScript handles type inference within complex control structures.

Problematic Example

Consider the following TypeScript code:

```
const arrayOfValues = [
  {
    number: 1


,
    label: "label1",
  },
  {
    number: 2,
  },
];

const method = (param: typeof arrayOfValues) => {
  const indexArray = [1, 2];

  indexArray.forEach((index) => {
    if (param[index].label) {
      // param[index].label => string | undefined
      console.log(param[index].label); // param[index].label => string | undefined
    }
  });
};
```

In this example, we have an `arrayOfValues` that contains objects with `number` and `label` properties. However, you will notice that the second object in the array does not have the `label` property defined. This makes the type of `label` inferred as `string | undefined`.

Inference Problem

When we pass `arrayOfValues` to the `method` function and use `forEach` to iterate over an array of indexes, we make a check in each iteration to see if `label` is present. While inside the `if` block, one might expect TypeScript to understand that `label` is not `undefined` due to the check, the reality is that TypeScript still considers `param[index].label` to be `string | undefined` both inside and outside the `if`.

Why does this happen?

TypeScript does not carry a "state" of the type through the flow of the code in the same way it would in a simpler context. Although within the `if` block we have already verified that `label` exists, TypeScript does not have a "memory" of this check for future references in the same code block. This is especially true when iterating or using more complex structures like `forEach`, `for`, etc., where type checks do not "propagate" beyond the immediate scope in which they are performed.

Tip for Handling Inference in Iterative Blocks

To better handle these cases and ensure that the code is type-safe without relying on TypeScript's inference, you might consider the following practices:

- **Assign to Temporary Variables**: Sometimes, assigning to a temporary variable within the block can help.

```
indexArray.forEach((index) => {
  const label = param[index].label;
  if (label) {
    console.log(label); // TypeScript understands that label is string here
  }
});
```

- **Type Refinement with Guard Types**: Use guard types to refine the types within iterative or conditional blocks.

```
indexArray.forEach((index) => {
  if (typeof param[index].label === "string") {
    console.log(param[index].label); // Now it is safe to say it is a string
  }
});
```

Classes, Interfaces, Enums, and Const: How to Use Them for Typing in TypeScript?

Each of these elements has its own magic to help us write cleaner, scalable, and secure code. Let's break down each and see how and when to use them. 🎩✨

1. Classes

Classes in TypeScript are not only a template for creating objects but can also be used as types.

```
class Automobile {
  constructor(
    public brand: string,
    public model: string,
```

```
  ) {}
}

let myCar: Automobile = new Automobile("Toyota", "Corolla");
```

In the example, `Automobile` not only defines a class but also a type. When we say `let myCar: Automobile`, we are using the class as a type, ensuring that `myCar` meets the structure and behavior defined in the `Automobile` class.

2. Interfaces

Interfaces are powerful in TypeScript due to their ability to define contracts for classes, objects, and functions without generating JavaScript at compile time. They are ideal for defining data shapes and are extensively used in object-oriented programming and integration with libraries.

```
interface Vehicle {
  brand: string;
  start(): void;
}

class Truck implements Vehicle {
  constructor(
    public brand: string,
    public loadCapacity: number,
  ) {}
  start() {
    console.log("The truck is starting...");
  }
}
```

Interfaces not only allow typing objects and classes but can also be extended and combined, which is excellent for keeping code organized and reusable.

3. Enums

Enums allow defining a set of named constants. They are useful for handling a set of related values, providing a way to group them under a single type.

```
enum Color {
  Red,
  Green,
  Blue,
}

let favoriteColor: Color = Color.Green;
```

Using enums improves code readability and reduces the possibility of errors by restricting the values a variable can take.

4. Const Assertions

In TypeScript, `const` not only defines a constant at runtime but can also be used to make type assertions. Using `as const`, we can tell TypeScript to treat the type more specifically and literally.

```
let config = {
  name: "Application",
  version: 1,
} as const;

// config.name = "Another App"; // Error, because name is a constant.
```

This is especially useful for defining objects with properties that will never change their values once assigned.

👓 Type vs Interface in TypeScript: When and How to Use Them

Although both can be used to define types in TypeScript, they have their particularities and ideal use cases. Let's break down the differences and understand when it's better to use each one. 🚀

What is `interface`?

An `interface` in TypeScript is primarily used to describe the shape objects should have. It is a way to define contracts within your code as well as with external code.

```
interface User {
  name: string;
  age?: number;
}

function greet(user: User) {
  console.log(`Hello, ${user.name}`);
}
```

Interfaces are ideal for object-oriented programming in TypeScript, where you can use them to ensure certain classes implement specific methods and properties.

Advantages of using `interface`:

- **Extensibility**: Interfaces are extensible and can be extended by other interfaces. Using `extends`, an interface can inherit from another, which is great for keeping large codebases well-organized.

- **Declaration Merging**: TypeScript allows `interface` declarations to be merged automatically. If you define the same interface in different places, TypeScript combines them into a single interface.

What is `type`?

The `type` alias can be used to create a custom type and can be assigned to any data type, not just objects. `type` is more versatile than interfaces in certain aspects.

```
type Point = {
  x: number;
  y: number;
};

type D3Point = Point & { z: number };
```

Advantages of using `type`:

- **Union and Intersection Types**: With `type`, you can easily use union and intersection types to combine existing types in complex and useful ways.
- **Primitive and Tuple Types**: `type` can be used for aliasing primitive types, unions, intersections, and tuples.

When to use `interface` or `type`?

- **Use `interface` when:**
  - You need to define a 'contract' for classes or the shape of objects.
  - You want to take advantage of interfaces' extension and merging capabilities.
  - You are creating a type definition library or API that will be used in other TypeScript projects.
- **Use `type` when:**
  - You need to use unions or intersections.
  - You want to use tuples and other types that cannot be expressed with an `interface`.
  - You prefer to work with more flexible types and do not need to extend or implement them from classes.

The Concept of Shape in TypeScript

Now let's talk about a fundamental concept in TypeScript that helps us manage the structure and type of our objects: the **shape**. This concept is crucial for understanding how TypeScript handles typing and how we can make the most of our code. So, let's dive in! 🚀

What is Shape?

The concept of **shape** in TypeScript refers to the structure an object must have to be considered of a certain type. Basically, when we define a type or an interface, we are defining the shape that any object of that type must follow.

```
interface User {
  name: string;
  age: number;
}

let user: User = {
  name: "John",
  age: 25,
};
```

In this example, `User` defines the shape that the `user` object must have: it must have the properties `name` and `age` of types `string` and `number`, respectively.

Type Inference and Shape

TypeScript is very good at inferring types based on the values we provide. However, type inference also relies on the shape of objects.

```
let anotherUser = {
  name: "Anna",
  age: 30,
};
```

Here, TypeScript will infer that `anotherUser` has the shape `{ name: string; age: number; }` without us having to specify it explicitly.

Traps of Inference with Shape

Sometimes, relying on type inference can lead to tricky situations, especially when working with complex objects and arrays. Let's revisit an example of how this can be problematic:

```
const arrayOfValues = [
  {
    number: 1,
    label: "label1",
  },
```

```
  {
    number: 2,
  },
];

const method = (param: typeof arrayOfValues) => {
  const indexArray = [1, 2];

  indexArray.forEach((index) => {
    if (param[index].label) {
      console.log(param[index].label);
    }


  });
};
```

In this previously seen case, `param[index].label` remains `string | undefined` both outside and inside the `if` block, despite having checked its existence. Why does this happen? Because TypeScript cannot guarantee that the shape will remain constant throughout the iteration without storing the check in a variable.

Proper Handling of Shape

To handle these situations correctly, it is better to store the checks in a variable, which gives TypeScript a clearer hint about the shape:

```
const method = (param: typeof arrayOfValues) => {
  const indexArray = [1, 2];

  indexArray.forEach((index) => {
    const item = param[index];
    if (item.label) {
      console.log(item.label);
    }
  });
};
```

Now, TypeScript understands that `item.label` inside the `if` is a `string` and not `undefined`.

Understanding union and intersection in TypeScript

Let's explore two fundamental operators in TypeScript that allow us to handle types flexibly and powerfully: | (union) and & (intersection). These operators are key to defining complex types and handling different scenarios in our programs. Let's dive into them!

The | (Union) Operator

The | operator in TypeScript is used to combine types so that a value can be one of those types. That is, if we have `TypeA | TypeB`, we are saying that a variable can be of type `TypeA` or of type `TypeB`.

```
type Result = "success" | "error";

let state: Result;

state = "success"; // valid
state = "error"; // valid
state = "other"; // invalid, TypeScript will raise an error
```

In this example, `state` can be `"success"` or `"error"`, but it cannot be any other value.

Combining Types with Shared Properties

When we use the | operator to combine types that share some properties, these properties are retained in the union only if they are common to all included types. Let's look at an example to better understand this concept:

```
interface Dog {
  type: "dog";
  barks: boolean;
}

interface Cat {
  type: "cat";
  meows: boolean;
}

type Animal = Dog | Cat;

function processAnimal(animal: Animal) {
  // We can only access the 'type' property common to both types
  console.log(animal.type);

  // This would raise an error, as 'barks' or 'meows' depend on the specific type
  // console.log(animal.barks); // Error: 'barks' does not exist on type 'Animal'.
  // console.log(animal.meows); // Error: 'meows' does not exist on type 'Animal'.
}

let myDog: Dog = { type: "dog", barks: true };
let myCat: Cat = { type: "cat", meows: true };

processAnimal(myDog); // Expected output: "dog"
processAnimal(myCat); // Expected output: "cat"
```

In this example, `Animal` is a union of `Dog` and `Cat`. Although both types have the `type` property, specific properties like `barks` and `meows` are only available when working with a specific type (`Dog` or `Cat`), not in the

`Animal` type as a whole.

The & (Intersection) Operator

On the other hand, the `&` operator in TypeScript is used to create a type that has all the properties of the types we are combining. That is, `TypeA & TypeB` means a type that has all the properties of `TypeA` and all the properties of `TypeB`.

```
interface Person {
  name: string;
}

interface Employee {
  salary: number;
}

type NamedEmployee = Person & Employee;

let employee: NamedEmployee = {
  name: "John",
  salary: 3000,
};
```

In this case, `NamedEmployee` is a type that has both `name` and `salary`, combining the properties of `Person` and `Employee`.

Using | and & Together

We can combine `|` and `&` to create even more complex and specific types according to our needs:

```
type Options = { mode: "modeA" | "modeB" } & { size: "small" | "large" };

let configuration: Options = {
  mode: "modeA",
  size: "small",
};
```

In this example, `configuration` must have both `mode` (which can be `"modeA"` or `"modeB"`) and `size` (which can be `"small"` or `"large"`).

Key Differences

- **`|` (Union)**: Used to combine types where a value can be any of those types.
- **`&` (Intersection)**: Used to combine types where a value must have all the properties of those types.

Understanding `typeof` in TypeScript

Now let's see the use of the `typeof` operator in TypeScript and how it can help us handle complex types more efficiently.

Concept of `typeof`

In TypeScript, `typeof` is an operator that allows us to refer to the type of a variable, property, or expression at compile time. This operator returns the static type of the expression to which it is applied. It is very useful when we need to refer to an existing type instead of defining it explicitly.

```
let x = 10;
let y: typeof x; // y will be of type 'number'
```

In the above example, `typeof x` is evaluated as the type of the variable `x`, which is `number`. This allows us to assign `x`'s type to another variable `y` without having to specify it manually.

Utilization for Complex Types

One of the biggest advantages of `typeof` is its ability to handle complex types more clearly and concisely. For example, when working with types that are the result of complex unions or intersections, we can use `typeof` to capture those types efficiently.

```
interface Person {
  name: string;
  age: number;
}

type Employee = {
  id: number;
  position: string;
} & typeof myPerson; // Captures the type of 'myPerson'

const myPerson = { name: "John", age: 30 };

let employee: Employee;

employee = { id: 1, position: "Developer", name: "John", age: 30 };
```

In this example, `typeof myPerson` captures the type of `myPerson`, which is `{ name: string; age: number; }`. Then, this type is combined (`&`) with the additional properties of `Employee`. This allows us to define `Employee` in a way that directly leverages `myPerson`'s type without having to repeat its structure.

Benefits of `typeof`

- **Reflects Changes Automatically:** If we modify `myPerson`, the type of `Employee` will automatically adjust to reflect those changes.
- **Avoids Code Duplication:** We don't need to manually define the structure of `Person` twice; `typeof` ensures consistency.
- **Simplified Maintenance:** When `myPerson`'s type changes, uses of `typeof` are updated automatically, reducing errors and maintenance time.

Exploring `as const` in TypeScript

This operator can help us define immutable constant values, improving the security and accuracy of our code. Let's see how it works and how we can use it to get the most out of it.

What is `as const`?

The `as const` operator tells TypeScript to treat the value of an expression as a literal constant. This means that each value will be considered immutable, and its type will be reduced to its most specific possible form. This is particularly useful when we want to ensure that values do not change in the future.

Basic Example

Let's start with a simple example to see how `as const` works:

```
let colors = ["red", "green", "blue"] as const;
```

Without `as const`, `colors` would be of type `string[]`, allowing any string in the array. But by using `as const`, `colors` becomes a specific literal type: `readonly ["red", "green", "blue"]`. Now, TypeScript knows that `colors` contains exactly those three elements and nothing else.

Application to Objects

The use of `as const` is not limited to arrays; it can also be applied to objects. This is especially useful when working with configurations or data that should not change.

```
const configuration = {
  mode: "production",
  version: 1.2,
  options: {
    debug: false,
  },
} as const;
```

In this case, the `configuration` object has an immutable type with the exact values we have defined. This means that `configuration.mode` is of type `"production"`, `configuration.version` is of type `1.2`, and `configuration.options.debug` is of type `false`.

Benefits of `as const`

- **Immutability:** Values cannot be changed, preventing accidental errors.
- **Literal Types:** Types are reduced to their most specific forms, improving type precision.
- **Type Safety:** Ensures that values do not change at runtime, providing greater code security.

Use in Functions

Let's see how `as const` can improve accuracy in the context of functions:

```
function getConfig() {
  return {
    mode: "production",
    version: 1.2,
    options: {
      debug: false,
    },
  } as const;
}

const config = getConfig

();
// config.mode is "production", not string
// config.version is 1.2, not number
// config.options.debug is false, not boolean
```

Here, the `getConfig` function returns an object whose type is immutable thanks to `as const`. This ensures that the returned values have the most specific types possible.

Adventure in TypeScript: Type Assertion and Type Casting

Let's explore how to use them correctly, the precautions we should take, and the crucial difference between `unknown` and `any`. Let's dive in!

What is Type Assertion?

Type Assertion is a way to tell TypeScript to treat a variable as if it were of a specific type. It's like telling the compiler: "Trust me, I know what I'm doing." This can be useful in situations where we are sure of a variable's

type, but TypeScript cannot infer it correctly.

There are two main syntaxes for Type Assertion in TypeScript:

- **Using the as operator:**

```
let value: any = "This is a string";
let length: number = (value as string).length;
```

- **Using the <type> operator:**

```
let value: any = "This is a string";
let length: number = (<string>value).length;
```

Both syntaxes achieve the same result, but `as` is more commonly used in modern TypeScript code, especially when working with JSX in React.

Precautions with Type Assertion

Type Assertion is powerful, but it can also be dangerous if used incorrectly. Here are some things to keep in mind:

- **Confidence in the Type:** Ensure that the assertion is valid. If you get it wrong, you can introduce hard-to-detect runtime errors.

```
let value: any = "This is a string";
let number: number = value as number; // Runtime error!
```

- **Avoid Unnecessary Assertions:** Do not use Type Assertion if TypeScript can infer the type correctly.

```
let value = "This is a string"; // TypeScript infers 'value' as string
let length: number = value.length; // No need for Type Assertion
```

Type Casting in TypeScript

Type casting is similar to Type Assertion but often refers to converting one type to another at runtime, something more common in languages like C# or Java. In TypeScript, casting is generally achieved through conversion functions.

Example:

```
let value: any = "123";
let number: number = Number(value); // Casting from string to number
```

Although TypeScript is a superset of JavaScript, it does not add explicit casting features, relying instead on JavaScript conversion functions.

unknown vs any: Know the Difference

any and **unknown** are two special types in TypeScript that allow working with values of any type, but they have key differences in their use and safety.

- **any:**
  - Allows any value to be assigned to a variable.
  - Disables all type checks, which can lead to runtime errors.
  - Should be used sparingly.

```
let value: any = "This is a string";
value = 42; // No error, but may cause runtime issues
value.nonExistentMethod(); // No compile-time error, but will fail at runtime
```

- **unknown:**
  - Also allows any value to be assigned to a variable.
  - Forces type checks before accessing properties or methods, making the code safer.

```
let value: unknown = "This is a string";

if (typeof value === "string") {
  console.log(value.length); // Safe, TypeScript knows it's a string
}

// value.nonExistentMethod(); // Compile-time error
```

Combined Example

Let's see an example that combines Type Assertion, **any**, and **unknown**:

```
function processValue(value: unknown) {
  if (typeof value === "string") {
    let length = (value as string).length;
    console.log(`The length of the string is ${length}`);
  } else if (typeof value === "number") {
```

```
    let double = (value as number) * 2;
    console.log(`The double of the number is ${double}`);
  } else {
    console.log("The value is neither a string nor a number");
  }
}

let anyValue: any = "Text";
processValue(anyValue);

anyValue = 100;
processValue(anyValue);

anyValue = true;
processValue(anyValue); // The value is neither a string nor a number
```

In this example, we use **unknown** to receive values of any type, then check their type before performing specific operations. We also show how **any** can be flexible but should be handled carefully to avoid errors.

Functional Overloading in TypeScript: Pure Magic

Hello, devs! Today we are going to delve into the fascinating world of functional overloading in TypeScript. Let's explore how we can define multiple signatures for a function and how to use types so that our functions change their output based on the input parameter type. Let's get started!

What is Functional Overloading?

In TypeScript, functional overloading allows us to define multiple signatures for a function so that it can accept different types of arguments and behave differently based on the input type.

This is particularly useful when we have a function that can operate in different ways depending on the parameters it receives.

Basic Syntax

The basic syntax for defining function overloading in TypeScript includes several function signatures followed by an implementation that covers all cases.

```
function myFunction(param: string): string;
function myFunction(param: number): number;
function myFunction(param: boolean): boolean;

// Implementation that covers all overloads
function myFunction(
  param: string | number | boolean,
): string | number | boolean {
  if (typeof param === "string") {
```

```
    return `String received: ${param}`;
  } else if (typeof param === "number") {
    return param * 2;
  } else {
    return !param;
  }
}

// Using the overloaded function
console.log(myFunction("Hello")); // String received: Hello
console.log(myFunction(42)); // 84
console.log(myFunction(true)); // false
```

In this example, `myFunction` can accept a `string`, `number`, or `boolean` and will behave differently based on the argument type.

Practical Examples

- **Function to manipulate arrays and strings:**

```
function manipulate(data: string): string[];
function manipulate(data: string[]): string;
function manipulate(data: string | string[]): string | string[] {
  if (typeof data === "string") {
    return data.split("");
  } else {
    return data.join("");
  }
}

// Using the overloaded function
console.log(manipulate("Hello")); // ['H', 'e', 'l', 'l', 'o']
console.log(manipulate(["H", "e", "l", "l", "o"])); // "Hello"
```

- **Function to handle different types of inputs and produce different outputs:**

```
function calculate(input: number): number;
function calculate(input: string): string;
function calculate(input: number | string): number | string {
  if (typeof input === "number") {
    return input * input;
  } else {
    return input.toUpperCase();
  }
}

// Using the overloaded function
console.log(calculate(5)); // 25
console.log(calculate("hello")); // "HELLO"
```

Important Considerations

- **Unified Implementation:** The function implementation must be able to handle all the types of parameters defined in the overload signatures.
- **Compatible Return Type:** The return type must be compatible with all the types defined in the overload signatures.
- **Proper Use of Type Guards:** It is essential to use type guards (`typeof`, `instanceof`) correctly to ensure that the implementation handles each type appropriately.

Case with Complex Types

Function Overloading with Complex Types

First, we define our interfaces for the complex types `Cat` and `Dog`, which extend a base interface `Animal`.

```
interface Animal {
  type: string;
  sound(): void;
}

interface Cat extends Animal {
  type: "cat";
  breed: string;
}

interface Dog extends Animal {
  type: "dog";
  color: string;
}
```

Next, we define the overload declarations for the `processAnimal` function to specify the input types and the output types.

```
function processAnimal(animal: Cat): string;
function processAnimal(animal: Dog): number;
```

Then, we implement the `processAnimal` function using function overloading. Depending on whether the parameter is a `Cat` or a `Dog`, the function will return a `string` or a `number`, respectively.

```
function processAnimal(animal: Cat | Dog): string | number {
  if ("breed" in animal) {
    // The object is a cat
```

```
    console.log(`It's a ${animal.breed} cat`);
    animal.sound();
    return animal.breed;
  } else {
    // The object is a dog
    console.log(`It's a ${animal.color} dog`);
    animal.sound();
    return animal.color.length;
  }
}
```

Implementing the Types

We create instances of `Cat` and `Dog` and use the `processAnimal` function to process these objects. Depending on the type of object, the function will return a `string` or a `number`.

```
const myCat

: Cat = {
  type: "cat",
  breed: "Siamese",
  sound: () => console.log("Meow"),
};
const myDog: Dog = {
  type: "dog",
  color: "Black",
  sound: () => console.log("Woof"),
};

const catResult = processAnimal(myCat); // Output: It's a Siamese cat \n Meow
const dogResult = processAnimal(myDog); // Output: It's a Black dog \n Woof

console.log(catResult); // Output: Siamese
console.log(dogResult); // Output: 5
```

In this example, `processAnimal(myCat)` will return the cat's breed as a `string`, while `processAnimal(myDog)` will return the length of the dog's color as a `number`.

Additional Example: Overloading with Property Checks

Now, let's see another example using function overloading and property checks with the `in` operator.

```
interface Vehicle {
  type: string;
  maxSpeed(): void;
}

interface Car extends Vehicle {
```

```
  type: "car";
  brand: string;
}

interface Bicycle extends Vehicle {
  type: "bicycle";
  brakeType: string;
}

function describeVehicle(vehicle: Car): string;
function describeVehicle(vehicle: Bicycle): boolean;

function describeVehicle(vehicle: Car | Bicycle): string | boolean {
  if ("brand" in vehicle) {
    // The object is a car
    console.log(`It's a ${vehicle.brand} car`);
    vehicle.maxSpeed();
    return vehicle.brand;
  } else {
    // The object is a bicycle
    console.log(`It's a bicycle with ${vehicle.brakeType} brakes`);
    vehicle.maxSpeed();
    return vehicle.brakeType.length > 5;
  }
}

const myCar: Car = {
  type: "car",
  brand: "Toyota",
  maxSpeed: () => console.log("200 km/h"),
};
const myBicycle: Bicycle = {
  type: "bicycle",
  brakeType: "disc",
  maxSpeed: () => console.log("30 km/h"),
};

const carResult = describeVehicle(myCar); // Output: It's a Toyota car \n 200 km/h
const bicycleResult = describeVehicle(myBicycle); // Output: It's a bicycle with disc brakes \n 30 km/h

console.log(carResult); // Output: Toyota
console.log(bicycleResult); // Output: false
```

In this example, `describeVehicle(myCar)` will return the car's brand as a `string`, while `describeVehicle(myBicycle)` will return a `boolean` indicating whether the length of the bicycle's brake type is greater than 5 characters.

TypeScript Utilities: Essential Helpers

TypeScript offers a variety of utility types that facilitate the manipulation and management of complex types. These helpers allow transforming, filtering, and creating new types based on existing types. Below, we will explore some of the most common helpers and how they can be used in daily development.

Partial

`Partial<T>` converts all the properties of a type `T` into optional properties. It is useful when we want to work with incomplete versions of a type.

```
interface User {
  name: string;
  age: number;
  email: string;
}

const partialUser: Partial<User> = {
  name: "John",
};
```

Required

`Required<T>` converts all the properties of a type `T` into required properties. It is the opposite of `Partial`.

```
interface Configuration {
  darkMode?: boolean;
  notifications?: boolean;
}

const completeConfig: Required<Configuration> = {
  darkMode: true,
  notifications: true,
};
```

Readonly

`Readonly<T>` converts all the properties of a type `T` into read-only properties.

```
interface Book {
  title: string;
  author: string;
}

const book: Readonly<Book> = {
  title: "1984",
  author: "George Orwell",
};
```

```
// book.title = 'Animal Farm'; // Error: cannot assign to 'title' because it is a read-only property
```

Record

`Record<K, T>` constructs a type of object whose properties are keys of type **`K`** and values of type **`T`**.

```
type Role = "admin" | "user" | "guest";

const permissions: Record<Role, string[]> = {
  admin: ["read", "write", "delete"],
  user: ["read", "write"],
  guest: ["read"],
};
```

Pick

`Pick<T, K>` creates a type by selecting a subset of the properties **`K`** from a type **`T`**.

```
interface Person {
  name: string;
  age: number;
  address: string;
}

const personNameAge: Pick<Person, "name" | "age"> = {
  name: "Maria",
  age: 30,
};
```

Omit

`Omit<T, K>` creates a type by omitting a subset of the properties **`K`** from a type **`T`**.

```
interface Product {
  id: number;
  name: string;
  price: number;
}

const productWithoutId: Omit<Product, "id"> = {
  name: "Laptop",
  price: 1500,
};
```

Exclude

`Exclude<T, U>` excludes from `T` the types that are assignable to `U`.

```
type NumbersOrString = string | number | boolean;

type OnlyNumbersOrString = Exclude<NumbersOrString, boolean>; // string | number
```

Extract

`Extract<T, U>` extracts from `T` the types that are assignable to `U`.

```
type Types = string | number | boolean;

type OnlyBooleans = Extract<Types, boolean>; // boolean
```

NonNullable

`NonNullable<T>` removes `null` and `undefined` from a type `T`.

```
type PossiblyNull = string | number | null | undefined;

type NoNulls = NonNullable<PossiblyNull>; // string | number
```

ReturnType

`ReturnType<T>` obtains the return type of a function `T`.

```
function getUser(id: number) {
  return { id, name: "John" };
}

type User = ReturnType<typeof getUser>; // { id: number, name: string }
```

Complete Example

Let's see a practical example using several of these helpers together:

```
interface User {
  id: number;
  name: string;
  email?: string;
  address?: string;
}

// Convert all properties to optional
```

```typescript
type PartialUser = Partial<User>;

// Convert all properties to required
type RequiredUser = Required<User>;

// Create a read-only type
type ReadonlyUser = Readonly<User>;

// Select only some properties
type BasicUser = Pick<User, "id" | "name">;

// Omit some properties
type UserWithoutId = Omit<User, "id">;

// Create a record of roles to permissions
type Role = "admin" | "editor" | "reader";
const permissions: Record<Role, string[]> = {
  admin: ["create", "read", "update", "delete"],
  editor: ["create", "read", "update"],
  reader: ["read"],
};

// Exclude types
type ID = string | number | boolean;
type IDWithoutBooleans = Exclude<ID, boolean>; // string | number

// Extract types
type OnlyBooleans = Extract<ID, boolean>; // boolean

// Remove null and undefined
type MayBeNull = string | null | undefined;
type NotNull = NonNullable<MayBeNull>; // string
```

Generics in TypeScript

Generics in TypeScript are a powerful tool that allows creating reusable and highly flexible components. Generics provide a way to define types in a way that is not yet determined, allowing functions, classes, and types to work with any type specified at the time of the call or instantiation. Below, we will explore the basics and advanced concepts of generics in TypeScript, including practical examples.

Basic Concepts

Generics are declared using the angle notation <T>, where T is a generic type parameter. This type parameter can be any letter or word, although T is commonly used.

Generic Functions

Generic functions allow working with any type of data without sacrificing typing.

```
function identity<T>(value: T): T {
  return value;
}

const numberValue = identity<number>(42); // 42
const textValue = identity<string>("Hello World"); // 'Hello World'
```

Generic Classes

Generic classes allow creating data structures that can work with any type.

```
class Box<T> {
  content: T;

  constructor(content: T) {
    this.content = content;
  }

  getContent(): T {
    return this.content;
  }
}

const numberBox = new Box<number>(123);
console.log(numberBox.getContent()); // 123

const textBox = new Box<string>("Text");
console.log(textBox.getContent()); // 'Text'
```

Generic Interfaces

Generic interfaces allow defining contracts that can adapt to different types.

```
interface Pair<K, V> {
  key: K;
  value: V;
}

const numberTextPair: Pair<number, string> = { key: 1, value: "One" };
const textBooleanPair: Pair<string, boolean> = { key: "active", value: true };
```

Advanced Use of Generics

Generic Constraints

We can restrict the types that a generic can accept using `extends`.

```
interface WithName {
  name: string;
}

function greet<T extends WithName

>(obj: T): void {
  console.log(`Hello, ${obj.name}`);
}

greet({ name: "John" }); // Hello, John
// greet({ lastName: 'Doe' }); // Error: object does not have 'name' property
```

Generics in Higher-Order Functions

We can use generics in functions that accept and return other functions.

```
function process<T>(items: T[], callback: (item: T) => void): void {
  items.forEach(callback);
}

process<number>([1, 2, 3], (number) => console.log(number * 2)); // 2, 4, 6
```

Complete Example with Complex Types and `in`

Next, we will combine what we have learned about generics with an advanced type check using the `in` keyword.

```
interface Animal {
  type: string;
  sound(): void;
}

interface Cat extends Animal {
  type: "cat";
  breed: string;
}

interface Dog extends Animal {
  type: "dog";
  color: string;
}

function processAnimal<T extends Animal>(animal: T): string {
  if ("breed" in animal) {
```

```
    return `It's a ${animal.breed} cat`;
  } else if ("color" in animal) {
    return `It's a ${animal.color} dog`;
  } else {
    return `It's an animal of unknown type`;
  }
}

const myCat: Cat = {
  type: "cat",
  breed: "Siamese",
  sound: () => console.log("Meow"),
};
const myDog: Dog = {
  type: "dog",
  color: "Black",
  sound: () => console.log("Woof"),
};

console.log(processAnimal(myCat)); // Output: It's a Siamese cat
console.log(processAnimal(myDog)); // Output: It's a Black dog
```

In this example, `processAnimal` is a generic function that can process any type of animal that extends from `Animal`. We use the `in` keyword to check for the existence of a property and thus determine the exact type of the object.

The Magic of Enums

First, we define two enums. Enums are those friends who always bring something useful to the party. They allow us to group named constants so that we don't have to guess what each value means.

```
enum Numbers1 {
  "NUMBER1" = "number1",
  "NUMBER2" = "number2",
}

enum Numbers2 {
  "NUMBER3" = "number3",
}
```

Combining Superpowers

Now, we mix these two enums into a single object. For this, we use the spread operator (`...`). And note, as `const` is the key here to make TypeScript treat this object as an immutable constant.

```
const myNumbers = { ...Numbers1, ...Numbers2 } as const;
const mixValues = Object.values(myNumbers);
```

Derived Types from Combined Enums

And now what? Well, now we use `typeof` and `[number]` to create a type that represents the combined values of the enums. How is this?

```
type MixNumbers = (typeof mixValues)[number];
```

But, why `[number]`?

Good question, dear reader. When we do `Object.values(myNumbers)`, we get an array of values. So, `typeof mixValues` gives us the type of this array, which is `string[]` in our case. By using `[number]`, we are saying "I want the type of the elements inside this array". It's like telling TypeScript: "Hey, give me the type of what's inside, not the container."

Now, the more technical and precise reason: enums in TypeScript generate an internal structure that uses both keys and values to create a kind of bi-directionality. This means that in the enum object, each value has an automatically associated numeric key. When we use `[number]`, we are leveraging this feature to get the type of the values being indexed numerically.

Creating a Type Based on Our Values

Finally, we create a type `Enums` that uses a mapped index to define properties based on the values of `MixNumbers`. Each property can be of any type (`any`), because sometimes life is just that flexible.

```
type Enums = {
  [key in MixNumbers]: any;
};
```

The Complete Example

Let's see the full code in action:

```
enum Numbers1 {
  "NUMBER1" = "number1",
  "NUMBER2" = "number2",
}
```

```
enum Numbers2 {
  "NUMBER3" = "number3",
}

const myNumbers = { ...Numbers1, ...Numbers2 } as const;
const mixValues = Object.values(myNumbers);

type MixNumbers = (typeof mixValues)[number];

type Enums = {
  [key in MixNumbers]: any;
};

// Usage example
const example: Enums = {
  number1: "This is number 1",
  number2: 42,
  number3: { detail: "Number 3 as an object" },
};

console.log(example);
```

Code Breakdown

- **Enum Definition**: `Numbers1` and `Numbers2` are our initial superheroes, each with their own powers.
- **Enum Combination**: We mix our heroes' powers into a single team using `...` and `as const`.
- **Creating Derived Types**: We use `typeof` and `[number]` to create a type that represents the combined values.
- **Defining the Enums Type**: We use a mapped index to define properties based on `MixNumbers`.