Maurício Linhares
Posted on Feb 8, 2022

❤️ 62 🦄 9

GoF Design patterns that still make sense in Go

#architecture #programming #go #tutorial

Since its release in 1994, the [Design Patterns book]() continues to be a seminal work in building software. The book created a new shared vocabulary and named these repeated solutions we see all over different codebases. So much so there have been multiple other books on design patterns, documenting even more examples. You can quickly explain that your solution is an Adapter for anyone who has read about it without detailing what an Adapter is.

The material, as expected, isn't without critique, [Peter Novig]() wrote his analysis on how many of these patterns are unnecessary or replaceable by simpler constructs in dynamic languages. The book was written mainly using C++ (and some Smalltalk), focusing on covering what was available in C++, so dynamic languages change or remove the need for some of these patterns.

A couple of weeks ago, a Twitter thread caught my attention, saying people shouldn't read this book anymore as many of the patterns are old or don't make much sense in most mainstream programming languages. I still have fond memories of when I read it back in college (2003, many moons ago) and wondered if it is true? Have languages evolved past this book, or do we still use the patterns and vocabulary defined there?

So, coming from a Golang perspective, what are some of the patterns in the book that are still usable nowadays? Let's have a look!

Builder

Builders are very much alive. They might have fancy names like [functional options]() or [fluent interfaces]() but the goal is still the same of the honored builder, simplify the creation of a complex object so that you don't end up with a single function call receiving dozens of parameters.

We'll now look at how you could set up a builder that produces *http.Request objects:

```go
package gof_go

import (
   "context"
   "io"
   "net/http"
)

// NewBuilder creates a builder given a URL, we're going to use this so we don'
// the actual builder and have to worry about null/empty values on the builder
// You could just use a struct directly here but it makes it a bit harder to va
// defaults so we'll go for the simpler interface based solution.
func NewBuilder(url string) HTTPBuilder {
   return &builder{
      headers: map[string][]string{},
      url:     url,
      body:    nil,
      method:  http.MethodGet,
      ctx:     context.Background(),
      close:   false,
   }
}

// HTTPBuilder defines the fields we want to set on this builder, you could add
// fields here.
type HTTPBuilder interface {
   AddHeader(name, value string) HTTPBuilder
   Body(r io.Reader) HTTPBuilder
   Method(method string) HTTPBuilder
   Close(close bool) HTTPBuilder
   Build() (*http.Request, error)
}

type builder struct {
   headers map[string][]string
   url     string
   method  string
   body    io.Reader
   close   bool
   ctx     context.Context
}

func (b *builder) Close(close bool) HTTPBuilder {
   b.close = close

   return b
}

func (b *builder) Method(method string) HTTPBuilder {
   b.method = method

   return b
}

func (b *builder) AddHeader(name, value string) HTTPBuilder {
   values, found := b.headers[name]

   if !found {
      values = make([]string, 0, 10)
   }

   b.headers[name] = append(values, value)

   return b
}

func (b *builder) Body(r io.Reader) HTTPBuilder {
   b.body = r

   return b
}

func (b *builder) Build() (*http.Request, error) {
   r, err := http.NewRequestWithContext(b.ctx, b.method, b.url, b.body)
   if err != nil {
      return nil, err
   }

   for key, values := range b.headers {
      for _, value := range values {
         r.Header.Add(key, value)
      }
   }

   r.Close = b.close

   return r, nil
}
```

So we have a builder that sets up some sane defaults (the body is empty, the method is GET, there's a default context). Here's what it looks like in use:

```
func TestBuilder_Build(t *testing.T) {
   request, err := NewBuilder("https://example.com/").
      AddHeader("User-Agent", "Golang patterns").
      Build()

   assert.NoError(t, err)

   assert.Equal(t, "Golang patterns", request.Header.Get("User-Agent"))
   assert.Equal(t, http.MethodGet, request.Method)
   assert.Equal(t, "https://example.com/", request.URL.String())
}
```

We could go complex with multiple headers, change the method, set a custom context, and it would still look clean and easy to read. That's the main advantage of using a builder to instantiate complex objects. You get to go as deep as needed but should still be allowed to create something with sane defaults quickly. While this builder follows the *fluent API* style, it's not the only way to produce a builder, as long as you find a way to separate the parametrization from the creation of a complex object that would still be a builder (like using the function opts linked above). The main goal here is to make it easier to build complex objects correctly.

Abstract factory/factory method

As the language lacks inheritance and focuses on composition, the primary way you'd represent factories is by having a function that produces an object as a parameter instead of abstract classes or methods.

A common reason you'd do this is to simplify unit tests. Imagine I'm testing a type that opens a socket connection to some other service, while you could use a net.Dialer directly, which would make testing it harder. If I give the object a method that produces a `net.Conn` object instead, I can easily replace the implementation by changing the factory function.

Here's what it would look like:

```
package gof_go

import (
   "bytes"
   "context"
   "fmt"
   "github.com/stretchr/testify/assert"
   "github.com/stretchr/testify/require"
   "net"
   "testing"
   "time"
)

type mockAddress struct {
   network string
   address string
}

func (m *mockAddress) Network() string {
   return m.network
}

func (m *mockAddress) String() string {
   return m.address
}

// mockConnection represents a connection used for testing only
type mockConnection struct {
   closed  bool
   buffer  *bytes.Buffer
   address *mockAddress
}

func (m *mockConnection) Read(b []byte) (n int, err error) {
   return m.buffer.Read(b)
```

```
}

func (m *mockConnection) Write(b []byte) (n int, err error) {
   return m.buffer.Write(b)
}

func (m *mockConnection) LocalAddr() net.Addr {
   return m.address
}

func (m *mockConnection) RemoteAddr() net.Addr {
   return m.address
}

func (m *mockConnection) SetDeadline(t time.Time) error {
   return nil
}

func (m *mockConnection) SetReadDeadline(t time.Time) error {
   return nil
}

func (m *mockConnection) SetWriteDeadline(t time.Time) error {
   return nil
}

func (m *mockConnection) Close() error {
   m.closed = true
   return nil
}

type socketClient struct {
   address string
   factory func(ctx context.Context, network, address string) (net.Conn, error)
}

func (s *socketClient) ping(ctx context.Context) error {
   c, err := s.factory(ctx, "tcp4", s.address)
   if err != nil {
      return err
   }

   defer func() {
      if err := c.Close(); err != nil {
         fmt.Printf("failed to close socket: %v\n", err)
      }
   }()

   if _, err := c.Write([]byte("PING")); err != nil {
      return err
   }

   return nil
}

func TestSocketClient(t *testing.T) {
   connection := &mockConnection{
      buffer: bytes.NewBuffer(make([]byte, 0, 1024)),
	}

	c := &socketClient{
		address: "example.com:40",
		factory: func(ctx context.Context, network, address string) (net.Conn, er
			connection.address = &mockAddress{
				address: address,
				network: network,
			}

			return connection, nil
		},
	}

	require.NoError(t, c.ping(context.Background()))
	assert.True(t, connection.closed)
	assert.Equal(t, "PING", connection.buffer.String())
}
```

We need mocks for `net.Addr` and `net.Conn` as we want to return in memory structs for those. Then our `socketClient` type has a `factory` property with type `func(ctx context.Context, network, address string) (net.Conn, error)`, which is exactly the same function signature as net.Dialer#DialContext so the concrete implementation here would be the `net.Dialer` function.

Instead of thinking about factories as classes you use or extend, you can have them be a function you call that produces the object. There is no need for inheritance, abstract classes, or interfaces, just a function signature.

Adapter

It continues to be a widely used pattern all over, the database/sql package is an excellent example of the pattern. Every database driver implements the driver.Driver interface and registers it with the `database/sql` package. So it doesn't matter what database you're using, it all looks the same as you'll be interacting with objects from the `database/sql` package only most of the time.

We also see this same pattern on libraries like go-cloud where the specific details of cloud providers are hidden behind the standard interface the library presents.

Decorator

Decorators add functionality to an existing object you might have no control over without breaking its interface. You build a decorator by creating an object that wraps another but has the same methods and forwards calls of these methods to the object being *decorated*, adding some functionality on top of them.

Typical use cases are adding buffers to IO classes (which is the example we'll see in a bit), adding metrics or tracing to types you don't have control over, like external libraries, and many more. The main goal is hiding from the calling code that something has changed.

The lack of inheritance makes decorators in Go a bit more problematic, as you can see on [this ancient issue on collecting metrics when resolving DNS entries](). Given that the code that calls the `net.Resolver` code expects the exact type and there is no inheritance, we can't *wrap* the resolver as we would in languages that allow inheriting. You can only use decorators in Go if you're dealing with interfaces or function signatures. If you have to wrap a struct, the only way is to change the code that depends on the struct to an interface with the same methods.

Now we'll look at a decorator that adds a buffer to `io.Reader` objects (this is not supposed to be the most efficient and optimized version, it's just an example):

```
func NewBufferedReader(wrapped io.Reader, length int) io.Reader {
   if length <= 0 {
      length = 1024
   }

   return &bufferedReader{
      currentIndex:  0,
      lastIndex:     0,
      buffer:        make([]byte, length, length),
      wrappedReader: wrapped,
      err:           nil,
   }
}

type bufferedReader struct {
   currentIndex  int
   lastIndex     int
   buffer        []byte
   wrappedReader io.Reader
   err           error
}

func (b *bufferedReader) Read(p []byte) (n int, err error) {
   if len(p) == 0 {
      return 0, errors.New("an empty slice was provided to Read")
   }

   availableBytes := b.lastIndex - b.currentIndex

   if availableBytes == 0 {
      if b.err != nil {
         return 0, b.err
      }

      if read, err := b.wrappedReader.Read(b.buffer); err == nil || err == io.E
         b.err = err
         b.currentIndex = 0
         b.lastIndex = read
         availableBytes = read

         if availableBytes == 0 {
            return 0, b.err
         }
      } else {
         b.err = err
```

```
            return 0, err
        }
    }

    expectedBytes := len(p)

    bytesToRead := availableBytes
    if availableBytes > expectedBytes {
        bytesToRead = expectedBytes
    }

    copy(p, b.buffer[b.currentIndex:b.currentIndex+bytesToRead])
    b.currentIndex += bytesToRead

    return bytesToRead, b.err
}
```

We wrap any existing `io.Reader` and add a buffer on top of it. For the code that was using a reader before, nothing has changed, it is still an `io.Reader` object, but we have introduced new functionality without breaking the contract. It's also how the IO package works, with decorators adding features on top of reader and writer objects. This pattern is still alive in Go, albeit not in as many places as in other languages.

Facade

Still here! Facades hide a complex or extensive API behind a small interface you can interact with without understanding all the details behind the scenes. Like we mentioned before, go-cloud is an excellent example of both adapter and facade, as it hides the complex details of talking to cloud providers behind a straightforward interface.

Proxy

Proxies are an interface to some other object that might be expensive to create or interact with directly. A typical case for using proxies is in ORM tools when loading an association. Think a `User` has `Posts`. You don't necessarily need to load the posts for the user every time you're loading a user so that ORM tools will hide the `Posts` behind a proxy object. They will wait until you try to do something with `Posts` that requires looking at the objects. They'll now load them from the data source where they live, instantiating the `Posts` collection.

If you never try to access them, you won't pay for the cost of loading them, and that's the advantage of using a proxy. You don't have to have that object directly available to you all the time. You can apply these lazy load techniques to produce more efficient code and use fewer resources.

Like the Decorator example above, the lack of inheritance also forces proxies to only be available for interface or function types. Reflection in Go also doesn't offer dynamic proxies (creating a proxy object from scratch in runtime) as you'd find in languages like Java and C# or a `method_missing` capability as we have in Ruby, you have to generate the proxy code at compilation time.

Chain of responsibility and Command

Two patterns together? Yes!

Almost every single case of Chain of responsibility I've ever seen has been with Command. I'm not even sure if it makes sense to implement a chain of responsibility without Command objects (or functions). If you've done any HTTP server development in go you've seen them together already:

```go
package http

type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

This is our Command for handling HTTP requests. It doesn't matter how you're going to implement your functionality. As long as it matches this interface, it's going to work. We also have the same interface defined as a function signature:

```go
package http

type HandlerFunc func(ResponseWriter, *Request)
```

Both should be interchangeable. The only thing that matters is that they have the same inputs and outputs. Both take an `http.ResponseWriter` and a `http.Request` and have no return types.

Good, we have the Command. Where does Chain of Responsibility come from then? Middlewares!

Here's the middleware interface:

```go
package gof_go

import (
   "fmt"
   "net/http"
)

type HTTPMiddleware func(w http.ResponseWriter, r *http.Request, next http.Hand

func LoggingMiddleware(w http.ResponseWriter, r *http.Request, next http.Handle
   fmt.Printf("REQUEST %v\n", r.URL.String())
   next.ServeHTTP(w, r)
}

func OkHandler(w http.ResponseWriter, r *http.Request) {
   w.WriteHeader(http.StatusOK)
   if _, err := w.Write([]byte("OK")); err != nil {
      fmt.Printf("failed to write to body: %v", err)
   }
}

func MidddlewareToHandler(middleware HTTPMiddleware, next http.Handler) http.Ha
```

```
    return http.HandlerFunc(func(writer http.ResponseWriter, request *http.Reques
        middleware(writer, request, next)
    })
}
```

Almost the same as the `http.Handler`, the only difference is that it also takes a `next` parameter that will be called if the current handler thinks it should. Let's see what it looks like being used:

```
package gof_go

import (
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
    "io"
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestOkHandler(t *testing.T) {
    ts := httptest.NewServer(MidddlewareToHandler(LoggingMiddleware, http.Handle
    defer ts.Close()

    res, err := http.Get(ts.URL)
    require.NoError(t, err)

    ok, err := io.ReadAll(res.Body)
    require.NoError(t, err)
    require.NoError(t, res.Body.Close())

    assert.Equal(t, "OK", string(ok))
}
```

Here we use the `MiddlewareToHandler` method to add the logging middleware to the `OkHandler`, but you could add any number of features with other middlewares here. Authorization, feature flipping, rate limiting, and all without changing the commands that handle the requests and in any order you'd like. As the `HTTPMiddleware` implementors have complete control over if the request continues or not, you can add functionality (almost like decorators) and decide if the request flow continues or not.

While this example only adds a single middleware, the actual stack could go as deep and add as many middlewares as you'd need by making more calls to `MiddlewareToHandler`. You can even go fancy and use a builder object to create the actual Chain of Responsibility, like the gorilla mux project does.

Iterator

While you can build iterators in Go for specific use cases, the fact that the language doesn't have a collections library like other languages makes it less common generally. With generics coming along in 1.18 we might see collection libraries showing up and there might be a standard iterator or an easier way for objects that are not in the language itself (like arrays, slices, and maps) to be iterated over using `for ... range` loops.

The most famous iterator in the language might be sql.Rows, which is an object everyone sees once they're dealing with SQL in Go.

Observer

Observers are almost first-class citizens in go, given the existence of channels. Most of the code you'd write to support observers, especially one producer, one consumer pattern, is already baked in how we use channels in the language. It starts to get a bit more complicated when you need to publish a message and have multiple consumers see the same message, as you'd need various channels to make a fan-out behavior work.

Strategy

Strategies are known as *classes that implement multiple algorithms but expose the same interface*. You could have numerous sorting methods (merge sort, bubble sort, quick sort) all implemented in separate classes but with the same interface (a single method that takes an array, for instance).

While you could build strategies in Go using single-method interfaces, that's not very idiomatic. You're better off defining a function signature and having each algorithm being a function that implements the same signature.

```go
package gof_go

import "testing"

type Sorter func(a []int)

func MergeSort(a []int) {
   // implementation
}

func QuickSort(a []int) {
   // implementation
}

func TestSorter(t *testing.T) {
   var sorter Sorter
   sorter = MergeSort
   sorter([]int{10, 7, 5, 2, 4})
}
```

Using functions we remove the need to have classes for every algorithm when it would just be a method in the class anyway.

Template Method

Due to the lack of inheritance and the availability of functions as first-class objects, becomes the Template Function. An abstract method available for you to implement in a subclass becomes a function taken as a parameter just like we had with factories before.

Sorting objects is a pretty typical example of Template Method still in action in Go:

```go
package gof_go

import (
   "github.com/stretchr/testify/assert"
   "sort"
   "testing"
)

type Racer struct {
   Position int
   Name     string
}

func TestSortRacers(t *testing.T) {
   racers := []*Racer{
      {
         Position: 10,
         Name:     "Alonso",
      },
      {
         Position: 2,
         Name:     "Verstappen",
      },
      {
         Position: 1,
         Name:     "Hamilton",
      },
      {
         Position: 12,
         Name:     "Vettel",
      },
   }

   sort.SliceStable(racers, func(i, j int) bool {
      return racers[i].Position < racers[j].Position
   })

   assert.Equal(t, []*Racer{
      {
         Position: 1,
         Name:     "Hamilton",
      },
      {
         Position: 2,
         Name:     "Verstappen",
      },
      {
         Position: 10,
         Name:     "Alonso",
      },
        {
            Position: 12,
            Name:     "Vettel",
        },
    }, racers)
}
```

Like other implementations, we don't know what algorithm is used to sort the objects. All we do is provide a way to compare them with a function. It removes the need to use a separate class as you'd do with implementations of Template Method in languages that don't have functions as first-class citizens.

What Is Dead May Never Die

As we've seen, many of the patterns are still here with us, in the standard library, shared third-party libraries, and frameworks. The patterns I haven't mentioned are sometimes too specific for the problems they're solving (like interpreter and visitor), so it's a bit harder to find their usage in the wild, which doesn't mean they're unnecessary.

While it might be harder to read the book nowadays if you have no C++ or Smalltalk experience, other books cover the same patterns in a newer format, like the Head First Design Patterns.

So while languages have evolved, they haven't removed the need for these patterns and the advantages they can bring to your codebase when used correctly. Keep them on your toolbelt, so you know when to use and how to spot them in code, its going to improve your coding vocabulary and the solutions you'll build.

Reduce Costs

Top comments (3)

Bruno Dias • Feb 10 '22

Nice! Great material for Software Engineering/Software Design classes @danielfireman

shogg • Feb 11 '22 • Edited

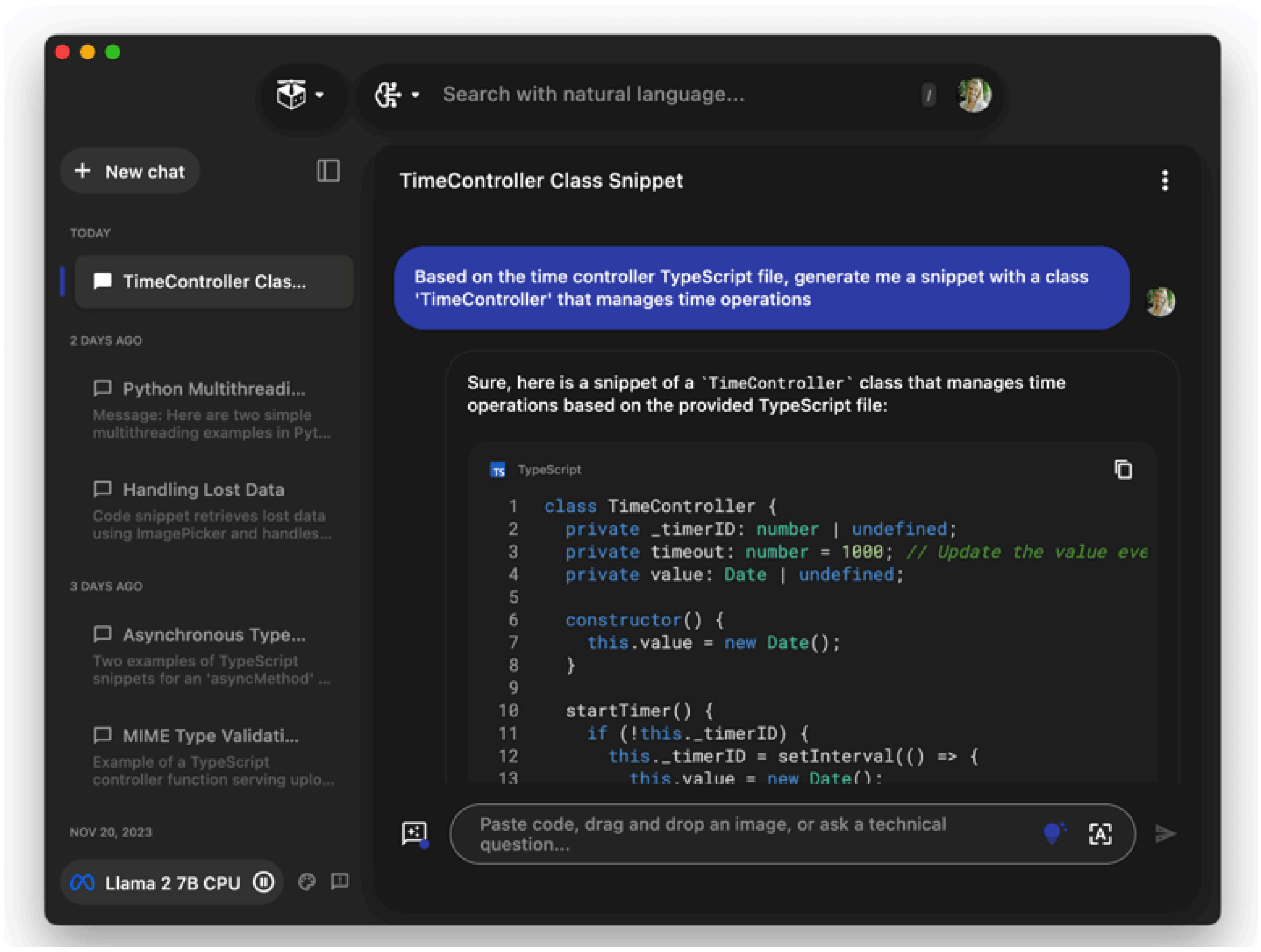
I don't like the observer pattern implemented with channels. The pattern is not about threads or asynchronicity. It's Sub-Pub, register for an event, receive the event. The event-sender doesn't care about who is interested in his events, that's the main point.

On another note: for me Go channels and Go maps in public APIs are code smells, anti patterns. I only use them in implementations.

Victor Dorneanu • Sep 5 '22

Thanks for sharing! I really liked your explanations as they were more straight to the point. Initially I've started with refactoring.guru/design-patterns/c... but sometimes I couldn't understand it properly.

Code of Conduct • Report abuse


Search with natural language...
New chat
TODAY
TimeController Clas...
2 DAYS AGO
Python Multithreadi...
Message: Here are two simple multithreading examples in Pyt...
Handling Lost Data
Code snippet retrieves lost data using ImagePicker and handles...
3 DAYS AGO
Asynchronous Type...
Two examples of TypeScript snippets for an 'asyncMethod' ...
MIME Type Validati...
Example of a TypeScript controller function serving uplo...
NOV 20, 2023
Llama 2 7B CPU
TimeController Class Snippet
TypeScript
Paste code, drag and drop an image, or ask a technical question...

Based on the time controller TypeScript file, generate me a snippet with a class 'TimeController' that manages time operations

Sure, here is a snippet of a `TimeController` class that manages time operations based on the provided TypeScript file:

```
class TimeController {
  private _timerID: number | undefined;
  private timeout: number = 1000; // Update the value eve
  private value: Date | undefined;

  constructor() {
    this.value = new Date();
  }

  startTimer() {
    if (!this._timerID) {
      this._timerID = setInterval(() => {
        this.value = new Date();
```

Our centralized storage agent works on-device, unifying various developer tools to proactively capture and enrich useful materials, streamline collaboration, and solve complex problems through a contextual understanding of your unique workflow.

👥 Ideal for solo developers, teams, and cross-company projects

Learn more

Maurício Linhares

LOCATION
Winter Garden, FL

WORK
Senior Software Engineer at DigitalOcean

JOINED
Jan 19, 2020

More from Maurício Linhares

Building distributing a command line tool in Golang
#go #tutorial #beginners #programming

Transitioning a project to new owners

#programming #management #leadership #architecture

Optimizing your MySQL queries

#mysql #database #performance #tutorial

Sentry PROMOTED

•••

Youtube Tutorial Series 📺

So you built a Next.js app, but you need a clear view of the entire operation flow to be able to identify performance bottlenecks before you launch. But how do you get started? Get the essentials on tracing for Next.js from @nikolovlazar in this video series 👀

Watch the Youtube series