

Microservices

Training overview

1. Go basics

Syntax, data structures, interfaces, ...

2. Go basics

Best practices, profiling, docker, rest APIs, ...

3. Microservices

Monoliths, containers, Kubernetes, packaging, ...

4. Microservices

CI/CD, scaffold, logging, monitoring, troubleshooting, ...

5. Workshop

Building microservices with Go

Part 1

- Questions?
- Mono, micro, macro
- Networking
- JSON
- Exercise 1
- API Testing

Part 2

- Gin Gonic
- Exercise 2
- gRPC
- Connect-Go
- Exercise 3

Questions?

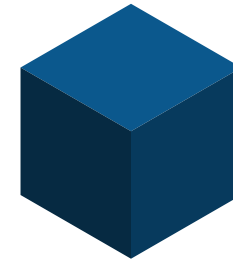
Mono, micro, macro

Mono, micro, macro

Microservices - why have one problem when you can have a dozen?

Monolith - 1/3

- Wikipedia: "Stone block made of one single piece"
- Refers to applications that are:
 - Composed of one piece
 - Tightly coupled
 - Contains all the business logic

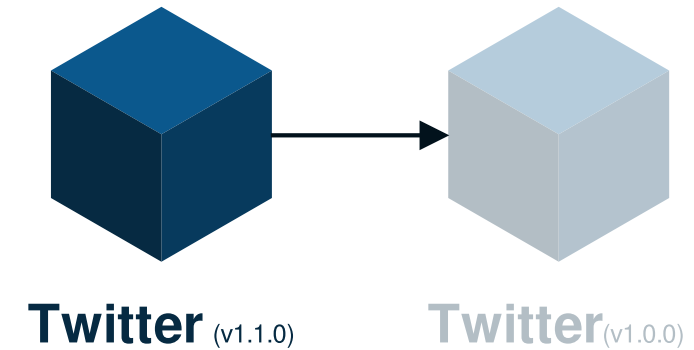


Twitter

Monolith - 2/3

Advantages

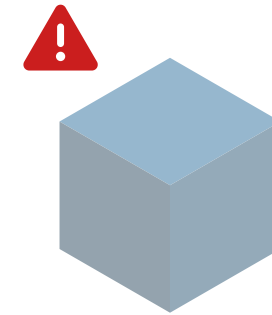
- Simple to deploy
- Easy to understand
- Possibly improved performance
- Easy to test and debug



Monolith - 3/3

Disadvantages

- Difficult to scale
- Single point of failure
- Hard to adapt new technology

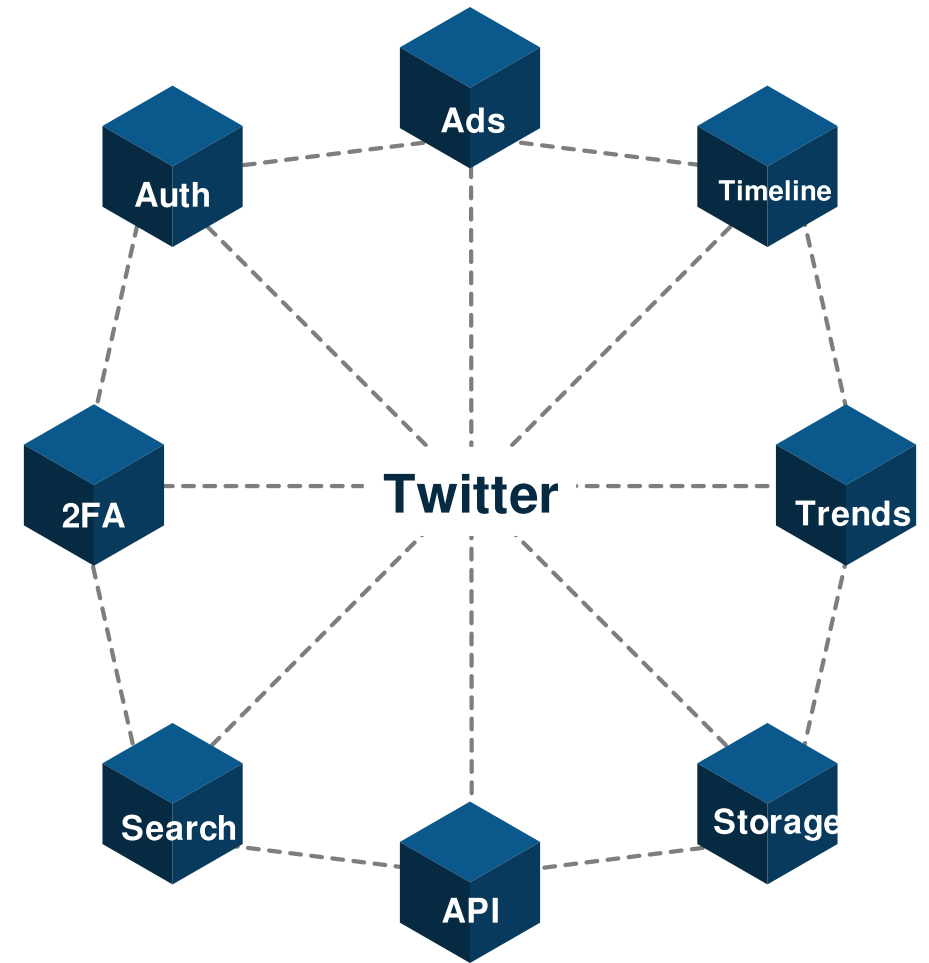


Twitter

● **Unhealthy**_(0/1)

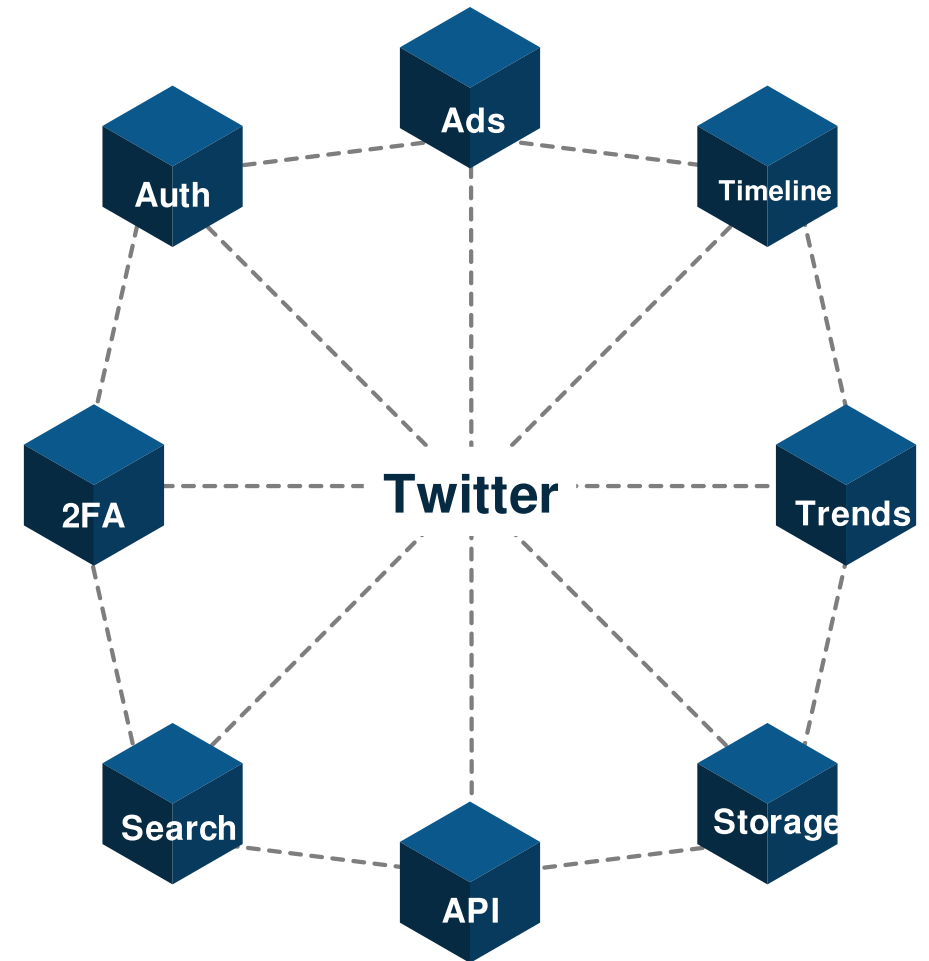
Definition of a "microservice"?

- Anyone?



Microservice - 1/3

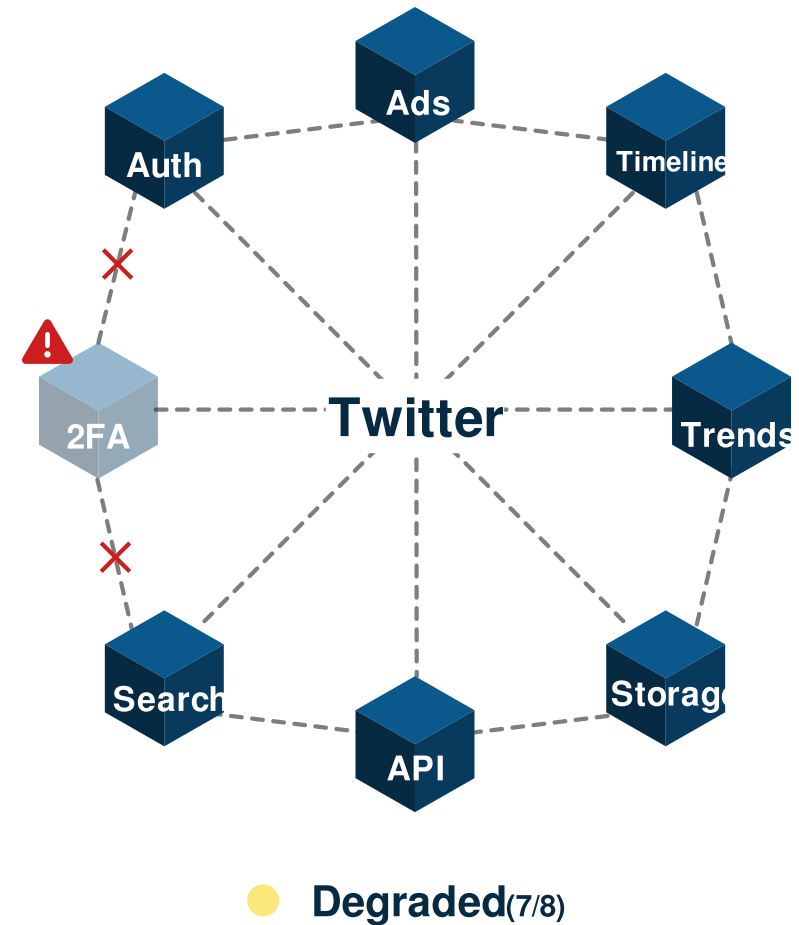
- There is no formal definition, kind of a "buzzword"
- Generally defined as a service that is:
 - Small and independent
 - Loosely coupled
 - Often communicates over a network



Microservice - 2/3

Advantages

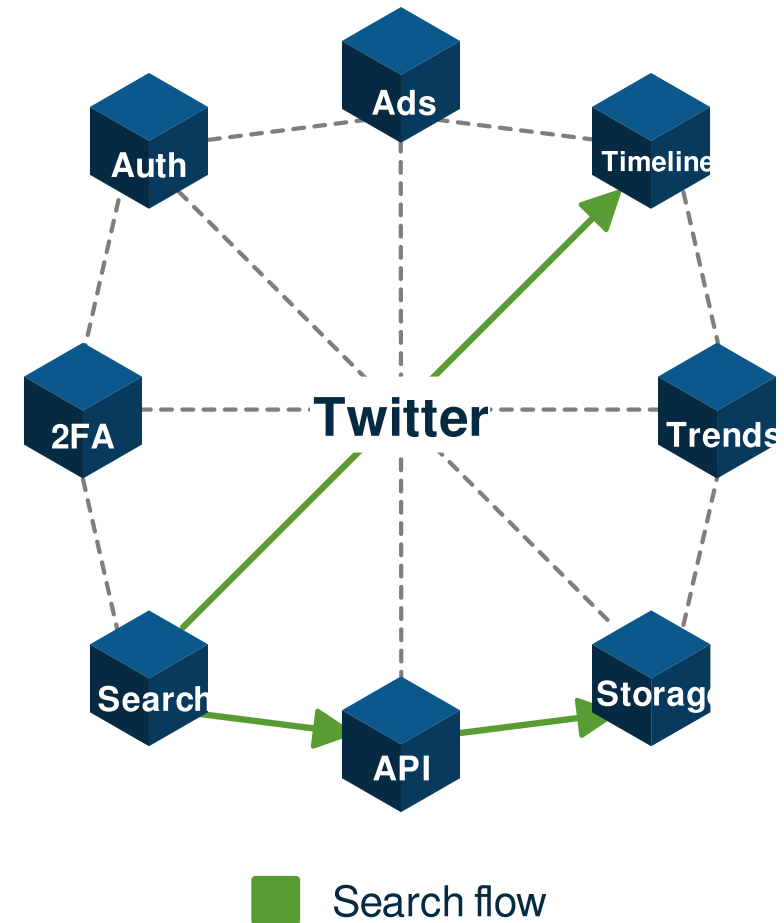
- Easy to scale
- No single point of failure
- Easy to adapt new technology
- Improved observability



Microservice - 3/3

Disadvantages

- Hard to deploy
- Can be difficult to understand
- Possibly reduced performance
- Hard to test and debug



Macroservice - 1/3

- Again, no formal definition
- "Pragmatic approach" to microservices
- Hybrid of monolith and microservice architecture

Macroservice - 2/3

Why?

- Cost of creating a microservice is high
- For every service:
 - Deployment
 - CI/CD pipeline
 - Monitoring & logging
 - Communication with other services

Macroservice - 3/3

Word of advice

- Do not create a microservice for every single responsibility
- Be conservative in creating or splitting up microservices

In short:

- Be pragmatic, don't over-engineer.
- Refactoring is (or should be) easy.

Networking

- Most services need to communicate over a network
- Most common is HTTP, but there are other protocols
- `net` package contains TCP/UDP server and client
- `net/http` package contains (HTTP/2 ready) server and client

HTTP server - 1/4

- Server can be started by using `http.ListenAndServe(...)`
- `ListenAndServe` takes an address (`string`) and a handler (`http.Handler`)

Handler definition

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

HTTP server - 2/4

Basic server setup

```
type handler struct {}

func (h handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("Hello, world!"))
}

func main() {
    if err := http.ListenAndServe(":8000", handler{}); err != nil {
        log.Fatal(err)
    }
}
```

HTTP server - 3/4

Alternatively, use `http.HandlerFunc` to create a handler from a function.

Handler function

```
func main() {  
    handler := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {  
        w.WriteHeader(http.StatusOK)  
        _, _ = w.Write([]byte("Hello, world!"))  
    })  
  
    if err := http.ListenAndServe(":8000", handler); err != nil {  
        log.Fatal(err)  
    }  
}
```

Only `http.Handler` is accepted, how come we can use a function?

Definition

```
type HandlerFunc func(ResponseWriter, *Request)
//              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//              Same signature as ServeHTTP

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
//              ^^^^^^^^^
//              Implements `http.Handler`

    f(w, r)
//  ^^^^^^^
//  Calls itself
}
```

Breakdown - 1/2

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

Summary

- Defined as an alias for a function
- Function has the same signature as `ServeHTTP`
- Any function with the same signature can be converted to `HandlerFunc`

Usage

```
http.HandlerFunc(handler)
```

Breakdown - 2/2

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {  
    f(w, r)  
}
```

Summary

- `HandlerFunc` implements `http.Handler`
- Methods can be attached to any type (even primitive types)
- The receiver (`f`) is a function so it can be called

HTTP server - 4/4

- Only a single handler can be used but handlers are composable
- Routing can still be done by using `http.ServeMux`

Example

```
mux := http.NewServeMux()
mux.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
    //          ^^^
    //          Exact match > Prefix match

    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("Hello, world!"))
})
```

Request

- Context can be retrieved by using `Request.Context()`
 - When user cancels request this context will be cancelled
- `Request` struct contains incoming data (URL, headers, body, etc.):

Example

```
var req *http.Request

// POST /auth/login HTTP/1.1
// Authorization: Bearer TOKEN

method := req.Method // POST
path := req.URL.Path // /auth/login
auth := req.Header.Get("Authorization") // Bearer TOKEN
```

Response

- `ResponseWriter` is used to write the response to the client, it contains:
 - `Header() http.Header` to set headers
 - `WriteHeader(int)` to set the HTTP status code
 - `Write([]byte) (int, error)` to write the response body

Example

```
var w http.ResponseWriter

w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusOK)
_, _ = w.Write([]byte(`{"message": "Hello, world!"}`))
```

JSON

Unmarshal JSON - 1/3

- Decoding/deserialization is called unmarshalling
- Use `encoding/json` package can to unmarshal JSON data
- Uses reflection to map JSON data to Go types

Unmarshal JSON - 2/3

Types

- Array maps to `[]T`
- Object maps to `struct`
- String maps to `string`
- Number maps to `int` / `float`

Unmarshal JSON - 3/3

```
type User struct {
    Email string
}

func main() {
    var user User

    data := []byte(`{"Email": "test@test.com"}`)
    _ = json.Unmarshal(data, &user)
// .          ^^^^^
//          Pass pointer to: user

    fmt.Println(user.Email)
    // Prints: test@test.com
}
```

Marshal JSON - 1/2

Process of encoding/serialization is called marshalling

Example

```
type User struct {  
    Email string  
}  
  
func main() {  
    user := User{Email: "test@test.com"}  
    data, _ := json.Marshal(user)  
  
    fmt.Println(data) // Prints: {"Email":"test@test.com"}  
}
```


Marshal JSON - 2/2

Note: only exported fields are marshalled

Example

```
type User struct {  
    email string  
    Username string  
}  
  
func main() {  
    user := User{email: "test@test.com", "Username": "foo"}  
    data, _ := json.Marshal(user)  
  
    fmt.Println(data) // Prints: {"Username":"foo"}  
}
```

Struct tags

- Struct tags can be used to customize the JSON encoding
- Format: `json:"name,option1,option2"`
- `omitempty` option can be used to omit empty values

Example

```
type User struct {  
    Email string `json:"email"`,  
    //          ^^^^^^^^^^^^^^^  
    //          Custom name for JSON field  
}
```

Number types

- Not all types map directly to Go
- For more advanced use-cases, use `json.Number` type
 - Parses a JSON number as a string
 - Can be converted to `int64` / `float64`

Example

```
type User struct {  
    ID json.Number `json:"id"`  
}  
  
id, _ := user.ID.Int64()
```

Custom (un)marshalling

- Use `MarshalJSON()` / `UnmarshalJSON()` to customize (un)marshalling
- Both are interfaces that can be implemented on any type

Example

```
type URN string

func (u URN) MarshalJSON() ([]byte, error) {
    return json.Marshal(fmt.Sprintf("urn:%s", u))
}
```

An example - 1/2

- Lets say that we build a todo app
- Which contains an endpoint to get all todos

GET /todos

```
{
  "items": [
    {"id": 1, "title": "Buy milk", "completed": false},
    {"id": 2, "title": "Buy eggs", "completed": true}
  ],
  "total": 2
}
```

An example - 2/2

- We can map the data to Go types

Mapping in Go

```
type Todo struct {  
    Id          int    `json:"id"`  
    Title       string `json:"title"`  
    Completed   bool   `json:"completed"`  
}  
  
type TodoListResponse struct {  
    Items []Todo `json:"items"`  
    Total int   `json:"total"`  
}
```

XML

Works similarly

Marshal XML

```
package main

import (
    "encoding/xml"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    p := &Person{Name: "John Doe", Age: 25}
    output, _ := xml.Marshal(p)

    println(output)
    // Output: <Person><Name>John Doe</Name><Age>25</Age></Person>
}
```


Demonstration

- JSON
- XML
- (Un)marshal structs
- Struct tags

Exercise 1

Prerequisites

- Update to Go 1.21
- Create a new project (e.g. `go mod init modulepath`)

Important!

- Assignment should be completed
- Next slides depend on this

What is S3?

- Scalable object storage service by AWS.
- Stores/retrieves data from anywhere on the web.
- Simple specification based on REST & XML



S3

- We will build a (tiny) S3 server
- The assignment is split into two parts

Parts

1. Implement the initial `ListBuckets` endpoint using the Go standard library
2. Continue with the other endpoints using Gin Gonic (HTTP framework)

S3 / part 1

Go to

training.brainhive.nl

API Testing

Strategies

- Unit tests; `httpptest`
- Smoke tests; `hurl`
- Contract testing; `...`

Unit tests - 1/2

- `httptest` package can be used to write tests for servers and clients
- `httptest.NewRequest()` can be used to create a new (fake) request
- Request flow can be tested by using `httptest.NewRecorder()`
 - Implements the `http.ResponseWriter` interface
 - Writes HTTP response to `Result()` method

Unit tests - 2/2

Example

```
func TestListBuckets(t *testing.T) {  
    req := httptest.NewRequest(http.MethodGet, "/", nil)  
    w := httptest.NewRecorder()  
  
    handler(w, req)  
  
    res := w.Result()  
    defer res.Body.Close()  
  
    // Assert response  
}
```


Hurl

Command line tool that runs HTTP requests defined in a simple plain text format.

Hurl example

```
GET http://localhost:7000/  
  
HTTP 200  
Content-Type: application/xml  
[Asserts]  
header "Content-Type" contains "utf-8"
```

Contract testing

Easily verify compatibility of changes across interdependent services

Demonstration

- Unit tests
- Hurl

Gin Gonic

Gin is a (popular) HTTP web framework for Go

Features

- Zero allocation router
- Fastest http router and framework
- Complete unittest suite
- API frozen, no breaking changes

Engine

- Everything is built around the `gin.Engine` type
- Includes middleware, routes and configuration
- `gin.Default()` creates a new engine with default middleware

Example

```
r := gin.Default()

r.Use(gin.Logger())
//      ^^^^^^^^^^^^^^^
//      Setup global middleware

_ = r.Run(":8000")
```

Routes / GET

- Routes are defined using the `gin.Engine` methods
- HTTP methods are methods on the engine type

Example

```
r := gin.Default()
r.GET("/status", status)

func status(c *gin.Context) {
    c.JSON(http.StatusOK, []string{"a", "b"})
    //          ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //          Any Go type that marshals to JSON
}
```


Routes / POST

- Works similar to GET requests
- Parse incoming data using `c.Bind(..)`
 - Example in next slide

```

type StatusRequest struct {
    Status string `json:"status"`
}

func main() {
    r := gin.Default()
    r.POST("/status", status)
    _ = r.Run(":8000")
}

func status(c *gin.Context) {
    var req StatusRequest

    if err := c.Bind(&req); err != nil {
        c.JSON(http.StatusBadRequest, "invalid request")
        return
    }

    // Do something with: req
}

```

Routes / params

- Parameters can be extracted from the path
- Format is: `:paramName`

Example

```
r := gin.Default()
r.GET("/status/:id", status)

func status(c *gin.Context) {
    id := c.Param("id")
    // Do something with: id
}
```

Routes / wildcard

- Wildcard parameters can be extracted from the path
- Format is: `*paramName`

Example

```
r := gin.Default()
r.GET("/status/*id", status)

func status(c *gin.Context) {
    id := c.Param("id")
    // Do something with: id
}
```

S3 / part 2

Go to

training.brainhive.nl

gRPC

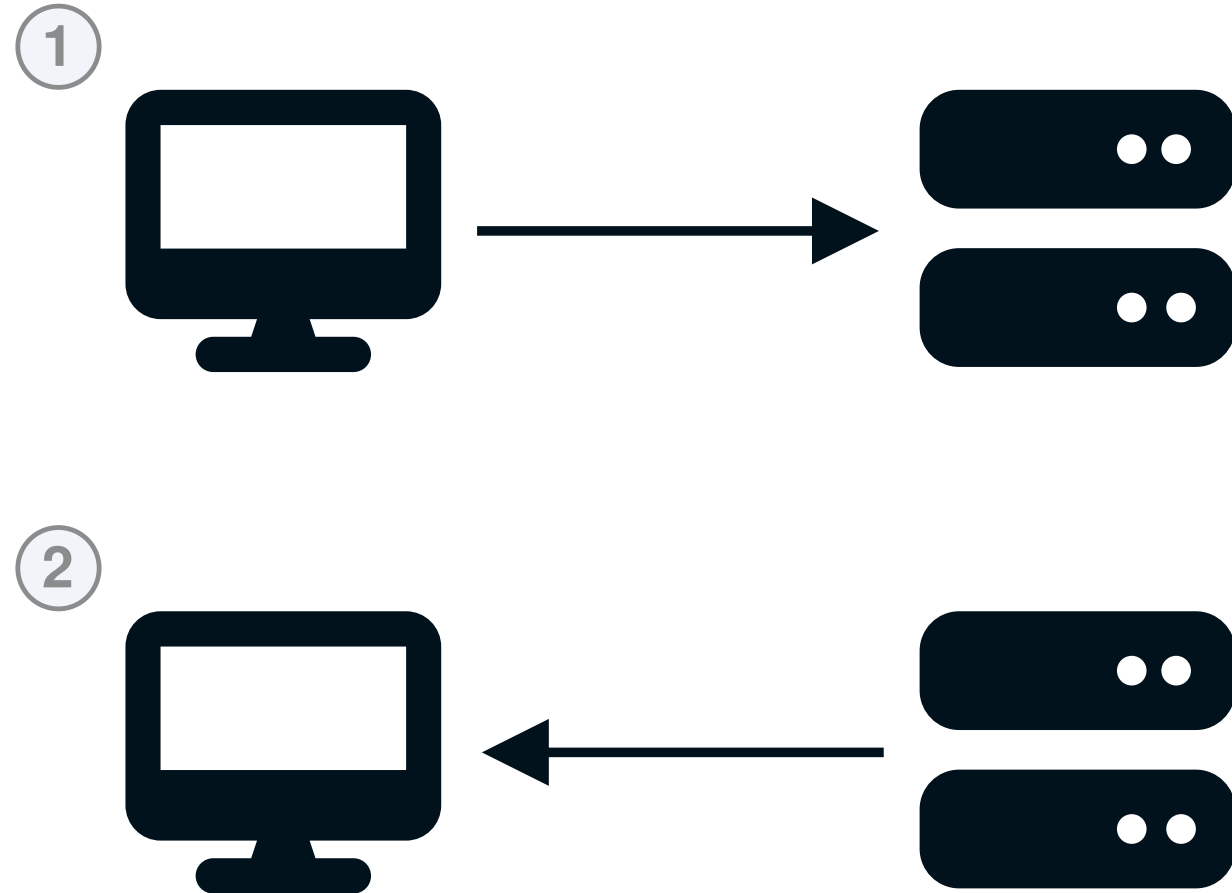
Introduction

- Modern, open-source, high-performance RPC framework
- Low latency, back/forwards compatible, highly scalable
- Works with protocol buffers for serialization/deserialization
- Supports:
 - Client/server/bi-directional streaming
 - Cancellation and timeouts
 - Metadata and authentication

Request flow - 1/4

Unary RPC

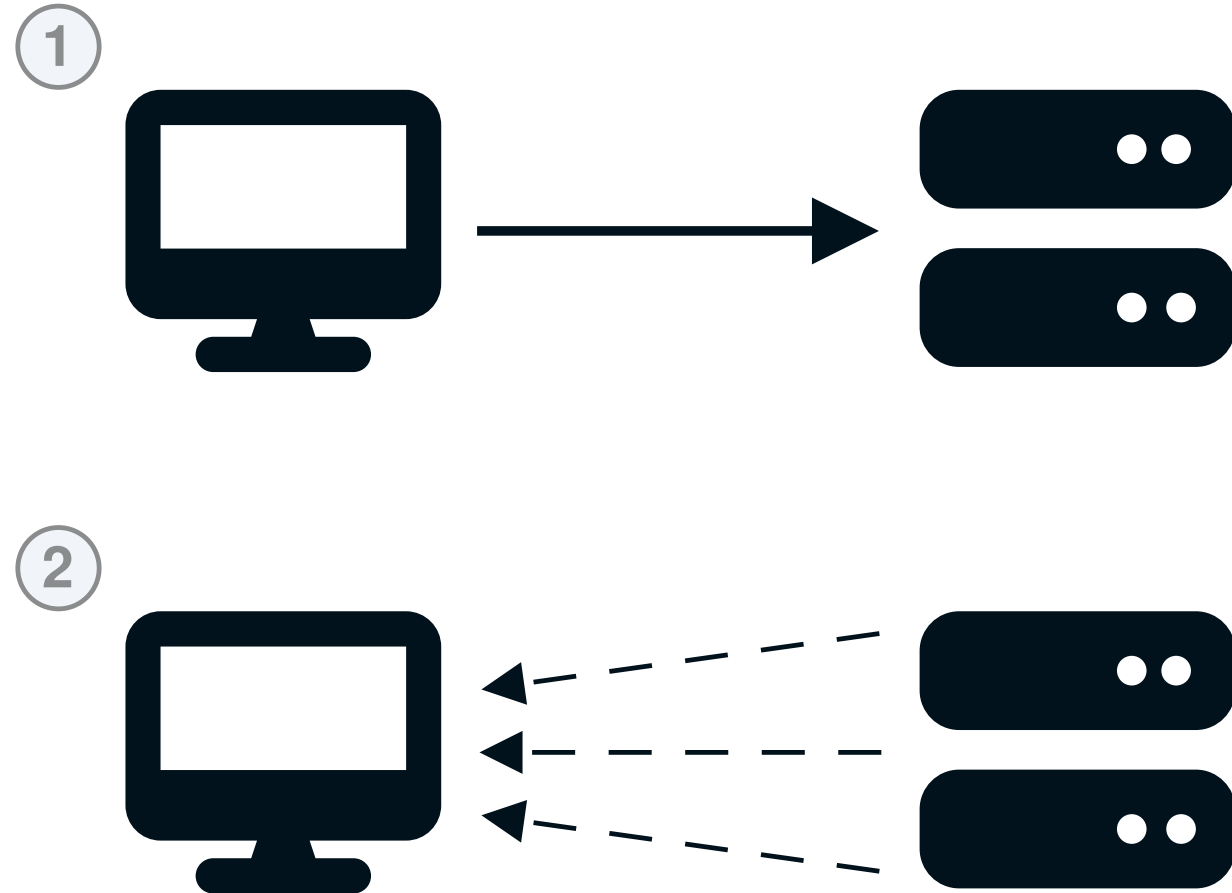
- Similar to HTTP request/response flow
- Client sends metadata and request message
- Server responds with, either:
 - Success: response message
 - Error: status code and details



Request flow - 2/4

Server streaming RPC

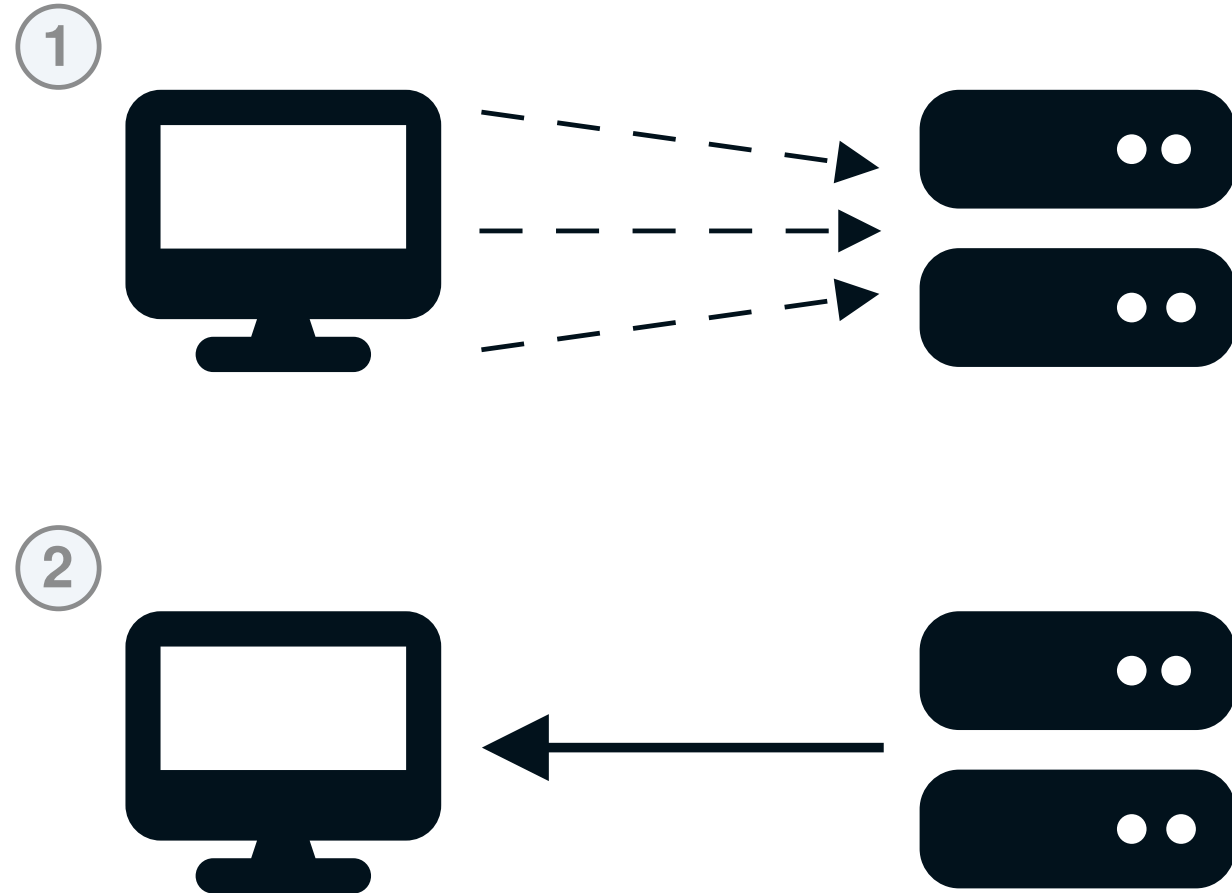
- Client sends metadata and request message
- Server continuously responds with response messages



Request flow - 3/4

Client streaming RPC

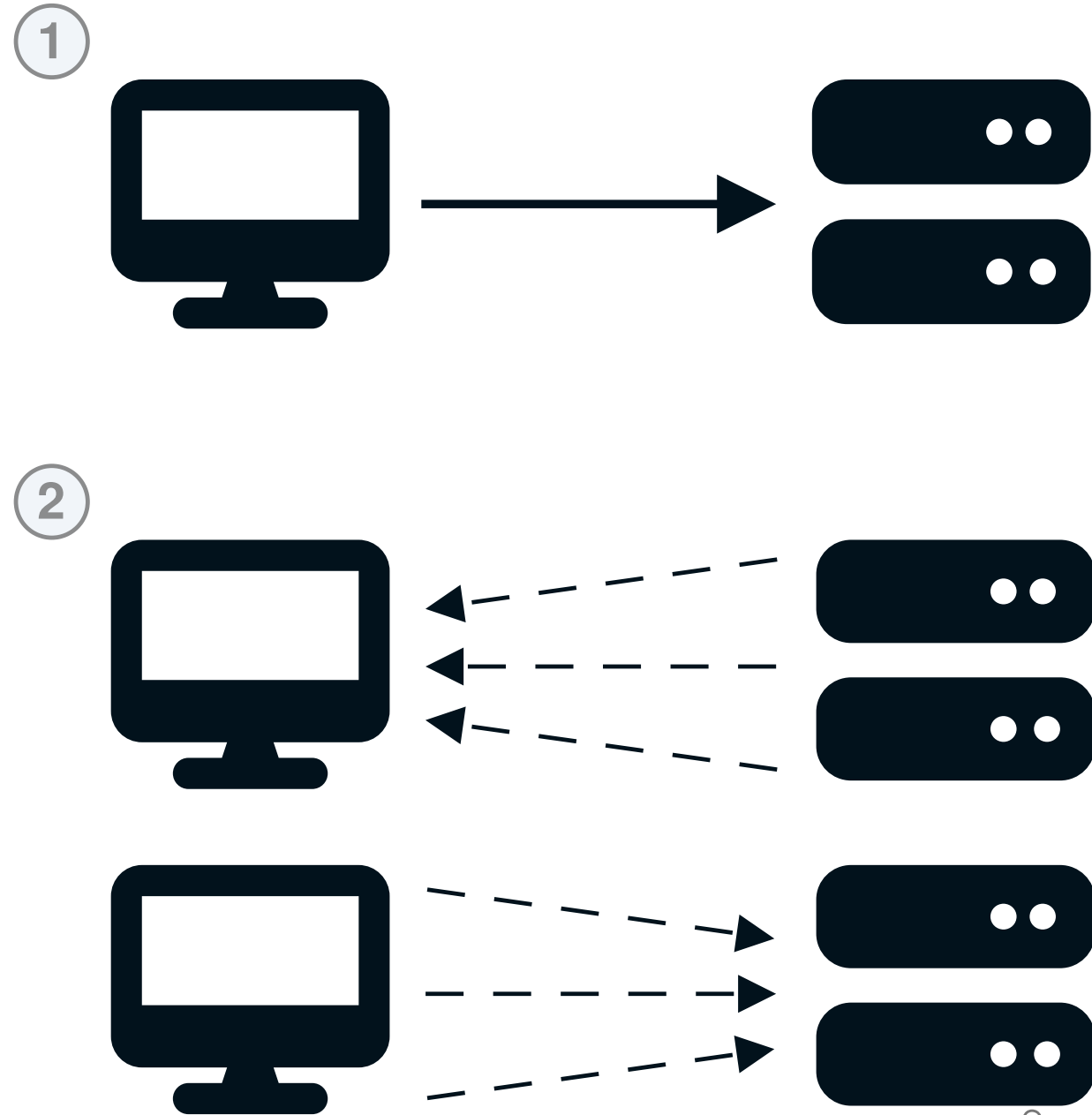
- Client continuously sends request messages
- Server sends metadata and response message when all messages are received



Request flow - 4/4

Bidirectional streaming RPC

- Client sends metadata and request message
- Client continuously responds with response messages
- Server continuously responds with response messages
- Actual behaviour is application specific



Protocol Buffers

Also known as "protobuf"

Proto service definiton

Example

```
service PingService {  
    rpc Ping (PingRequest) returns (PingResponse);  
}  
  
message PingRequest {  
}  
  
message PingResponse {  
    bool pong = 1;  
    //    ^^^^    ^^^^    ^  
    //    Type Name    Field number  
}
```

What is it?

- Binary (de)serialization format
- Developed by Google, currently at version 3
- Has its own compiler (`protoc`) which generates code

Example

```
message Order {  
    int32 id      = 1;  
    float price   = 2;  
}
```

Messages and fields

- Field numbers are used to identify fields (instead of names)
- For backwards compatibility, field numbers should not be changed
- Fields can unused if deprecated (but not removed)

Example

```
message Order {  
    int32 id          = 1;  
    float  price_old  = 2; // Field is deprecated, not removed  
    bool   disabled   = 3;  
    string price      = 4; // New field  
}
```

Backwards compatibility - 1/3

- Imagine a simple service that has been released to the public
- The email address is no longer being used, so it can be removed
- The `username` field is added to replace the email address

Old version

```
message User {  
    int32 id      = 1;  
    float price   = 2;  
    // ^^^^^  
    // Type as float  
    bool disabled = 3;  
}
```


Backwards compatibility - 2/3

What will happen if we don't follow these best practices?

New "naive" version

```
message User {  
    int32 id      = 1;  
    string price  = 2;  
    // ^^^^^  
    // Type is now string  
    bool disabled = 3;  
}
```

Backwards compatibility - 3/3

- Both fields have the same field number, this won't work
- Old clients will try to decode price as float
- This will fail as the new version uses a string

Solution

```
message User {  
    int32 id      = 1;  
    float price_old = 2;  
    bool disabled = 3;  
    string price   = 4;  
}
```

Arrays

- Fields can be repeated (which maps to `[]T`)
- Supports zero or more values

Example

```
message User {  
    int32 id = 1;  
    repeated string emails = 1;  
}
```

Enums

- Enums can be used to define a set of named constants
- Enum values are always integers, defaults to 0
- Zero value is always the first value and must be defined

Example

```
enum Status {  
    STATUS_UNKNOWN    = 0;  
    STATUS_PENDING    = 1;  
    STATUS_COMPLETED  = 2;  
}
```

Hierarchy

- Messages can be composed of other messages
- And can also be nested inside other messages

Example

```
message User {  
    message Address {  
        string zipcode = 1;  
        string house_no = 2;  
    }  
  
    int32 id = 1;  
    Address address = 2;  
}
```

Error handling

- gRPC uses a status code to indicate the result of a request
- These errors can be created using `status.Error(...)`

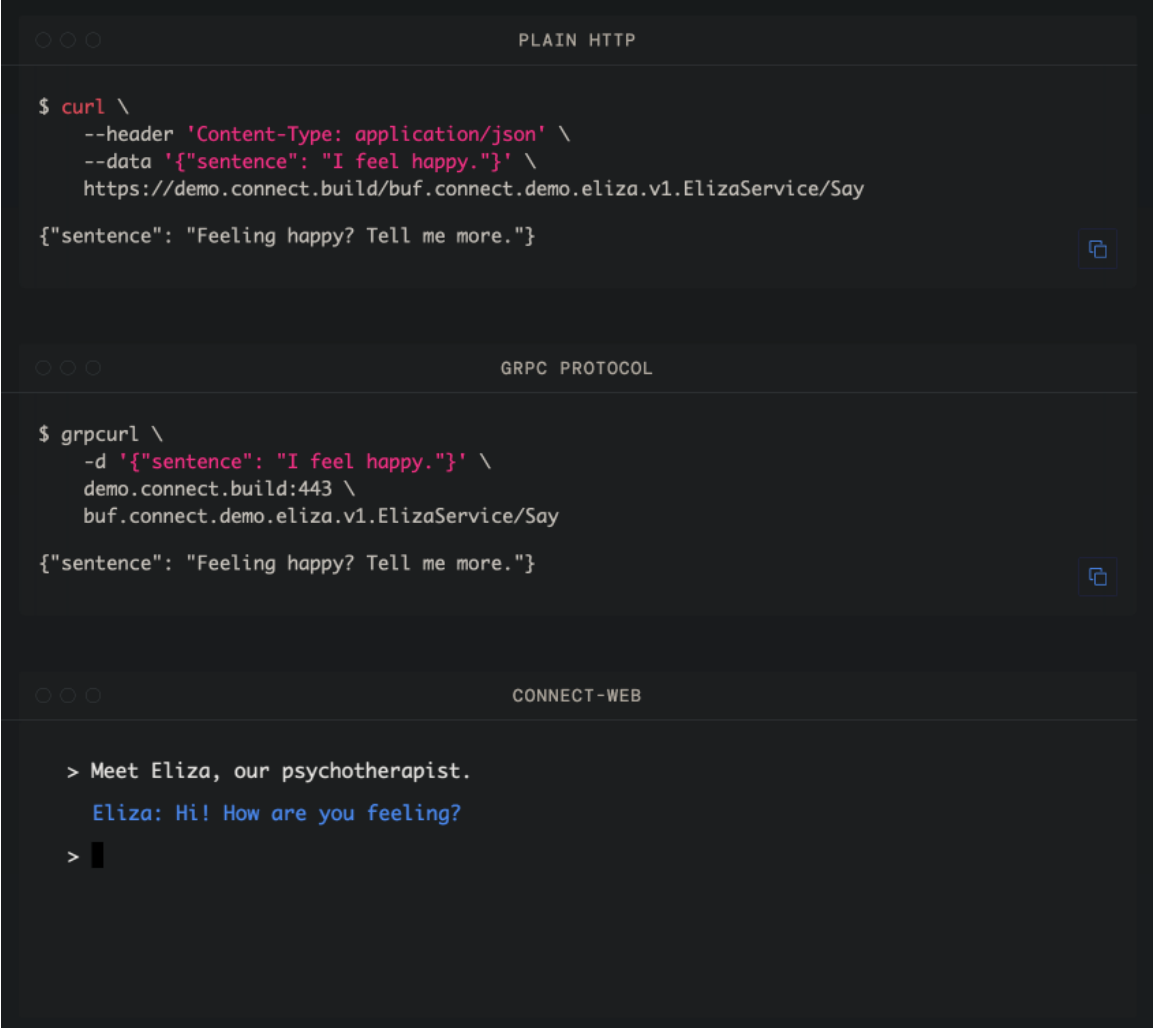
Example

```
import (  
    "google.golang.org/grpc/status"  
    "google.golang.org/grpc/codes"  
)  
  
func ListTasks(..) (*pb.ListTasksResponse, error) { // Any gRPC handler  
    return nil, status.Error(codes.NotFound, "user was not found")  
}
```

Connect-Go

Features

- Fully compatible with gRPC
- Uses Go standard library for networking
- Includes a REST-like curl friendly protocol



```
PLAIN HTTP

$ curl \
  --header 'Content-Type: application/json' \
  --data '{"sentence": "I feel happy."}' \
  https://demo.connect.build/buf.connect.demo.eliza.v1.ElizaService/Say

{"sentence": "Feeling happy? Tell me more."}

GRPC PROTOCOL

$ grpcurl \
  -d '{"sentence": "I feel happy."}' \
  demo.connect.build:443 \
  buf.connect.demo.eliza.v1.ElizaService/Say

{"sentence": "Feeling happy? Tell me more."}

CONNECT-WEB

> Meet Eliza, our psychotherapist.
  Eliza: Hi! How are you feeling?
> 
```


Demonstration

Exercise 3

Go to

training.brainhive.nl