- Go is a statically typed, compiled, high-level programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson.
- Syntactically similar to C.
- Also has memory safety and garbage collection.
- One of the most efficient language and runtime available currently.
- Competitor to Rust, which is also a memory safe language.

by Ashish

# Github

- Version Control and Collaboration
- Remote Hosting
- Community and Open Source
- Portfolio and Showcase
- Workflow Automation
- Integration Ecosystem

# Go Compiler

- Compiler: translator for your code
- Runtime: manages the execution of your program, memory allocation, garbage collection, etc.
- Important aspects of Go language:
  - Compiled Language with Garbage Collection
  - Role of the Go Runtime
  - Garbage Collection
  - Concurrency and Goroutines
  - Runtime Libraries and Support
  - Cross-Platform Compatibility

# Import

- Tree Shaking and Its Application
  - Tree shaking
  - Static Analysis
  - Dead Code Elimination
- Examples in Popular Frameworks
  - React
  - Angular
- Benefits of Tree Shaking
  - Reduced Bundle Size
  - Improved Performance
  - Efficient Dependency Management

by Ashish

# Arithmetic Operators

- Basic Arithmetic Operators
  - Addition +
  - Subtraction -
  - Multiplication *
  - Division /
  - Remainder (Modulus) %
- Operator Precedence
  1. Parentheses ()
  2. Multiplication *, Division /, Remainder %
  3. Addition +, Subtraction -
- Overflow
- Underflow
- Why Be Mindful of Overflow and Underflow
  - Program Stability
  - Data Integrity
  - Type Safety
- Mitigation Strategies
  - Range Checking
  - Type Conversion
  - Error Handling

by Ashish

# For Loop

- for initialization; condition; post {
  // Code block to be executed repeatedly
  }
- Initialization: Executed before the first iteration. Typically used to initialize loop variables.
- Condition: Evaluated before each iteration. If `false`, the loop terminates.
- Post: Executed after each iteration. Usually increments or updates loop variables.
- Eg: for i=1; i<=5; i++ {
  // Code block to be executed repeatedly }
- Break: Terminates the loop immediately, transferring control to the next statement after the loop.
- Continue: Skips the current iteration and moves to the next iteration of the loop.

by Ashish

# Defer

- Practical Use Cases
  - Resource Cleanup
  - Unlocking Mutexes
  - Logging and Tracing
- Best Practices
  - Keep Deferred Actions Short
  - Understand Evaluation Timing
  - Avoid Complex Control Flow

by Ashish

# Recover

- Practical Use Cases
  - Graceful Recovery
  - Cleanup
  - Logging and Reporting
- Best Practices
  - Use with Defer
  - Avoid Silent Recovery
  - Avoid Overuse

by Ashish

# Exit

- Practical Use Cases
  - Error Handling
  - Termination Conditions
  - Exit Codes
- Best Practices
  - Avoid Deferred Actions
  - Status Codes
  - Avoid Abusive Use

# Init

- Practical Use Cases
  - Setup Tasks
  - Configuration
  - Registering Components
  - Database Initialization
- Best Practices
  - Avoid Side Effects
  - Initialization Order
  - Documentation

by Ashish

# Closures

- Practical Use Cases
  - Stateful Functions
  - Encapsulation
  - Callbacks
- Usefulness of Closures
  - Encapsulation
  - Flexibility
  - Readability
- Considerations
  - Memory Usage
  - Concurrency
- Best Practices
  - Limit Scope
  - Avoid Overuse

by Ashish

# Recursion

- Practical Use Cases
  - Mathematical Algorithms
  - Tree and Graph Traversal
  - Divide and Conquer Algorithms
- Benefits of Recursion
  - Simplicity
  - Clarity
  - Flexibility
- Considerations
  - Performance
  - Base Case
- Best Practices
  - Testing
  - Optimization
  - Recursive case

by Ashish

# Pointers

- A pointer is a variable that stores the memory address of another variable.
- Use Cases
  - Modify the value of a variable indirectly
  - Pass large data structures efficiently between functions
  - Manage memory directly for performance reasons
- Pointer Declaration and Initialization
  - Declaration Syntax:
    var ptr *int
    *`ptr` is a pointer to an integer (`*int`)*
  - Initialization:
    var a int = 10
    ptr = &a  // ptr now points to a's memory address
- Pointer Operations: Limited to referencing (`&`) and dereferencing (`*`)
- Nil Pointers
- Go does not support pointer arithmetic like C or C++
- Passing Pointers to Functions
- Pointers to Structs
- Use pointers when a function needs to modify an argument's value
- `unsafe.Pointer(&x)` converts the address of `x` to an `unsafe.Pointer`

by Ashish

# Strings And Runes

- Runes and characters
  - Similarities
    - Representing Characters
    - Storage Size
  - Differences
    - Unicode Support
    - Type and Size
    - Encoding and Handling
- Practical Considerations
  - Internationalization
  - Portability
  - Efficiency

by Ashish

# FMT Package

- Printing Functions
  - Print()
  - Println()
  - Printf()
- Formatting Functions
  - Sprint()
  - Sprintf()
  - Sprintln()
- Scanning Functions
  - Scan()
  - Scanf()
  - Scanln()
- Error Formatting Functions
  - Errorf()

# FMT Package

- Format Specifiers
  - %v: The value in a default format.
  - %+v: The value in a default format with field names.
  - %#v: A Go-syntax representation of the value.
  - %T: The type of the value.
  - %%: A literal percent sign.
  - %d: Decimal integer.
  - %b: Binary representation.
  - %x: Hexadecimal representation with lowercase letters.
  - %X: Hexadecimal representation with uppercase letters.
  - %f: Floating-point number.
  - %s: String.

by Ashish

# Structs

- Structs are defined using the `type` and `struct` keywords followed by curly braces `{}` containing a list of fields.
- Fields are defined with a name and a type.
- Anonymous Structs
- Anonymous Fields
- Methods
  - func (value/pointer receiver) methodName(arguments, if any...) <return type, if any> {
    // Method implementation
    }
- Method Declaration
  - Value receiver method
    - func (t Type) methodName() {
      // Method implementation
      }
  - Pointer receiver method
    - func (t *Type) methodName() {
      // Method implementation
      }
- Comparing Structs

# Struct Embedding

- Best Practices and Considerations
  - Composition over Inheritance
  - Avoid Diamond Problem
  - Clarity and Readability
  - Initialization

by Ashish

# Generics

- Benefits of Generics
  - Code Reusability
  - Type Safety
  - Performance
- Considerations
  - Type Constraints
  - Documentation
  - Testing

by Ashish

# Text Templates

- Actions
  - Variable Insertion: {{.FieldName}}
  - Pipelines: {{functionName .FieldName}}
  - Control Structures: {{if .Condition}} ... {{else}} ... {{end}}
  - Iteration: {{range .Slice}} ... {{end}}
- Advanced Features
  - Nested Templates: {{template "name" .}}
  - Functions
  - Custom Delimiters
  - Error Handling: template.Must()
- Use Cases
  - HTML Generation
  - Email Templates
  - Code Generation
  - Document Generation
- Best Practices
  - Separation of Concerns
  - Efficiency
  - Security

by Ashish

# Epoch

- Starting Point: 00:00:00 UTC on January 1, 1970 (not counting leap seconds)
- Epoch Time Units
  - Seconds
  - Milliseconds
- Epoch Time Values
  - Positive Values
  - Negative Values
- Unix Time Functions
  - time.Now()
  - time.Unix()
  - time.Since()
- Epoch Applications
  - Database Storage
  - System Timestamps
  - Cross-Platform Compatibility
- Considerations
  - Leap Seconds
  - Precision

# Random Numbers

- Pseudo-Random Number Generation (PRNG)
  - Seed: starting point for generating a sequence of random numbers
  - rand.Intn(n)
  - rand.Float64()
- Considerations
  - Deterministic Nature
  - Thread Safety
  - Cryptographic Security

by Ashish

# Base64 Coding

- Why Base64?
  - Text Transmission
  - Storage
  - URLs and Data URLs
- Why is Encoding Important?
  - Data Storage
  - Data Transmission
  - Data Interoperability
- Common Examples of Encoding
  - Text Encoding
    - ASCII
    - UTF-8
    - UTF-16
  - Data Encoding
    - Base64
    - URL Encoding
  - File Encoding
    - Binary Encoding
    - Text Encoding

by Ashish

# Base64 Coding

- Use Cases
  - Binary Data Transfer
  - Data Storage
  - Embedding Resources
- Security Considerations
  - It is not Encryption
  - Proper handling of padding
  - Use appropriate variants

by Ashish

# bufio package

- Key Components
  - bufio.Reader
    - func NewReader(rd io.Reader) *Reader
    - func (r *Reader) Read(p []byte) (n int, err error)
    - func (r *Reader) ReadString(delim byte) (line string, err error)
  - bufio.Writer
    - func NewWriter(wr io.Writer) *Writer
    - func (w *Writer) Write(p []byte) (n int, err error)
    - func (w *Writer) WriteString(s string) (n int, err error)

- Use Cases and Benefits
  - Buffering
  - Convenience Methods
  - Error Handling

- Best Practices
  - Check Errors
  - wrap Reader and Writer instances for efficient buffered I/O operations
  - Don't forget to call Flush

# Hashing

- Key Components
  - Deterministic
  - Fast Computation
  - Pre-image Resistance
  - Collision Resistance
  - SHA-256
  - SHA-512
  - Salting

- Best Practices
  - Use of Standard Libraries
  - Algorithm Updates

by Ashish

# Writing Files

- Key Components
  - `os` Package
    - Functions
      - Create(name string) (*File, error)
      - OpenFile(name string, flag int, perm FileMode) (*File, error)
      - Write(b []byte) (n int, err error)
      - WriteString(s string) (n int, err error)

- Best Practices
  - Error Handling
  - Deferred Closing
  - Permissions
  - Buffering

by Ashish

# Line Filters

- Key Components
  - Reading Lines Individually
  - Applying Criteria or Transformations
  - Processing Each Line
  - Filtering Lines Based on Content
  - Removing Empty Lines
  - Transforming Line Content
  - Filtering Lines by Length
  - `bufio` Package
    - Scanner.Scan()
    - Scanner.Text()

- Best Practices
  - Efficiency
  - Error Handling
  - Input Sources
  - Flexibility

- Practical Applications
  - Data Transformation
  - Text Processing
  - Data Analysis

by Ashish

# File Paths

- Key Concepts
  - Absolute Path
  - Relative Path

- `path/filepath` Package
  - Functions
    - `filepath.Join`
    - `filepath.Split`
    - `filepath.Clean`
    - `filepath.Abs`
    - `filepath.Base`
    - `filepath.Dir`

- Best Practices
  - Platform Independence
  - Handling Errors
  - Security

- Practical Applications
  - File I/O Operations
  - Directory Navigation
  - Path Normalization

by Ashish

# Directories

- Key Concepts
  - os.Mkdir
  - os.MkdirAll
  - os.ReadDir
  - os.Chdir
  - os.Remove
  - os.RemoveAll
- Best Practices
  - Error Handling
  - Permissions
  - Cross-Platform Compatibility
- Practical Applications
  - Organizing Files
  - File System Navigation
  - Batch Processing

by Ashish

# Temporary Files / Directories

- Why Use Temporary Files and Directories?
  - Temporary Storage
  - Isolation
  - Automatic Cleanup
  - Default Values and Usage
- Best Practices
  - Security
  - Naming
- Practical Applications
  - File Processing
  - Testing
  - Caching
- Considerations
  - Platform Differences
  - Concurrency

by Ashish

# Embed Directive

- Why Use `embed` Directive?
  - Simplicity
  - Efficiency
  - Security
  - Default Values and Usage

- Supported Types
  - Files
  - Directories

- Use Cases
  - Web Servers
  - Configuration Files
  - Testing

- Considerations
  - File Size
  - Update Strategy
  - Compatibility

by Ashish

# Command Line

- Key Concepts
  - `os.Args` Slice
  - Parsing Arguments
  - `flag` Package
  - Default Values and Usage

- Considerations
  - Order of Arguments
  - Flag Reuse
  - Order of Flags
  - Default Values
  - Help Output

- Best Practices
  - Clear Documentation
  - Consistent Naming
  - Validation

by Ashish

# Command Line Sub Commands

- Advantages
  - Modularity
  - Clarity
  - Flexibility

- Best Practices
  - Clear Documentation
  - Consistent Naming
  - Error Handling

- Considerations
  - Help and Usage
  - Flags and Arguments
  - Nested Subcommands

by Ashish

# Environment Variables

- Best Practices
  - Security
  - Consistency
  - Documentation
- Considerations
  - Cross-Platform Compatibility
  - Default Values

by Ashish

# Logging

- Best Practices
  - Log Levels
  - Structured Logging
  - Contextual Information
  - Log Rotation
  - External Services

# JSON

- Best Practices
  - Use XML Tags
  - Validate XML
  - Handle Nested Structures
  - Handle Errors
  - Custom Marshalling/Unmarshalling
- Real-World Scenarios
  - Web Services and APIs
    - Spring Framework
    - Microsoft .NET Applications
  - Data Interchange and Storage
    - RSS and Atom Feeds
    - Electronic Data Interchange (EDI)
  - Industry Standards
    - Healthcare (HL7)
    - Finance (FIXML)

# JSON

- JSON (JavaScript Object Notation)
- `json.Marshal` - convert Go data structures into JSON (encoding)
- `json.Unmarshal` - convert JSON into Go data structures (decoding)
- Best Practices
  - Use JSON Tags
    - Mapping Struct Fields to JSON Keys
    - Omitting Fields – if empty (`omitempty`) or always (`-`)
    - Renaming Fields
    - Controlling JSON Encoding/Decoding Behavior
  - Validate JSON
  - Use `omitempty`
  - Handle Errors
  - Custom Marshalling/Unmarshalling

# XML

- Best Practices
  - Use XML Tags
  - Validate XML
  - Handle Nested Structures
  - Handle Errors
  - Custom Marshalling/Unmarshalling
- Real-World Scenarios
  - Web Services and APIs
    - Spring Framework
    - Microsoft .NET Applications
  - Data Interchange and Storage
    - RSS and Atom Feeds
    - Electronic Data Interchange (EDI)
  - Industry Standards
    - Healthcare (HL7)
    - Finance (FIXML)

by Ashish

# IO Package

- Why is the `io` Package Important?
  - Facilitates interaction with various data sources (files, networks, in-memory buffers)
  - Provides a consistent interface for handling I/O operations.
- Core Interfaces
  - io.Reader
  - io.Writer
  - io.Closer
- Common Types and Functions
  - io.Reader
  - io.Writer
  - io.Copy()
  - io.MultiReader()
  - io.Pipe()
- Working with Buffers
  - bytes.Buffer
  - bufio Package

by Ashish

# IO Package

| | io Package | bufio Package |
|---|---|---|
| Purpose | Basic interfaces and functions for I/O operations | Buffered I/O operations to improve efficiency by reducing the number of system calls |
| Core Interfaces / Types | Core Interfaces: io.Reader, io.Writer, io.Closer, io.Seeker | Core Types: bufio.Reader, bufio.Writer, bufio.Scanner |
| Common Use Cases | Direct I/O Operations, Simple Data Transfer, Pipes | Buffered Reading/Writing, Line-by-Line Reading, Chunked Reading |
| Unique Use Cases | Abstract I/O, Simple Data Manipulation | Efficient I/O Operations, Complex Tokenization |

# Maths Package

- `Pi`: The value of π (pi).
- `E`: The base of natural logarithms.
- `Phi`: The golden ratio.
- `Sqrt2`: The square root of 2.
- `SqrtE`: The square root of E.
- `SqrtPi`: The square root of Pi.
- `SqrtPhi`: The square root of Phi.
- `Ln2`: The natural logarithm of 2.
- `Log2E`: The base-2 logarithm of E.
- math.Abs(-3.14)
- math.Sqrt(16)
- math.Pow(2, 3)
- math.Exp(1)
- math.Log(math.E)
- math.Log10(100)

# Maths Package

- Trigonometric Functions
  - fmt.Println(math.Sin(math.Pi / 2)) // Output: 1
  - fmt.Println(math.Cos(math.Pi)) // Output: -1
  - fmt.Println(math.Tan(math.Pi / 4)) // Output: 1

- Inverse Trigonometric Functions
  - fmt.Println(math.Asin(1)) // Output: 1.570796326794896 6
  - fmt.Println(math.Acos(0)) // Output: 1.570796326794896 6
  - fmt.Println(math.Atan(1)) // Output: 0.7853981633974483
  - fmt.Println(math.Atan2(1, 1)) // Output: 0.7853981633974483

- Hyperbolic Functions
  - fmt.Println(math.Sinh(1)) // Output: 1.1752011936438014
  - fmt.Println(math.Cosh(1)) // Output: 1.5430806348152437
  - fmt.Println(math.Tanh(1)) // Output: 0.7615941559557649

by Ashish

# Maths Package

- Considerations
  - Precision
  - Performance
  - Special Values

- Best Practices
  - Use Constants
  - Error Handling
  - Documentation

by Ashish

# Goroutines

- Why Use Goroutines?
  - Simplify concurrent programming
  - Efficiently handle parallel tasks such as I/O operations, calculations, and more
  - Provide a way to perform tasks concurrently without manually managing threads

- Basics of Goroutines
  - Creating Goroutines (Use the `go` keyword to start a new Goroutine)
  - Goroutine Lifecycle
  - Goroutine Scheduling

by Ashish

# Goroutines

- Goroutine Scheduling in Go
  - Managed by the Go Runtime Scheduler
  - Uses M:N Scheduling Model
  - Efficient Multiplexing

- Common Pitfalls and Best Practices
  - Avoiding Goroutine Leaks
  - Limiting Goroutine Creation
  - Proper Error Handling
  - Synchronization

by Ashish

# Goroutines

- Goroutines are lightweight threads managed by the Go runtime, allowing concurrent execution of code.
- Syntax: Goroutines are created using the go keyword followed by a function call or anonymous function.
- Non-Blocking: Goroutines enable non-blocking concurrent execution, allowing programs to perform multiple tasks simultaneously.
- Asynchronous: Goroutines execute asynchronously, meaning that they run independently of the main program flow.
- Channel Communication: Communication between goroutines is achieved using channels, facilitating safe data sharing and synchronization.
- Synchronization: Goroutines can be synchronized using synchronization primitives like channels, mutexes, and wait groups to coordinate execution.
- Scalability: Goroutines are highly scalable, allowing programs to create thousands or even millions of concurrent tasks efficiently.
- Context Switching: Goroutines are managed by the Go runtime, which handles context switching between them, making it efficient and lightweight.
- Example: Common examples of goroutine usage include concurrent I/O operations, parallel processing of data, and handling asynchronous tasks.

by Ashish

# Channels

- Why Use Buffered Channels?
  - Asynchronous Communication
  - Load Balancing
  - Flow Control
- Creating Buffered Channels
  - make(chan Type, capacity)
  - Buffer Capacity
- Key Concepts of Channel Buffering
  - Blocking Behavior
  - Non-Blocking Operations
  - Impact on Performance
- Best Practices for Using Buffered Channels
  - Avoid Over-Buffering
  - Graceful Shutdown
  - Monitoring Buffer Usage

by Ashish

# Channels

- Why Use Channels?
  - Enable safe and efficient communication between concurrent Goroutines
  - Help synchronize and manage the flow of data in concurrent programs

- Basics of Channels
  - Creating Channels (make(chan Type))
  - Sending and Receiving Data (<-)
  - Channel Directions
    - Send-only: `ch <- value`
    - Receive-only: `value := <- ch`

- Common Pitfalls and Best Practices
  - Avoid Deadlocks
  - Avoiding Unnecessary Buffering
  - Channel Direction
  - Graceful Shutdown
  - Use `defer` for Unlocking

# Channels

- Common Pitfalls and Best Practices
  - Avoid Deadlocks
  - Avoiding Unnecessary Buffering
  - Channel Direction
  - Graceful Shutdown
  - Use `defer` for Unlocking

by Ashish

# Channel Synchronization

- Why is Channel Synchronization Important?
  - Ensures that data is properly exchanged between Goroutines
  - Coordinates the execution flow to avoid race conditions and ensure predictable behavior
  - Helps manage the lifecycle of Goroutines and the completion of tasks

- Common Pitfalls and Best Practices
  - Avoid Deadlocks
  - Close Channels
  - Avoid Unnecessary Blocking

by Ashish

# Multiplexing using Select

- Why Use Multiplexing?
  - Concurrency
  - Non-Blocking Operations
  - Timeouts and Cancellations

- Best Practices for Using `select`
  - Avoiding Busy Waiting
  - Handling Deadlocks
  - Readability and Maintainability
  - Testing and Debugging

by Ashish

# Non Blocking Operations

- Why Use Non-Blocking Operations?
  - Avoid Deadlocks
  - Improve Efficiency
  - Enhance Concurrency

- Best Practices for Non-Blocking Operations
  - Avoid Busy Waiting
  - Handle Channel Closure Properly
  - Combine with Contexts for Cancellations
  - Ensure Channel Capacity Management

by Ashish

# Non Blocking Operations

- Practical Usage
  - Context Cancellation
  - Timeouts and Deadlines
  - Context Values
- Best Practices
  - Avoid Storing Contexts in Structs
  - Propagating Context Correctly
  - Handling Context Values
  - Handling Context Cancellation
  - Avoid Creating Contexts in Loops
- Common Pitfalls
  - Ignoring Context Cancellation
  - Misusing Context Values

by Ashish

# Closing Channels

- Why Close Channels?
  - Signal Completion
  - Prevent Resource Leaks

- Best Practices for Closing Channels
  - Close Channels Only from the Sender
  - Avoid Closing Channels More Than Once
  - Avoid Closing Channels from Multiple Goroutines

- Common Patterns for Closing Channels
  - Pipeline Pattern
  - Worker Pool Pattern

- Debugging and Troubleshooting Channel Closures
  - Identify Closing Channel Errors
  - Use `sync.WaitGroup` for Coordination

by Ashish

# Closing Channels

- Practical Usage
  - Context Cancellation
  - Timeouts and Deadlines
  - Context Values
- Best Practices
  - Avoid Storing Contexts in Structs
  - Propagating Context Correctly
  - Handling Context Values
  - Handling Context Cancellation
  - Avoid Creating Contexts in Loops
- Common Pitfalls
  - Ignoring Context Cancellation
  - Misusing Context Values

by Ashish

# Context

- Why Use Context?
  - Cancellation
  - Timeouts
  - Values

- Basic Concepts
  - Context Creation
    - context.Background()
    - context.TODO()
  - Context Hierarchy (How contexts are created and derived)
    - context.WithCancel()
    - context.WithDeadline()
    - context.WithTimeout()
    - context.WithValue()

by Ashish

# Context

- Practical Usage
  - Context Cancellation
  - Timeouts and Deadlines
  - Context Values
- Best Practices
  - Avoid Storing Contexts in Structs
  - Propagating Context Correctly
  - Handling Context Values
  - Handling Context Cancellation
  - Avoid Creating Contexts in Loops
- Common Pitfalls
  - Ignoring Context Cancellation
  - Misusing Context Values

# Timers

- Why Use Timers?
  - Timeouts
  - Delays
  - Periodic Tasks

- The `time.Timer` Type
  - Creating a Timer
  - Timer Channel
  - Stopping a Timer

by Ashish

# Timers

- Practical Use Cases for Timers
  - Implementing Timeouts
  - Scheduling Delayed Operations
  - Periodic Tasks
  - Handle Large Numbers of Goroutines
  - Use `defer` for Unlocking

- Best Practices
  - Avoid Resource Leaks
  - Combine with Channels
  - Use `time.After` for Simple Timeouts

by Ashish

# Tickers

- Why Use Tickers?
  - Consistency
  - Simplicity

- Best Practices for Using Tickers
  - Stop Tickers When No Longer Needed
  - Avoid Blocking Operations
  - Handle Ticker Stopping Gracefully

by Ashish

# Wait Groups

- Why Use Wait Groups?
  - Synchronization
  - Coordination
  - Resource Management

- Basic Operations
  - Add(delta int)
  - Done()
  - Wait()

by Ashish

# Wait Groups

- Best Practices
  - Avoid Blocking Calls Inside Goroutines
  - Use `defer` to Call `Done`
  - Ensure Proper Use of `Add`
  - Handle Large Numbers of Goroutines
  - Use `defer` for Unlocking

- Common Pitfalls
  - Mismatch Between `Add` and `Done`
  - Avoid Creating Deadlocks

by Ashish

# Channel Directions

- Why Are Channel Directions Important?
  - Improve code clarity and maintainability
  - Prevent unintended operations on channels
  - Enhance type safety by clearly defining the channel's purpose

- Basic Concepts of Channel Directions
  - Unidirectional Channels
  - Send-Only Channels
  - Receive-Only Channels
  - Testing and Debugging

- Defining Channel Directions in Function Signatures
  - Send-Only Parameters (func produceData(ch chan<- int) )
  - Receive-Only Parameters (func consumeData(ch <-chan int) )
  - Bidirectional Channels (func bidirectional(ch chan int) )

by Ashish

# Mutex

- Why Use Mutexes?
  - Data Integrity
  - Synchronization
  - Avoid Race Conditions
- Basic Operations
  - Lock()
  - Unlock()
  - TryLock()
- Mutex and Performance:
  - Contention
  - Granularity

by Ashish

# Mutex

- Why Mutual Exclusion is Important
  - Data Integrity
  - Consistency
  - Safety

- How Mutual Exclusion is Achieved
  - Locks (Mutexes)
  - Semaphores
  - Monitors
  - Critical Sections

- Why They Are Often Used in Structs
  - Encapsulation
  - Convenience
  - Readability

# Mutex

- Best Practices for Using Mutexes
  - Minimize Lock Duration
  - Avoid Nested Locks
  - Prefer sync.RWMutex for Read-Heavy Workloads
  - Check for Deadlocks
  - Use `defer` for Unlocking

- Common Pitfalls
  - Deadlocks
  - Performance Issues
  - Starvation

by Ashish

# Atomic Counters

- Why Use Atomic Counters?
  - Performance
  - Simplicity
  - Concurrency

- Atomic Operations in Go
  - Window Duration
  - Request Counting
  - Reset Mechanism

- sync/atomic package:
  - atomic.AddInt32/atomic.AddInt64
  - atomic.LoadInt32/atomic.LoadInt64
  - atomic.StoreInt32/atomic.StoreInt64
  - atomic.CompareAndSwapInt32/atomic.CompareAndSwapInt64

by Ashish

# Atomic Counters

- What Does "Atomic" Mean?
  - Indivisible
  - Uninterruptible
- Why Use Atomic Operations?
  - Lost Updates
  - Inconsistent Reads
- How Atomic Operations Work?
  - Lock-Free
  - Memory Visibility
- Issues Without Atomic Operations:
  - Data Race
  - Inconsistent Results

by Ashish

# Atomic Counters

- What Might Go Wrong (without Atomic Counter):
  - Incorrect Final Count
  - Unpredictable Behavior
  - Possible Crashes or Corruption

- Best Practices
  - Use Atomic Operations for Simple Counters
  - Avoid Complex Operations
  - Ensure Memory Visibility
  - Monitor Performance

- Common Pitfalls
  - Incorrect Use of Atomic Operations
  - Overuse of Atomic Counters
  - Race Conditions

by Ashish

# Rate Limiting

- Why Use Rate Limiting?
  - Prevent Overload
  - Fairness
  - Abuse Prevention
  - Cost Management

- Common Rate Limiting Algorithms
  - Token Bucket Algorithm
  - Leaky Bucket Algorithm
  - Fixed Window Counter
  - Sliding Window Log
  - Sliding Window Counter

by Ashish

# Rate Limiting

## Fixed Window Counter

- How It Works
  - Each window has a counter that tracks the number of requests.
  - If the number of requests in the current window is below the limit, the request is allowed, and the counter is incremented.
  - If the number of requests reaches the limit, subsequent requests in the same window are denied.
  - At the start of a new window, the counter is reset.

- Key Points of Fixed Window Algorithm
  - Window Duration
  - Request Counting
  - Reset Mechanism

# Rate Limiting

## Token Bucket Algorithm

- How It Works
  - Tokens are added to the bucket at a fixed rate (refill rate).
  - Each request consumes one token from the bucket.
  - If the bucket has tokens, the request is allowed, and a token is removed.
  - If the bucket is empty, the request is denied.
  - The bucket has a maximum capacity to limit the number of accumulated tokens.

by Ashish

# Rate Limiting

- Practical Use Cases for Rate Limiting
  - API Rate Limiting
  - Traffic Shaping
  - Preventing Abuse
  - Load Management

- Best Practices
  - Choose the Right Algorithm
  - Handle Edge Cases
  - Monitor and Adjust
  - Graceful Handling of Rate Limits

by Ashish

# Rate Limiting

| | Token Bucket Algorithm | Fixed Window Algorithm | Leaky Bucket Algorithm |
|---|---|---|---|
| Concept | fixed number of requests per window | divides time into fixed-size intervals (windows) | controls the rate by using tokens |
| Advantages | - smooths out bursts of requests over time.<br>- more flexible and efficient for handling variable traffic patterns. | Simple to implement.<br>Easy to understand. | - constant rate, smooth and predictable.<br>- prevents sudden bursts of traffic. |
| Disadvantages | - slightly more complex to implement than Fixed Window.<br>- requires precise time-based operations to add tokens. | - can lead to bursts of requests at boundary of windows.<br>- not as flexible or smooth in rate limiting as other algorithms. | - may lead to request loss.<br>- less flexible for handling variable traffic patterns compared to Token Bucket. |

by Ashish

# Stateful Goroutines

- Why Use Stateful Goroutines?
  - State Management
  - Concurrency
  - Task Execution

- Key Concepts of Stateful Goroutines
  - State Preservation
  - Concurrency Management
  - Lifecycle Management

# Stateful Goroutines

- Handling Concurrency and Synchronization
  - Mutexes and Locks
  - Atomic Operations
  - Channels for Communication

- Common Use Cases
  - Task Processing
  - Stateful Services
  - Data Stream Processing

# Stateful Goroutines

- Best Practices
  - Encapsulate State
  - Synchronize Access
  - Synchronize Access
  - Monitor and Debug

# Sorting

- Why is Sorting Important?
  - Efficiency
  - Readability
  - Algorithms

- Built In Functions
  - sort.Ints([]int)
  - sort.Sort(sort.Interface)
  - sort.Strings

# Sorting by Functions

- Why Use Sorting by Functions?
  - Custom Criteria
  - Reusability
  - Flexibility

- Sorting by Functions
  - `sort.Interface` consists of three methods
    - Len() int
    - Less(i, j int) bool
    - Swap(i, j int)

# Sorting

- Best Practices
  - Reuse Sorting Functions
  - Optimize Comparison Logic
  - Test for Edge Cases

- Performance Considerations
  - Complexity
  - Stability
  - Memory Usage

by Ashish

# Testing / Benchmarking

- Why is Testing Important?
  - Reliability
  - Maintainability
  - Documentation

- Profiling
  - Profiling provides detailed insights into the performance of your application, including CPU usage, memory allocation, and goroutine activity.
  - Use `pprof` to collect and analyze CPU profile data.

by Ashish

# Testing / Benchmarking

- Best Practices
  - Write Comprehensive Tests
  - Maintain Test Coverage
  - Use Benchmarks Effectively
  - Profile Regularly

- Testing for Quality Assurance
- Benchmarking for Performance Optimization
- Profiling for Performance Analysis

# Spawning Processes

- Why Use Process Spawning?
  - Concurrency
  - Isolation
  - Resource Management

- os/exec package
  - exec.Command
  - cmd.Stdin / cmd.Stdout
  - cmd.Start / cmd.Wait
  - cmd.Output

by Ashish

# Spawning Processes

- Use Cases and Considerations
  - When to Use Process Spawning
    - Resource-Intensive Tasks
    - Isolation
    - Concurrency
  - Performance and Resource Management
    - Overhead
    - System Limits

by Ashish

# Signals

- Why Use Signals
  - Graceful Shutdown
  - Resource Cleanup
  - Inter-process Communication

- Signals in Unix like OS
  - SIGINT (Interrupt Signal)
  - SIGTERM (Terminate Signal)
  - SIGHUP (Hang Up Signal)
  - SIGKILL (Kill)

# Signals

- Using the ‘kill’ command
  - Find the Process ID (PID)
  - Send the Signal

- Signal Types and Usage
  - Interrupts - SIGINT
  - Terminations - SIGTERM
  - Stop/Continue - SIGCONT, SIGSTOP

# Signals

- Debugging and Troubleshooting
  - Debugging Signal Handling
  - Common Issues
    - Signal Lost
    - Deadlocks

# Reflect

- Why Use Reflection
  - Dynamic Type Inspection
  - Generic Programming
  - Serialization/Deserialization

- Few Methods
  - reflect.TypeOf
  - reflect.Value
  - reflect.ValueOf
  - reflect.ValueOf().Elem()

# Concurrency / Parallelism

- Concurrency
  - The ability of a system to handle multiple tasks simultaneously. It involves managing multiple tasks that are in progress at the same time but not necessarily executed at the same instant.
- Parallelism
  - The simultaneous execution of multiple tasks, typically using multiple processors or cores, to improve performance by running operations at the same time.

# Concurrency / Parallelism

- Practical Applications
  - Concurrency Use Cases
    - I/O Bound Tasks
    - Server Applications
  - Parallelism Use Cases
    - CPU Bound Tasks
    - Scientific Computing

# Concurrency / Parallelism

- Challenges and Considerations
  - Concurrency Challenges
    - Synchronization
    - Deadlocks
  - Parallelism Challenges
    - Data Sharing
    - Overhead
  - Performance Tuning

by Ashish

# Concurrency / Parallelism

| Aspect | Concurrency | Parallelism |
|---|---|---|
| **Definition** | Managing multiple tasks, not necessarily at the same time. | Executing multiple tasks simultaneously. |
| **Focus** | Task management and coordination | Performance through simultaneous execution |
| **Execution** | Tasks might be interleaved or scheduled | Tasks run at the same time on different cores |
| **Use Case** | Handling I/O operations, multiple connections | Computation-heavy tasks, large data processing |

# Race Conditions

- Best Practices to Avoid Race Conditions
  - Proper Synchronization
  - Minimize Shared State
  - Encapsulate State
  - Code Reviews and Testing
- Practical Considerations
  - Complexity of Synchronization
  - Avoiding Deadlocks
  - Performance Impact

by Ashish

# Deadlocks

- Causes of Deadlocks
  - Four Conditions for Deadlock
    - Mutual Exclusion
    - Hold and Wait
    - No Preemption
    - Circular Wait
- Detecting Deadlocks
  - Deadlock Detection Strategies
    - Static Analysis
    - Dynamic Analysis
  - Deadlock Detection Tools

# Deadlocks

- Best Practices for Avoiding Deadlocks
  - Lock Ordering
  - Timeouts and Deadlock Detection
  - Resource Allocation Strategies
- Best Practices and Patterns
  - Avoid Nested Locks
  - Use Lock-Free Data Structures
  - Keep Critical Sections Short
- Practical Considerations
  - Complex Systems
  - Testing for Deadlocks
  - Code Reviews

by Ashish

# RWMutex

- Key Concepts of sync.RWMutex
  - Read Lock (RLock)
  - Write Lock (Lock)
  - Unlock (Unlock and RUnlock)
- When to Use RWMutex
  - Read-Heavy Workloads
  - Shared Data Structures
- How RWMutex Works
  - Read Lock Behavior
  - Write Lock Behavior
  - Lock Contention and Starvation

by Ashish

# RWMutex

- Best Practices for Using RWMutex
  - Minimize Lock Duration
  - Avoid Lock Starvation
  - Avoid Deadlocks
  - Balance Read and Write Operations
- Advanced Use Cases
  - Caching with RWMutex
  - Concurrent Data Structures

by Ashish

# sync.NewCond

- Key Concepts of sync.NewCond
  - Condition Variables
  - Mutex and Condition Variables
- Methods of sync.Cond
  - Wait()
  - Signal()
  - Broadcast()
- Example Usage of sync.NewCond
  - Producer-Consumer Example

by Ashish

# sync.NewCond

- Best Practices for Using sync.NewCond
  - Ensure Mutex is Held
  - Avoid Spurious Wakeups
  - Use Condition Variables Judiciously
  - Balance Signal and Broadcast
- Advanced Use Cases
  - Task Scheduling
  - Resource Pools
  - Event Notification Systems

by Ashish

# sync.Pool

- Key Concepts of sync.Pool
  - Object Pooling
  - Object Retrieval and Return
- Methods of sync.Pool
  - Get()
  - Put(interface{})
  - New (Optional)

by Ashish

# sync.Pool

- Best Practices for Using sync.Pool
  - Use for Expensive Object Allocations
  - Keep Objects in Pool Clean
  - Avoid Complex Objects
  - Limit Pool Size
- Advanced Use Cases
  - Reusing Buffers
  - Managing Database Connections
  - High-Performance Applications
- Considerations and Limitations
  - Garbage Collection
  - Not for Long-Lived Objects
  - Thread Safety

by Ashish

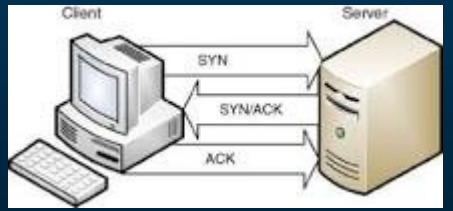
# How Internet Works

- Clients and Servers
- Protocols
- IP Addresses
- Domain Name System (DNS)

by Ashish

# A Web Request's Journey

- Step 1: Entering a URL
- Step 2: DNS Lookup, DNS Server Interaction
- Step 3: Establishing a TCP Connection
  - - browser sends a TCP SYN (synchronize) packet to the server.
  - - server responds with a SYN-ACK (synchronize-acknowledge) packet.
  - - browser sends an ACK (acknowledge) packet, completing the three-way handshake.
- Step 4: Sending an HTTP Request
- Step 5: Server Processing and Response
- Step 6: Rendering the Webpage




**Browser** **Server**

SYN ---------->
<---------- SYN-ACK
ACK ---------->

by Ashish

# How Internet Works

- Network Layers Involved:
  - Application Layer
    - Protocols: HTTP, HTTPS, DNS
    - Responsible for: High-level APIs, resource sharing, and remote file access.
  - Transport Layer
    - Protocols: TCP, UDP
    - Responsible for: End-to-end communication, error checking, and data flow control.
  - Internet Layer
    - Protocols: IP
    - Responsible for: Addressing, routing, and packet forwarding.
  - Link Layer
    - Protocols: Ethernet, Wi-Fi
    - Responsible for: Physical addressing and media access control.

# URL / URI

- URI (Uniform Resource Identifier)
- URL (Uniform Resource Locator)
- URN (Uniform Resource Name)
- Components of a URL
  - Scheme
  - Host
  - Port
  - Path
  - Query
  - Fragment

by Ashish

# Request Response Cycle

- Request-Response Cycle
  - Key Components
    - Client
    - Server
    - Protocol
  - Steps in the Request-Response Cycle
    - Step 1: Client Sends a Request
    - Step 2: DNS Resolution
    - Step 3: Establishing a Connection
    - Step 4: Server Receives the Request
    - Step 5: Server Sends a Response
    - Step 6: Client Receives the Response

by Ashish

# Request Response Cycle

- HTTP Request Components
- HTTP Response Components
- HTTP Methods: GET, POST, PUT, PATCH, DELETE
- Status Codes
- Headers
  - Request Headers
  - Response Headers
- Practical Use Cases and Examples
  - Accessing a Webpage
  - Submitting a Form
  - API Calls

by Ashish

# Request Response Cycle

- Common Issues and Troubleshooting
  - Client-Side Errors
    - 4xx Errors
  - Server-Side Errors
    - 5xx Errors
  - Network Issues
    - Timeouts
    - Troubleshooting

# Request Response Cycle

- Best Practices
  - Optimize Requests
  - Handle Errors Gracefully
  - Secure Communications

by Ashish

# Frontend / Client-Side

- Frontend (Client-Side)
  - User Interface (UI)
  - User Experience (UX)
  - Technologies Used
  - Frameworks and Libraries

# Frontend / Client-Side

- How Frontend Interacts with Backend
  - Client-Server Communication
    - HTTP Requests and Responses
    - APIs
  - Asynchronous Operations
    - AJAX (Asynchronous JavaScript and XML)
    - Fetch API

by Ashish

# Frontend / Client-Side

- Frontend Development Best Practices
  - Responsive Design
    - Definition
    - Techniques
  - Performance Optimization
    - Definition
    - Techniques
  - Accessibility
    - Definition
    - Techniques

by Ashish

# Frontend / Client-Side

- Practical Examples of Frontend Applications
  - Static Websites
  - Dynamic Web Applications
  - Single-Page Applications (SPAs)

by Ashish

# Backend / Server Side

- Key Components of Backend Development
  - Server
  - Application Logic
  - Database
  - APIs

by Ashish

# Backend / Server Side

- How Backend Interacts with Frontend
  - Client-Server Communication
    - HTTP Requests and Responses
    - APIs
  - Data Handling
    - Request Processing
    - Response Generation

by Ashish

# Backend / Server Side

- Key Concepts in Backend Development
  - HTTP Methods
  - Endpoints
  - Data Storage and Management
    - Databases
    - Data Models
  - Authentication and Authorization
  - Practical Examples of Backend Operations
    - User Registration
    - Data Retrieval
    - Payment Processing

by Ashish

# Backend / Server Side

- Backend vs. Frontend
  - Scope
  - Focus
  - Technologies
- Common Issues and Troubleshooting
  - Server-Side Errors
    - 5xx Errors
    - Troubleshooting
  - Performance Bottlenecks
    - Causes
    - Troubleshooting
  - Security Vulnerabilities
    - Risks
    - Mitigation

# HTTP

- HTTP/1.0
  - 1996
  - Features
    - Request-Response Model
    - Stateless
    - Connection

# HTTP

- HTTP/1.1
  - 1999
  - Features
    - Persistent Connections
    - Pipelining
    - Additional Headers
  - Limitations
    - Head-of-Line Blocking
    - Limited Multiplexing

# HTTP

- HTTP/2
  - 2015
  - Features
    - Binary Protocol
    - Multiplexing
    - Header Compression
    - Stream Prioritization
    - Server Push
  - Advantages
    - Reduced Latency
    - Efficient Use of Connections

# HTTP

- HTTP/3
  - 2020
  - Features
    - Based on QUIC
    - UDP-Based
    - Built-In Encryption
    - Stream Multiplexing
  - Advantages
    - Faster Connection Establishment
    - Improved Resilience

# REST API

- Application Programming Interface
  - set of rules and protocols that allows different software applications to communicate with each other
  - a standardized way for applications to interact and exchange data
- REST (Representational State Transfer)
  - Key Principles
    - Statelessness
    - Client-Server Architecture
    - Uniform Interface
    - Resource-Based
    - Stateless Communication
    - Cacheability

by Ashish

# REST API

- Key Components of RESTful APIs
  - Resources
  - Endpoints
  - HTTP Methods
  - Request and Response Formats
- Benefits of RESTful APIs
  - Scalability
  - Interoperability
  - Flexibility
  - Explanation
  - Cacheability

# Endpoints

- Components of an API Endpoint
  - Base URL - https://api.example.com/v1/
  - Path - `/users`, `/orders`, `/products`
  - Query Parameters - `?status=active&limit=10`
  - HTTP Method - `GET /users`, `POST /orders`

# Endpoints

- Types of Endpoints
  - Resource Endpoints
    - Single Resource: `/users/123`
    - Collection: `/users`
  - Action-Based Endpoints
    - `/users/123/activate`
    - `/orders/checkout`
  - Query-Based Endpoints
    - `/products?category=electronics`
    - `/orders?status=shipped&limit=10`

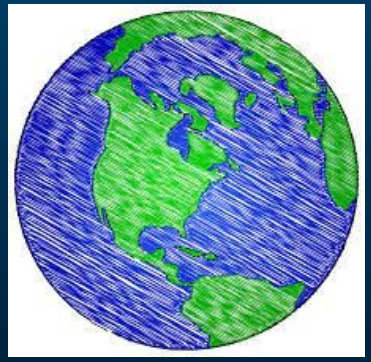
by Ashish

# Endpoints

- Designing API Endpoints
  - Resource Naming
  - Consistent and Predictable URLs
  - Versioning
  - Error Handling
- Best Practices
  - Use RESTful Principles
  - Ensure Security
  - Optimize Performance
  - Document Endpoints

Request

Response



Client sending request from anywhere in the world.

Web server hosted on a computer, connected over internet.
The web server responds to requests using one of the many ports on this computer.
That port is the single point of connection (receiver) for the rest of the world to this web server/application.

# Ports

- A computer has 65,535 ports
  - Well-known ports (0-1023)
  - Registered ports (1024-49151)
  - Dynamic or private ports (49152-65535)
- Common Ports
  - Port 80
  - Port 443
  - Port 25
  - Port 21
  - Ports 3000, 8080, 8000

# Ports

- Utility of Ports
  - Web Browsing
  - Email
  - File Transfer

by Ashish

# Modules

- Modules are collections of related Go packages
- Why Are Modules Important?
  - Versioning
  - Reproducibility
  - Organizational Clarity
- Key Commands for Working with Modules
  - go mod tidy
  - go get
  - go build
  - go run

by Ashish

# Modules

- How are packages different from modules
  - Definition and Scope
  - Purpose
  - Usage
- Relationship Between Modules and Packages
- Versioning
  - Packages – Not versioned
  - Modules

# HTTP/1.1

- Connection Behavior
  - New Connection for Each Request
  - Connection Reuse
  - Connection Closure
- Performance Considerations
  - Latency
  - Resource Consumption

by Ashish

# HTTPS (HTTP over TLS/SSL)

- Connection Behavior
  - TLS Handshake
  - Persistent Connections
  - HTTP/1.1 Features
- Performance Considerations
  - Latency
  - Connection Resumption

# HTTP2

- Connection Behavior
  - Multiplexing
  - Single Connection
  - Prioritization
- Performance Considerations
  - Reduced Latency
  - Lower Resource Usage

# SSL vs. TLS

- SSL (Secure Sockets Layer)
  - SSL 1.0: Internal use only and was never released.
  - SSL 2.0: 1995 by Netscape. Security flaws.
  - SSL 3.0: 1996, still had vulnerabilities.
- TLS (Transport Layer Security)
  - TLS 1.0: 1999 as an upgrade to SSL 3.0.
  - TLS 1.1: 2006 with improvements over TLS 1.0.
  - TLS 1.2: 2008, significant security improvements.
  - TLS 1.3: 2018, improved security and performance

by Ashish

# API Folder Structure

```
Project-root/
├── cmd/ (This folder contains the main entry point of your application. Typically one per binary.)
│   └── api/
│       ├── server.go (Entry point)
│       └── .env
├── internal/ (This folder contains the private application code, including your API handlers, models, repos, etc.)
│   ├── api/
│   │   ├── handlers/ (Use plural)
│   │   ├── middlewares/ (Use plural)
│   │   └── routers.go (Consider keeping it singular if it represents a single module or it could also be a folder)
│   ├── models/ (Use plural)
│   └── repositories/ (Use plural)
│       ├── mongodb/
│       │   └── mongoconnect.go
│       └── sqlconnect/
│           └── sqlconfig.go
├── pkg/
│   └── utils/ (Use plural)
│       ├── error_handling.go
│       └── jwt_processing.go
├── proto/
├── go.mod
└── go.sum
```

# API Planning

In this project we are going to assume that we have been contracted to create a backend server/ API for a school.
The school is our client and we are going to plan the API as per the client requirements.
So, let’s get started...

by Ashish

# API Planning

- Project Goal: Create an API for a school management system that administrative staff can use to manage students, teachers, and other staff members.
- Key Requirements:
  - Addition of student/teacher/staff/exec entry
  - Modification of student/teacher/staff/exec entry
  - Delete student/teacher/staff/exec entry
  - Get list of all students/teachers/staff/execs
  - Authentication: Login, Logout
  - Bulk Modifications: students/teachers/staff/execs
  - Class Management:
    - Total count of a class with class teacher
    - List of all students in a class with class teacher
- Security and Rate Limiting:
  - Rate limit the application
  - Password reset mechanisms (forgot password, update password)
  - Deactivate user

# API Planning

- Best Practices:
  - Modularity
  - Documentation
  - Error Handling
  - Security
  - Testing

- Common Pitfalls:
  - Overcomplicating the API
  - Ignoring Security
  - Poor Documentation
  - Inadequate Testing

# API Planning

- Executives:
  - - `GET /execs`: Get list of executives
  - - `POST /execs`: Add a new executive
  - - `PATCH /execs`: Modify multiple executives
  - - `GET /execs/{id}`: Get a specific executive
  - - `PATCH /execs/{id}`: Modify a specific executive
  - - `DELETE /execs/{id}`: Delete a specific executive
  - - `POST /execs/login`: Login
  - - `POST /execs/logout`: Logout
  - - `POST /execs/forgotpassword`: Forgot password
  - - `POST /execs/resetpassword/reset/{resetcode}`: Reset password

by Ashish

# API Planning

- Student:
  - - First Name
  - - Last Name
  - - Class
  - - Email
- Teacher:
  - - First Name
  - - Last Name
  - - Subject
  - - Class
  - - Email
- Executives:
  - - First Name
  - - Last Name
  - - Role
  - - Email
  - - Username
  - - Password

by Ashish

# API Planning

- Students:
  - - `GET /students`: Get list of students
  - - `POST /students`: Add a new student
  - - `PATCH /students`: Modify multiple students
  - - `DELETE /students`: Delete multiple students
  - - `GET /students/{id}`: Get a specific student
  - - `PUT /students/{id}`: Update a specific student
  - - `PATCH /students/{id}`: Modify a specific student
  - - `DELETE /students/{id}`: Delete a specific student

by Ashish

# API Planning

- Teachers:
  - - `GET /teachers`: Get list of teachers
  - - `POST /teachers`: Add a new teacher
  - - `PATCH /teachers`: Modify multiple teachers
  - - `DELETE /teachers`: Delete multiple teachers
  - - `GET /teachers/{id}`: Get a specific teacher
  - - `PUT /teachers/{id}`: Update a specific teacher
  - - `PATCH /teachers/{id}`: Modify a specific teacher
  - - `DELETE /teachers/{id}`: Delete a specific teacher
  - - `GET /teachers/{id}/students`: Get students of a specific teacher
  - - `GET /teachers/{id}/studentcount`: Get student count for a specific teacher

by Ashish

# API Planning

- Functionality
  - total count of a class with class teacher
  - list of all students in a class with class teacher
  - rate limit the app
  - forgot password
  - update password

by Ashish

# API Planning

- Endpoints
  - mux.HandleFunc("GET /execs", handlers.GetExecsHandler) done
  - mux.HandleFunc("POST /execs", handlers.AddExecsHandler) done
  - mux.HandleFunc("PATCH /execs", handlers.PatchExecsHandler) done
  - mux.HandleFunc("GET /execs/{id}", handlers.GetOneExecHandler) done
  - mux.HandleFunc("PATCH /execs/{id}", handlers.PatchOneExecHandler) done
  - mux.HandleFunc("DELETE /execs/{id}", handlers.DeleteExecHandler) done

  - mux.HandleFunc("POST /execs/login", handlers.LoginHandler) done
  - mux.HandleFunc("POST /execs/logout", handlers.LogoutHandler) done
  - mux.HandleFunc("POST /execs/forgotpassword", handlers.ForgotPasswordHandler) done
  - mux.HandleFunc("POST /execs/resetpassword/reset/{resetcode}", handlers.ResetPasswordHandler) // done
  - mux.HandleFunc("POST /execs/{id}/updatepasssword", handlers.UpdatePasswordHandler)

  - mux.HandleFunc("GET /students", handlers.GetStudentsHandler) done
  - mux.HandleFunc("POST /students", handlers.AddStudentsHandler) done
  - mux.HandleFunc("PATCH /students", handlers.PatchStudentsHandler) done
  - mux.HandleFunc("DELETE /students", handlers.DeleteStudentsHandler) done
  - mux.HandleFunc("GET /students/{id}", handlers.GetStudentHandler) done
  - mux.HandleFunc("PUT /students/{id}", handlers.UpdateStudentHandler) done
  - mux.HandleFunc("PATCH /students/{id}", handlers.PatchStudentHandler) done
  - mux.HandleFunc("DELETE /students/{id}", handlers.DeleteStudentHandler) done

  - mux.HandleFunc("GET /teachers", handlers.GetTeachersHandler) done
  - mux.HandleFunc("POST /teachers", handlers.AddTeachersHandler) done
  - mux.HandleFunc("PATCH /teachers", handlers.PatchTeachersHandler) done
  - mux.HandleFunc("DELETE /teachers", handlers.DeleteTeachersHandler) done
  - mux.HandleFunc("GET /teachers/{id}", handlers.GetOneTeacherHandler) done
  - mux.HandleFunc("PUT /teachers/{id}", handlers.PatchOneTeacherHandler) done
  - mux.HandleFunc("PATCH /teachers/{id}", handlers.PatchOneTeacherHandler) done
  - mux.HandleFunc("DELETE /teachers/{id}", handlers.DeleteOneTeacherHandler) done
  - mux.HandleFunc("GET /teachers/{id}/students", handlers.GetStudentsHandlerByTeacherID) done
  - mux.HandleFunc("GET /teachers/{id}/studentcount", handlers.GetStudentCountHandlerByTeacherID) done

by Ashish

# CRUD Ops

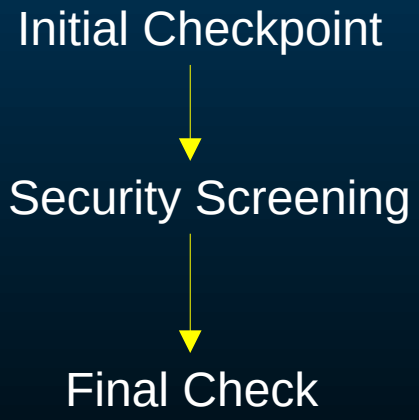
- CRUD Operations
  - Create
  - Read
  - Update
  - Delete

# CRUD Ops

- HTTP Methods
  - POST
  - GET
  - PUT
  - DELETE
  - PATCH

by Ashish

# Airport



# Middlewares

- Middleware in an API serves various purposes
  - Logging
  - Authentication and Authorization
  - Data Validation
  - Error Handling

# Middlewares

- Concept of ‘next’ in Middlewares
  - http.Handler` Interface

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

  - Middleware Pattern
  - Chaining Handlers

# CORS Middleware

- Cross-Origin Resource Sharing
  - Allow Specific Origins
  - HTTP Methods
  - Headers
  - Credentials
  - Preflight Requests

by Ashish

# HPP Middleware

- HTTP Parameter Pollution
- What HPP Middleware Does
  - Prevention of Parameter Pollution
  - Normalization
  - Security
  - Flexibility
- Path parameters in the URL (e.g., `/resource/:id`) are not subject to pollution in the same way as query or body parameters, so HPP does not address them

# Compression Middleware

- Why Use Compression Middleware?
  - Improved Performance
  - Reduced Bandwidth Usage
  - Better User Experience
  - Easy Integration

# Compression Middleware

- When You Might Not Need Compression Middleware
  - Small Payloads
  - Already Compressed Assets
  - CPU Overhead

# MariaDB (SQL)

- RDBMS
- SQL (Structured Query Language)
- Key Features of MariaDB
  - Open Source and Community-Driven
  - Compatibility with MySQL
  - Performance and Scalability
  - Storage Engines
  - Advanced Features
  - Extensive Documentation and Community Support

# MariaDB (SQL)

- How MariaDB is Different from Other Databases
  - Fork of MySQL
  - Licensing
  - Focus on Open Development
  - Enhanced Features and Performance

- Compatibility
  - Commands are identical between MySQL and MariaDB
  - Common GUI Tools like PHPMyAdmin, DBeaver, MySQL Workbench etc.
  - Both are relatively easy to install

# MariaDB (SQL)

| | MariaDB | MySQL |
|---|---|---|
| Licensing | Aims to remain free and open-source. | Owned by Oracle. available under the GNU General Public License (GPL). |
| Features | MariaDB often includes new features and performance improvements. | - |
| Storage Engines | Aria, ColumnStore, and others, providing more flexibility. | - |
| Replication and Clustering | Advanced replication features like Galera Cluster. | Group Replication and InnoDB Cluster. |

by Ashish

# .env File

- Environment Variables
  - Database credentials
  - API keys
  - Secret keys
  - Port numbers

- Why Use a .env File?
  - Security
  - Configuration Flexibility
  - Maintainability

by Ashish

# Models

- Models are used for
  - Data Representation
  - Data Validation
  - Abstraction
  - Documentation
  - Security

- Industry Standards for Models
  - Naming Conventions
  - Field Tags
  - Validation
  - Modularity

by Ashish

# Models

- Importance
  - Data Integrity
  - Maintainability
  - Scalability
  - Security

- What happens if we don’t use Models
  - Inconsistent Data
  - Code Duplication
  - Difficult Maintenance
  - Security Risks

by Ashish

# Models

- Best Practices
  - Keep Models Simple
  - Use Field Tags
  - Document Models
  - Versioning

- Common Pitfalls
  - Tight Coupling
  - Overcomplicating Models
  - Ignoring Validation
  - Lack of Documentation

by Ashish

# Data Validation

- Importance
  - Security
  - Data Integrity
  - User Experience

- Types of Data Validation
  - Format Validation
  - Presence Validation
  - Type Validation
  - Value Validation
  - Length Validation

by Ashish

# Data Validation

- Best Practices
  - Validate Early
  - Provide Clear Error Messages
  - Use Libraries and Frameworks
  - Implement Server-Side Validation
  - Be Consistent

- Common Pitfalls
  - Overly Restrictive Validation
  - Neglecting Security
  - Ignoring Data Types
  - Inadequate Testing

# Hashing (Passwords)

- BCRYPT
  - Well established, secure
  - Popular

- ARGON2
  - winner of the Password Hashing Competition (PHC)
  - three variants: Argon2d, Argon2i, and Argon2id
  - highly efficient

- PBKDF2 (Password-Based Key Derivation Function 2)
  - NIST-approved key derivation function
  - can be slower compared to bcrypt and Argon2

by Ashish

# Authentication

- The process of verifying the identity of a user or system
- Practical Examples
  - Username and Password
  - Tokens
  - Multi-Factor Authentication

# Authorization

- The process of determining what actions an authenticated user is allowed to perform.

- Practical Examples
  - Role-Based Access Control (RBAC)
  - Attribute-Based Access Control (ABAC)
  - Access Control Lists (ACLs)

# Cookies

- Carried between:
  - From Server to Client
  - From Client to Server
- Typical Information They Carry
  - Session ID
  - User Preferences
  - Tracking Information
- Usage in API/Server
  - Session Management
  - Authentication
  - Personalization

# Sessions

- Carried between:
  - From Server to Client
  - From Client to Server

- Typical Information They Carry
  - User Authentication Data
  - User Preferences
  - Shopping Cart Data

- Usage in API/Server
  - Authentication
  - Stateful Applications

# JSON Web Token (JWT)

- Carried between:
  - From Server to Client
  - From Client to Server

- Typical Information They Carry
  - User ID
  - Claims
  - Expiration Time

- Usage in API/Server
  - Authorization
  - Authentication
  - Information Exchange

# CSRF

- Cross-Site Request Forgery
- Stateless Nature
- Token-based Authentication

- Best Practices for CSRF Protection in APIs
  - Use Same-Site Cookies
  - Double Submit Cookies
  - Custom Headers
  - CSRF Tokens

- Common Pitfalls in CSRF Protection
  - Ignoring Stateless APIs
  - Weak Token Generation
  - Exposing Tokens

by Ashish

# Data Sanitization

- Importance
  - Security
  - Integrity
  - Performance
- Areas of Application
  - API/Server-Side
  - Frontend Development
- Data Sanitization in APIs/Server-Side Development
  - Input Sanitization
  - Output Sanitization
  - Database Interaction
- How Data Sanitization is Implemented
  - Escaping
  - Validation
  - Encoding
  - Whitelist Filtering

by Ashish

# Data Sanitization

- Best Practices
  - Sanitize All User Inputs
  - Use Established Libraries
  - Sanitize at Multiple Layers
  - Contextual Escaping
  - Regularly Update
- Common Pitfalls
  - Relying Solely on Client-Side Sanitization
  - Incomplete Sanitization
  - Improper Context Handling
  - Neglecting Output Sanitization
  - Over-Sanitization
- Examples of Data Sanitization
  - Preventing SQL Injection
  - Preventing XSS
  - Preventing URL Injection

by Ashish

# Protocol Buffers

- Language-agnostic binary serialization format developed by Google

- Key Features
  - Efficiency
  - Speed
  - Cross-Platform Compatibility
- Use Cases
  - Microservices Communication
  - APIs
  - Data Storage
  - Game Development
- Advantages of Using Protocol Buffers
  - Backward and Forward Compatibility
  - Strongly Typed
  - Support for Multiple Languages

by Ashish

# Protocol Buffers

- Basic Structure of a .proto File
  - ```
    syntax = "proto3";
    package example;
    // Message definition
    message Person {
        string name = 1;
        int32 id = 2;
        string email = 3;
    }
    ```
- Defining Fields
  - <field_type> <field_name> = <field_number>;
- Basic Field Types
  - int32, int64: Signed integers of varying sizes.
  - uint32, uint64: Unsigned integers.
  - float, double: Floating-point numbers.
  - bool: Boolean values.
  - string: A sequence of characters.
  - bytes: A sequence of raw bytes.
    ```

by Ashish

# Protocol Buffers

- Packages
- Package Naming Conventions
  - Lowercase
  - Dot Notation
  - Consistency
- Importing Packages
  - File: person.proto

```
syntax = "proto3";
package example;
// Message definition
message Person {
    string name = 1;
    int32 id = 2;
}
```

  - File: main.proto

```
syntax = "proto3";
package main;
// Importing another .proto file
import "example/person.proto";
message Company {
    repeated example.Person employees = 1; // Using the Person message from the example package
}
```

- Avoiding Naming Conflicts
  - File: user.proto

```
syntax = "proto3";
package user;
message User {
    string username = 1;
}
```

  - File: admin.proto

```
syntax = "proto3";
package admin;
message User {
    string adminId = 1;
}
```

by Ashish

# Protocol Buffers

- Messages
  - syntax = "proto3";
    package example;
    // Defining a message
    message Person {
        string name = 1;     // Field 1
        int32 id = 2;        // Field 2
        string email = 3;    // Field 3
    }
- A message can have the following components:
  - Field Declarations
  - Nested Messages
  - Enumerations
- Message Options
  - message OldPerson {
        option deprecated = true; // This message is deprecated
        string name = 1;
    }
- Best Practices for Messages
  - Use Meaningful Names
  - Keep Messages Focused
  - Plan for Evolution

by Ashish

# Protocol Buffers

- Fields
  - <field_type> <field_name> = <field_number>;
- Field Options
  - In `proto2`, fields could be marked as `required` or `optional`. In `proto3`, all fields are optional by default.
  - Use `repeated` to define a field that can contain multiple values of the same type.
  - You can specify additional options such as `default`, `packed`, and more.
- Field Numbers
  - Use numbers between 1 and 15 for frequently used fields, as these require only one byte in the binary encoding.
  - Use numbers between 16 and 2047 for less frequently used fields.
  - Avoid changing field numbers once they are assigned, as this can lead to incompatibility with serialized data.
- Best Practices for Fields
  - Use Meaningful Names
  - Avoid Reserved Field Numbers

by Ashish

# Protocol Buffers

- Numeric Types
  - int32/64: A 32/64-bit signed integer.
  - uint32/64: A 32/64-bit unsigned integer.
  - sint32/64: A 32/64-bit signed integer with zig-zag encoding, efficient for negative numbers.
  - fixed32/64: A 32/64-bit fixed-size integer.
  - sfixed32/64: A 32/64-bit signed fixed-size integer.
- Floating-Point Types
  - float: A single-precision (32-bit) floating-point number.
  - double: A double-precision (64-bit) floating-point number.
- Boolean Type: A Boolean value that can be either `true` or `false`.
- String Type: A sequence of UTF-8 characters, used for textual data.
- Bytes Type: A sequence of raw bytes, useful for binary data

by Ashish

# Protocol Buffers

- Enumerations
  - enum Gender {
    MALE = 0;
    FEMALE = 1;
    OTHER = 2;
    }
- Nested Messages
  - message Address {
    string street = 1;
    string city = 2;
    }
  - message Person {
    string name = 1;
    Address address = 2;  // Nested message
    }
- Field Options: Repeated Fields, Required and Optional
  - message Person {
    repeated string phone_numbers = 1;  // List of phone numbers
    }
- Comments

by Ashish

# Protocol Buffers

- Field Numbers
  - <field_type> <field_name> = <field_number>;
- Rules for Field Numbers
  - Use Positive Integers
  - Avoid Zero
  - Unique within a Message
- Best Practices for Field Numbers
  - Use Small Numbers for Frequently Used Fields
  - Use Larger Numbers for Less Common Fields
  - Reserve Numbers for Future Use
- Changing Field Numbers
  - Decommission Fields
  - Add New Fields

by Ashish

# Protocol Buffers

- Serialization and Deserialization in Protocol Buffers
  - Serialization: process of converting a data structure or object into a byte stream
  - Deserialization: reverse process of serialization

- Benefits of Serialization
  - Efficiency
  - Speed

- Best Practices for Serialization and Deserialization
  - Error Handling: Always implement error handling when serializing and deserializing to catch potential issues, such as data corruption or type mismatches.
  - Use Versioning: When evolving your message definitions, consider using field deprecation and reserved field numbers to maintain backward compatibility.
  - Avoid Circular References: Ensure that your messages do not contain circular references, as this can lead to stack overflow errors during serialization.

by Ashish

# Protocol Buffers

- How Serialization and Deserialization Work in Protocol Buffers
  - Define a Message

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    string email = 3;
}
```

  - Generate Code using protoc compiler
  - Create an Instance of the Message

```
p := &example.Person{
    Name:  "John Doe",
    Id:    123,
    Email: "john.doe@example.com",
}
```

  - Serialize the Message: This step happens internally.
  - Transmit or Store the Byte Slice
  - Deserialize the Byte Slice: This also happens internally by the generated code.
  - Access the Fields

by Ashish

# Protocol Buffers

- RPC (Remote Procedure Call)
  - a protocol that enables a program to invoke a procedure on a remote server seamlessly
- Key Features of RPC
  - Simplicity
  - Interoperability
  - Encapsulation
- Advantages of Using RPC with Protocol Buffers
  - Efficiency
  - Ease of Use
  - Cross-Language Support
- Best Practices for Using RPC
  - Versioning
  - Error Handling
  - Timeouts

by Ashish

# Protocol Buffers

- Versioning and Backward Compatibility in Protocol Buffers
- Importance of Backward Compatibility
  - Backward compatibility
  - Seamless Updates
  - Interoperability
  - Reduced Downtime
- Best Practices for Versioning and Backward Compatibility
  - Use Optional Fields
  - Avoid Changing Field Numbers
  - Deprecate Fields Instead of Removing Them
  - Use Versioned Messages
  - Test Compatibility

by Ashish

# Protocol Buffers

- Best Practices for Designing .proto Files
  - Use Meaningful Names
  - Group Related Fields
  - Utilize Enums
  - Avoid `repeated` Fields When Possible
  - Use Optional Fields Wisely
  - Document Your Fields
  - Implement Versioning
  - Avoid Deep Nesting
  - Consider Field Order

by Ashish

# gRPC

- RPC - Remote Procedure Call
- Key Features
  - Simplicity
  - Language Agnosticism
- gRPC
  - Developed by Google
  - Efficient Communication
  - Strongly Typed Messages
  - Cross-Language Support
- How gRPC Works
  - Define the Service
  - Generate Code
  - Implement the Server
  - Create the Client
  - Communication

# gRPC

- Advantages of gRPC
  - High Performance
  - Streaming Support
  - Built-in Features
  - Interoperability
  - Strongly Typed API

- Use Cases for gRPC
  - Microservices
  - Real-time Applications
  - Mobile and IoT Applications

# Stub

- A stub is a piece of code used to stand in for some other programming functionality.
- In programming, a stub can be used for various purposes
  - Testing
  - Prototyping
  - Remote Procedure Calls (RPC)
- Stub in RPC Context
  - Client-Side Stub
  - Server-Side Stub

# Service

- Collection of remote methods that can be called by clients
- Essential for structuring applications in a modular way
- Example of a Service Definition

```
syntax = "proto3";
package greeting;
// Define the service
service Greeter {
    // Define a remote procedure called SayHello
    rpc SayHello (HelloRequest) returns (HelloReply);
}
// Define the message for the request
message HelloRequest {
    string name = 1; // The name of the person to greet
}
// Define the message for the reply
message HelloReply {
    string message = 1; // The greeting message
}
```

by Ashish

# REST vs gRPC

| Feature | REST | gRPC |
|---|---|---|
| **Architecture** | Resource-oriented | Service-oriented |
| **Protocol** | HTTP/1.1/2 | HTTP/2 |
| **Data Format** | JSON, XML | Protocol Buffers (binary format) |
| **Communication Style** | Request-Response | Remote procedure calls |
| **Performance** | Slower due to text-based formats | Faster due to binary serialization |
| **Streaming Support** | Limited | Full support |
| **Error Handling** | HTTP status codes | gRPC status codes |
| **Code Generation** | Manual generation of endpoints | Automatic code generation from .proto files |

# REST vs gRPC

- Advantages of REST
  - Simplicity
  - Wide Adoption
  - Human-Readable
  - Statelessness

- Advantages of gRPC
  - Performance
  - Strongly Typed APIs
  - Streaming
  - Automatic Code Generation

# REST vs gRPC

- Disadvantages of gRPC
  - Complexity
  - Less Human-Readable
  - Browser Support
- Disadvantages of REST
  - Overhead
  - Limited Features
  - Versioning
- When to Use REST vs gRPC
  - REST
    - A simple CRUD (Create, Read, Update, Delete) API.
    - To expose resources to a wide variety of clients, including browsers.
    - To take advantage of existing REST tools and libraries.
  - gRPC
    - High performance and low latency.
    - Real-time communication or streaming capabilities.
    - To enforce strict contracts with clients using strongly typed APIs.

by Ashish

# gRPC Streaming

- Types of Streaming
  - Server Streaming
  - Client Streaming
  - Bidirectional Streaming

- Benefits of gRPC Streaming
  - Real-Time Communication
  - Efficiency
  - Handling Large Data

by Ashish

# Advanced gRPC Features

- Deadlines and Timeouts
- Compression
- Message Compression
- Reflection
- gRPC Gateway for RESTful APIs

by Ashish

# SQL/NoSQL Comparison

| Feature | SQL | NoSQL |
|---|---|---|
| **Database** | Collection of tables | Collection of collections |
| **Table vs. Collection** | A structured set of data organized into rows and columns | Similar to a table in SQL. It stores documents, which are JSON-like objects. |
| **Row vs. Document** | Represents a single record in a table | Equivalent of a row in SQL |
| **Column vs. Field** | Represents a single attribute or field in a table | Similar to a column in SQL. It stores data in a key-value pair format |
| **Schema** | Defines the structure of the table | Schema-less |
| **Primary Key vs. _id** | A unique identifier for each row in a table | _id field serves as the primary key for a document in a collection |
| **Indexes** | Used to speed up queries | Used to speed up queries in similar way |

by Ashish

# SQL/NoSQL Comparison

| Feature | SQL | NoSQL |
|---|---|---|
| **Joins vs. Embedding/Referencing** | To combine data from multiple tables | Does not support joins in the traditional SQL sense. Embedding is supported. |
| **SQL Queries vs. MongoDB Queries** | Uses structured query language (SQL) | Uses its own query language |
| **Transactions** | Allow you to execute multiple operations atomically | Also supports transactions |
| **Aggregation** | Provides aggregate functions like SUM(), COUNT(), AVG() | Aggregation framework allows you to perform complex data transformations and calculations |
| **Foreign Keys vs. Manual Referencing: (Table Relations)** | Enforce relationships between tables, ensuring referential integrity | You manually reference related documents using fields, but there's no enforcement of referential integrity |
| **ACID Compliance** | Typically ACID-compliant | Also ACID-compliant |

# MongoDB

- NoSQL Database

- Key Features of MongoDB:
  - Document-Oriented Storage
  - Dynamic Schema
  - Scalability
  - High Performance
  - Indexing
  - Aggregation Framework
  - Replication

# MongoDB

- Flexibility
- Ease of Use
- Fast Development Cycle
- Big Data Handling
- Versatility
- Community and Ecosystem