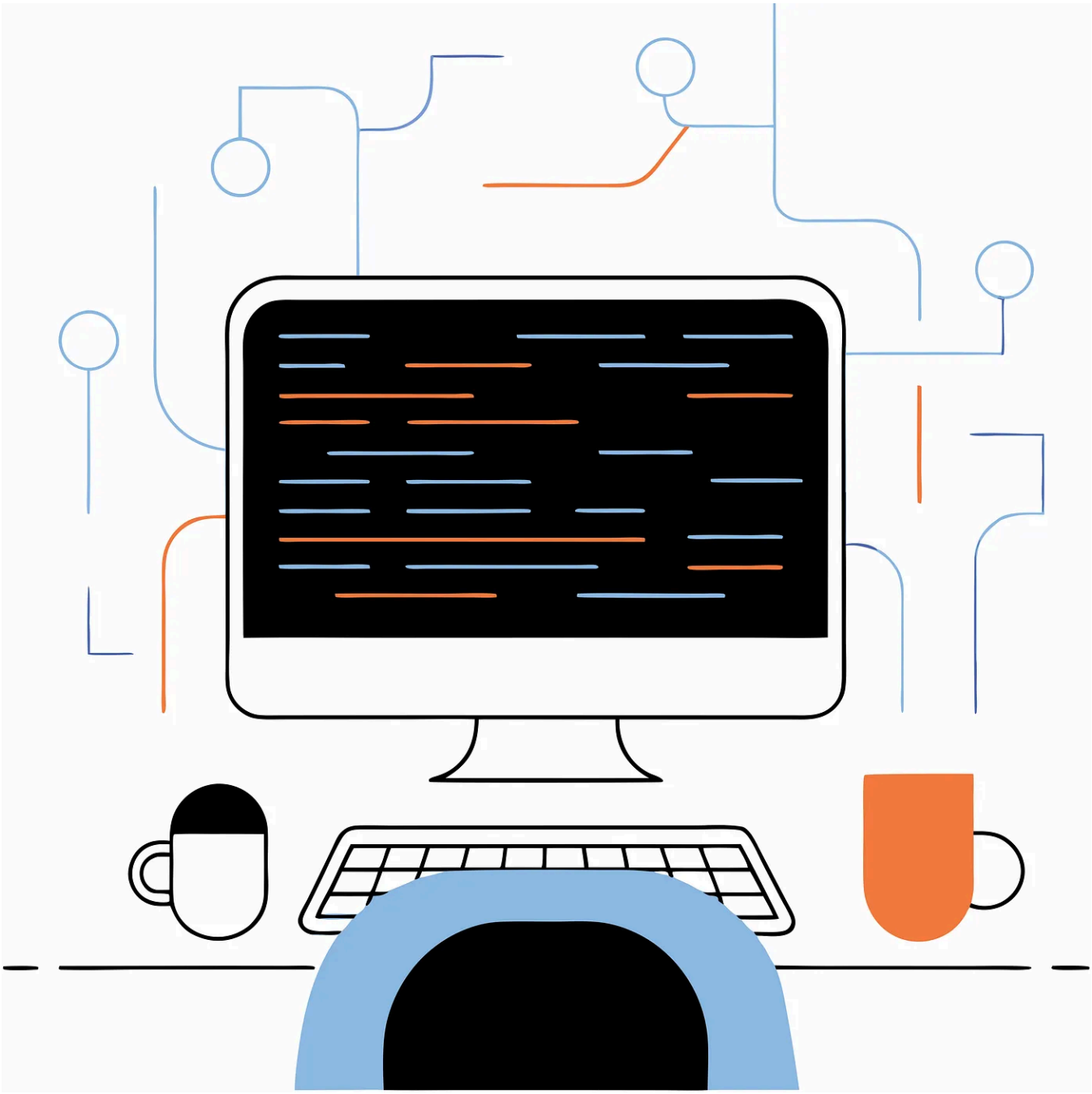
Rust is easy? Go is... hard?

Bryan Hyland · Follow

7 min read · Apr 13, 2025



I recently had a project pop up where I thought Go would be a really good fit. It's been a while since I actually have used Go for anything, so I also thought it'd be a fantastic opportunity to practice and not lose those skills. It turns out, Rust is actually easier to use than Go. Here is the story.

What makes Rust easier than Go?

Before I get into this part, you have to be aware that I have used Rust as exclusively as I can for a while now. I'm over the hump where myself and the borrow checker are no longer mortal enemies, and although not friends, we understand each other fairly well. Needless to say, Rust is not "easy" by any means in the beginning.

Now, that the disclaimer is out of the way, let me go into the features that make me say that Rust is actually easier than Go.

Enums

Enums are something that I've become very accustomed to using, whether it's in Rust, C#, Java, or C++. They are a pretty pivotal part of a language, but Go doesn't have a proper enum type. The Go developer community have had to come up with a workaround for this.

Go's `iota` Workaround

Go developers often use `iota` to simulate enums:

```
type Animal int

const(
    Dog Animal = iota // 0
    Cat               // 1
    Horse             // 2
```

While `iota` generates sequential integers automatically, it still lacks Rust's core advantages:

- **No Associated Data:** `iota` enums are just integers — no strings, structs, or other values.
- **Immutable Values:** Constant's can't be modified at runtime.
- **No Type Safety:** Accidentally passing `Animal(42)` won't trigger compile-time errors.

Rust's Enum Superpower

Compare this to Rust's enums, which bundle data and behavior.

```
enum Animal {
    Dog(String),
    Cat(String),
    Horse(String),
}

fn make_animal_sound(animal: &Animal) {
    match animal {
      Animal::Dog(sound) => println!("The dog says: {sound}!"),
      Animal::Cat(sound) => println!("The cat says: {sound}!"),
      Animal::Horse(sound) => println!("The horse says: {sound}!"),
```

```
    }
}

fn main() {
    let dog = Animal::Dog("Woof".to_string());
    let cat = Animal::Cat("Meow".to_string());
    let horse = Animal::Horse("Neigh".to_string());

    make_animal_sound(&dog);
    make_animal_sound(&cat);
    make_animal_sound(&horse);
}

/*
    Output:
        The dog says: Woof!
        The cat says: Meow!
        The horse says Neigh!
*/
```

- **Key Advantage:** Rust enums are mutable in the sense you can pass them different data at runtime and have that data handled using pattern matching. The data is also "bundled" in a sense that I don't need an extra struct to handle multiple forms of related application states.

Two examples that use this key Rust advantage are the GUI frameworks *Iced* and *Libcosmic*; they use it extensively.

Go doesn't have the ability to do this. I will concede that the workaround doesn't need to use `iota`, it can be used with other types. However, the disadvantages stated above do not change. Here is the Rust example above re-written in Go:

```
type Animal string

const (
  Dog Animal = "Woof!"
  Cat Animal = "Meow!"
  Horse Animal = "Neigh!"
)

func makeAnimalSound(animal Animal) {
    switch animal {
        case Dog:
          fmt.Printf("The dog says %s!\n", animal)
        case Cat:
```

```
            fmt.Printf("The cat says %s!\n", animal)
        case Horse:
            fmt.Printf("The horse says %s!\n", animal)
    }
}

func main() {

    makeAnimalSound(Dog)
    makeAnimalSound(Cat)
    makeAnimalSound(Horse)
}

/*
    Output:
        The dog says Woof!
        The cat says Meow!
        The horse says Neigh!
*/
```

I could never change “Woof!” to “Bark!” at runtime if my application needed it to. This is where I hit this limitation in the Go project I was working on. I needed to change the “*enum*” value based on certain factors at runtime. Since these values are constants, I couldn’t, and I had to find another way. This made it a more complex code base when it should’ve been a straight forward approach. At this point, I had a thought — “What if I use something like a trait?”

Traits vs. Interfaces

Rust’s traits let you define shared behavior *with* or *without* **default implementations,** reducing boilerplate code.

Example: Reusable K9 Trait

```
trait K9 {
    // No default implementation
    fn set_breed(&mut self, breed: Breed);

    // Default implementations
    fn make_sound(&self) {
        println!("Bark!");
    }

    fn print_breed(&self) {
        println!("I'm a K9 of unknown breed!");
```

```
    }
}

struct Dog {
    name: String,
    breed: Breed,
}

// Implement set_breed and override print_breed
// Use default of make_sound.
impl K9 for Dog {
    fn set_breed(&mut self, breed: Breed) {
        self.breed = breed;
    }

    fn print_breed(&self) {
        println!("I'm a {}!", self.breed);
    }
}

struct Wolf {
    breed: Breed,
}

impl K9 for Wolf {
    fn set_breed(&mut self, breed: Breed) {
        self.breed = breed;
    }

    // Override default make_sound
    fn make_sound(&self) {
        println!("Howl!");
    }
}
```

Go has something similar — interfaces. Here is the same code re-written in Go using it's interface feature.

```
package main

import "fmt"

// Define the K9 interface
type K9 interface {
    setBreed(breed string)
    makeSound()
    printBreed()
```

```
}

// Dog struct implementing K9
type Dog struct {
    name string
    breed string
}

func (d *Dog) setBreed(breed string) {
    d.breed = breed
}

func (d Dog) makeSound() {
    fmt.Println("Bark!")
}

func (d Dog) printBreed() {
    fmt.Printf("I am a %s and my name is %!\n", d.breed, d.name)
}

// Wolf struct implementing K9
type Wolf struct {
    breed string
}

func (w *Wolf) setBreed(breed string) {
    w.breed = breed
}

func (w Wolf) makeSound() {
    fmt.Println("Howl!")
}

func (w Wolf) printBreed() {
    fmt.Printf("I am a %s and I love to ", w.breed)
    w.makeSound()
}

// Main function
func main() {
    arya := Dog{name: "Arya", breed: “Golden Retriever”}
    arya.setBreed(“Black Lab”)
    arya.printBreed()
    arya.makeSound()

    fmt.Println()

    redWolf := Wolf(breed: “Red Wolf”)
    redWolf.printBreed()
```

And this is where I ran into the tediousness and bloat caused by Go's interfaces. Unlike Rust's traits, Go interfaces lack the concept of default implementations. This means that even if multiple structs need to perform the exact same logic, they can't share it directly. Instead, each struct must re-implement the logic separately, even if it's identical across types. While the method signatures are reusable, the actual implementation is not — resulting in repetitive code that quickly becomes tedious to maintain.

There are workarounds for this limitation, but often they add complexity and aren't as seamless or intuitive as Rust's traits. Rust traits allow developers to define shared behavior once and reuse it across multiple types, while still giving the flexibility to override default implementations when needed. This makes Rust traits far more ergonomic and efficient for scenarios requiring shared functionality.

Error Handling

One of the biggest frustrations I had to remember while working in Go on this project was its approach to error handling. In Go, errors are handled using an explicit `error` type, which requires constant checks throughout your code. The typical pattern looks like this:

```
file, err := os.Open("data.txt")
if err != nil {
    return err
}

defer file.Close()

// logic
```

While functional, this approach quickly becomes repetitive as you find yourself typing `if err != nil` every few lines. This verbosity can lead to bloated code that's harder to maintain.

Rust on the other hand, offers a far more flexible and ergonomic approach to error handling. With tools like `unwrap`, `unwrap_or`, `unwrap_or_else`, `unwrap_or_default`, `expect`, `Option`, and `Result`, developers have a variety of ways to handle errors based on their specific needs. Rust's `?` operator allows for automatic error propagation when working with functions that return a `Result`. Pair these features

with pattern matching, and you have a powerful toolkit for catching and handling any kind of error; without compromising readability.

Here's an example of a small sample of the different methods that can be used for error handling in Rust:

```
use std::fs::{File, read_to_string};
use std::io::{Error, Write};

fn read_file() -> Result<String, Error> {
    // Automatic error propogation if there is an error
    // using the ? operator.
    let mut file = File::create_new("data.txt")?;
    file.write_all("Example!".as_bytes())?;

    // Manual error handling using Result (Ok, Err)
    Ok( match read_to_string("data.txt") {
            Ok(content) => content,
            Err(e) => { return Err(e); }
        }
    )
}

fn main() {
    // Error handling that allows a panic if there's an error.
    // The message in the expect method is printed to the console
    // if there is an error.
    println!("{}", read_file()
        .expect("There should've been a file and content!")
    );
}
```

Key Advantages of Rust's Error Handling:

- **Automatic Propagation:** The `?` operator eliminates repetitive checks by propagating errors directly to the caller.
- **Versatile Tools:** Methods like `unwrap_or_else` allow developers to provide fallback logic seamlessly.
- **Pattern Matching:** Errors can be matched against specific cases for granular control.

- **Encourages Best Practices**: Rust's compiler ensures that errors are handled properly, reducing the risk of unhandled exceptions.

By contrast, Go's error handling often feels tedious and prone to boilerplate code. While workarounds exist (e.g., helper functions for common patterns), they add complexity rather than solve core issues.

Go Requires Workarounds

This Go project reminded me that workarounds are part of the design of the language. Go forces you to work around its limitations (e.g., using maps for enum data, rewriting interface methods). Rust's traits, enums, and error handling are **first-class language features**; no hacks needed. Rustaceans actually have it pretty easy, especially when we're working in pure Rust and don't have to cross the streams.

Final Thoughts

- **Go's Simplicity Fades**: Initial ease gives way to repetitive error checking, boilerplate, and workarounds.
- **Rust's Investment Pays Off**: Steeper learning curve unlocks the ability to have long-term productivity.
- **Conclusion**: For complex, maintainable systems, Rust's "hard" upfront cost saves you from Go's "hard" long-term grind.

Rust | Rust Programming Language | Go | Golang | Programming

Follow

Written by Bryan Hyland

25 Followers · 39 Following

Dedicated to creating safe and efficient software solutions. Software developer with a strong foundation in Cybersecurity principals.

Responses (20)

AdTech

What are your thoughts?

Gastón Haro
5 days ago

If ypu want to modify an Enum then I believe your coding practices are very wrong. Secondly why compare Go's interfaces to Traits when they are very different concepts? Just say that Go does no have traits period. Are those all the arguments you got to? Very poor article.

30 Reply

Walker O'Brien
4 days ago

These “workarounds” you speak of aren’t actually workarounds…. It’s you trying to make go act like rust. Go is not like rust on purpose and that’s okay! If you like those features then use rust but stop bashing on a great language because it made… more

6 Reply

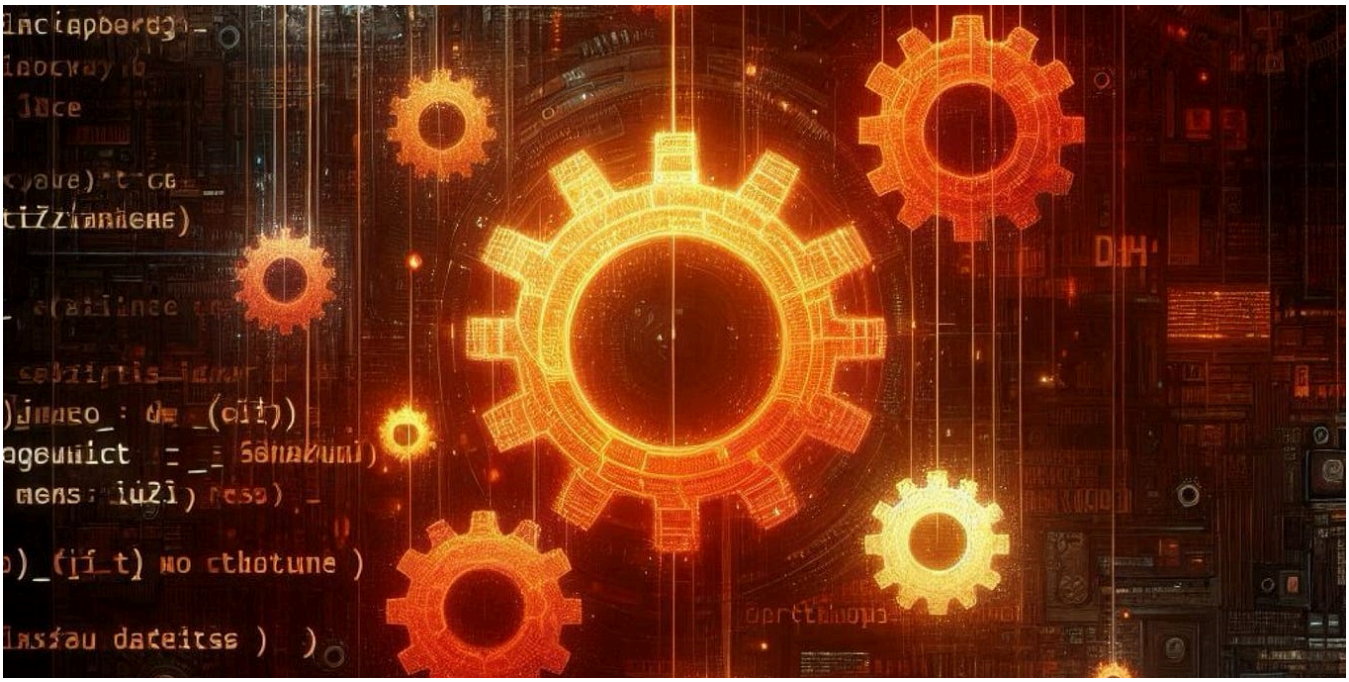
Cosku Bas
6 days ago (edited)

You missed that Go has embedding: https://gobyexample.com/struct-embedding for what you call "default" implementation

19 1 reply Reply

See all responses

More from Bryan Hyland



Bryan Hyland

Shadowing in Rust: What is it?

Let’s get out of the dark!

Mar 23 2

Bryan Hyland

Thoughts on Languages

Originally posted on my website, bhh32.com, on 20Oct24. Link to the original post here.

Mar 23 5

Bryan Hyland

Learn Go Not Java

Originally posted on my website, bhh32.com on 8Nov24. Original post can be found here.

Mar 23

Bryan Hyland

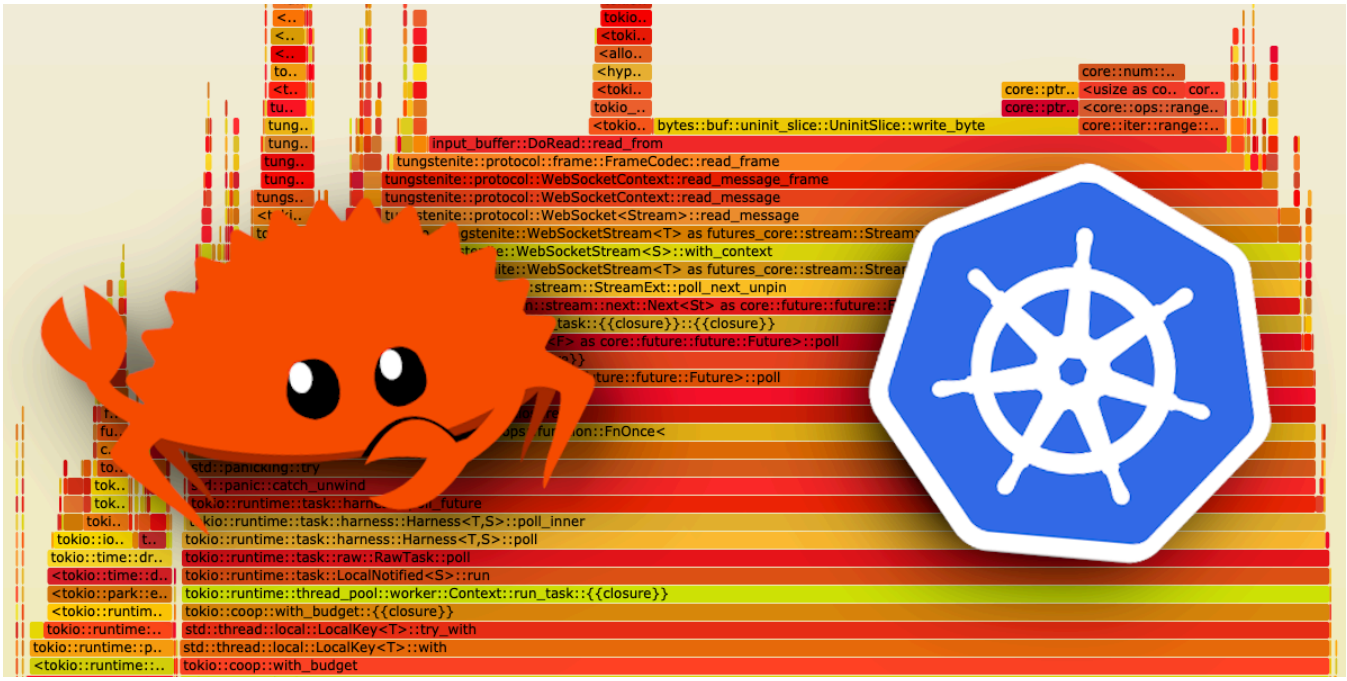
Rust Copy Types—Amazing and Confusing

Originally posted on my website, bhh32.com, on 11Mar25. You can view the original here.

Apr 7

See all from Bryan Hyland

Recommended from Medium



Harsh Vardhan

Rust Killed My Go Microservice (and It Deserved It)

I rewrote a production Go service in Rust. It was painful. It was slow. It was worth every line of code.

5d ago 261 15

Apr 13 2.5K 41

In Pythonic AF by Aysha R

I Tried Writing Python in VS Code and PyCharm—Here's What I Found

One felt like a smart coding companion. The other felt like assembling IKEA furniture blindfolded.

5d ago 652 54

David Lee

A Memory Leak Story of Map(Go vs. Rust)

Go's map implementation never shrinks. Ever.

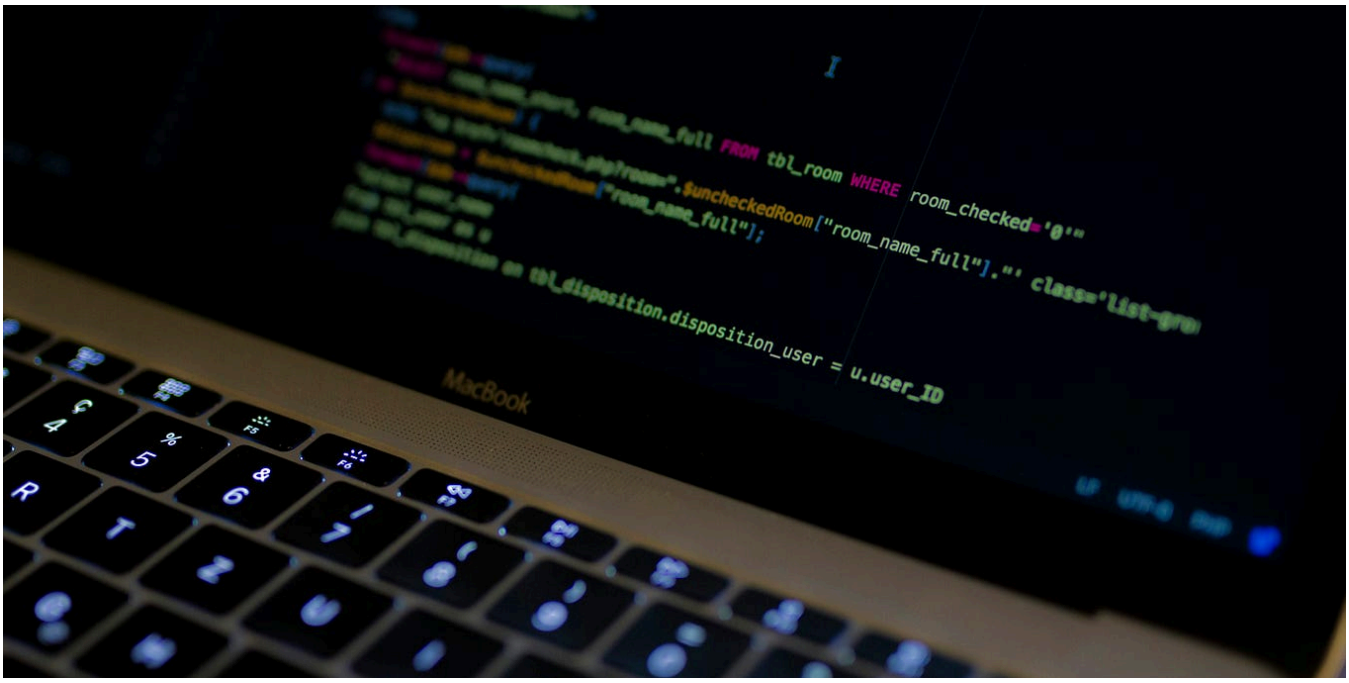
Apr 13 300 19

In Code Like A Girl by Aleena

How Assembly Programming Saved My Soul

Seriously, it Really Did!

6d ago 891 23



In Level Up Coding by Renaldi Purwanto

How I Built Safe and Injection-Free Database Queries in Go

Safe SQL Query Patterns in Go: Protecting Against Injection Vulnerabilities

5d ago 29