GOROOT and GOPATH

- When you install Go on your system, it creates a directory `/usr/local/go` in UNIX or `c:/go` in Windows. Then it copies all necessary code and binaries needed for Go to function in this directory.

- GOROOT env var points just there it is the path where the go installation lives, alongside with all the standard library, usually you don’t need to change it

- If somehow, your Go installation directory is different than what Go assumes where it would be, you can change GOROOT environment variable.

- GOPATH is for all other dependent projects not in the standard library, GOPATH/src for the project source , GOPATH/bin is for executable outside the standard go distribution

- The default location of `$GOPATH` is `$HOME/go`, and it's not usually necessary to set `GOPATH` explicitly.

Managing a Project in Go - workspaces reminder

- the legacy way of organising a project in go is called a workspace
- Prior to go 1.13 more accurately 1.11 go used workspaces in order to gather all the files necessary for a project
- Woking in workspace mode means that all of our project related files needs to be in one place pointed by GOPATH environment variable
- When we used to install any dependency packages using go get command, it saves the package files under $GOPATH/src path
- Every dependency must be retrieved using go get prior of using it

Managing a Project in Go - workspaces reminder

- When you download a package using `go get`, **you are are cloning the master repository at the most recent commit**
- you can't install multiple versions of the same dependency package
- A Go program couldn't import a dependency unless it is present inside $GOPATH
- Also, go build command creates **binary executable files** and **package archives** inside $GOPATH. Hence, **GOPATH** used to be a big deal in Go The downside here is that we need to create all of our projects in one directory, or constantly change our GOPATH
- pkg is where go installs the compiled binaries of third party dependencies, Whenever you install a 3rd party package, a package object package-name.a is created for that package in this directory.

Introducing Go Modules

- Go modules gives us a built-in dependency management system
- As of go 1.13 module are the official way to organise our source code as a project
- A module is nothing but a directory containing Go packages. These can be distribution packages or executable packages.
- *A module can also be treated as a package that contains nested packages.*
- A module is also like a package that you can share with other people
- Modules allows us to work from any directory, not just **GOPATH** which gives us the flexibility to relocate our source code anywhere.

Introducing Go Modules

- Go modules are stored inside **$GOPATH/pkg/mod/cache** directory (***module cache directory***). (Seems like we haven't been able to get rid of $GOPATH at all. But Go has to **cache** modules somewhere on the system to prevent repeated downloading of the same modules of the same versions).
- Modules allows us to install the precise version of a dependency package to avoid breaking changes.
- Using modules we are able to import multiple versions of the same dependency package
- Like package.json in **NPM**, each module has go.mod file that the dependencies of our project
- If we do -> import **"github.com/username/packagename"**
  In the above package import statement, the **package name** looks like a **URL**, because it is. Go can download and install packages located anywhere on the internet.

Creating Go Modules

- Unlike Workspaces when using modules we don't need to start under a folder structure (src,pkg and bin directories)
- We start from an empty directory and executing the following

  $ go mod init github.com/gameserver

*As you can see we passed to init a module identifier

```
module github.com/gameserver

go 1.14
```

Retrieving Dependencies

- Often we will need to use third party libraries
- As we did before with workspaces , we can run the following to fetch a dependent lib

  $ go get github.com/gorilla/mux

- Please note the indirect comment, this is due to the fact that we are listing the module but not using it.
- Once we use the mux `router the indirect Comment will go away stating it knows that We Are using the dependency

```
module github.com/gameserver

go 1.14

require (
   github.com/gorilla/mux v1.7.4 // indirect
)
```

Retrieving Dependencies

- we don't see the gorilla/mux on our path like we used to when using workspaces

- Go download and caches it for us, (it is located under the **$GOPATH/pkg/mod/cache)**

Listing Dependencies

- Occasionally you will need to know what dependencies your module relies on
- If we invoke the $ go list all command we will be able to see all the packages that our module relies on
- But if we wish to only get only the modules we can execute $ go list -m a
- We can also retrieve all the versions a module has by executing
  $ go list -m -versions github.com/gorilla/mux

How go Verifies Dependencies

- When we retrieve a dependency , go creates another file , go.sum
- When a dependency is being retrieved go created a unique hash and assign it to the dependency
- The hash is unique based on the source code of the specific version that was retrieved
- The main reason for this is to make sure that the dependent module was not tempered in any way
- To validate that the modules an app contain has not been modified we can issue the following command $ go mod verify

Cleaning unused Dependencies

- Sometimes we no longer needs a dependency, and wish to remove it so that our app will be as lean as possible

- go has no way of knowing if we stopped using a dependency unless it rebuilds the entire dependency graph, which is a heavy operation

- $ go mod tidy will do just that for us

Semantic Versioning With Modules

- Semantic versioning is applied to go system in a very strict way
- An example for a version number -> v1.6.8
- The v letter is a must and it indicates that we are talking on a version
- The first digit “1” after the v indicate a major revision , when moving to a higher number it does not guarantee backward compatibility, but any version starting with that same should be backward compatible
- The second digit after the dot “6” indicated minor revision, what that tells us is that the module has new features but backward compatibility remains
- The last digit after the last point “8” is the patch, it promise not to introduce new features but only to fix bugs

Versioning rules for go modules

- All versions prior to v1 do not have to preserve backward compatibility
- Backward compatibility should be preserved within major versions
- Each Major version should pass version 1 have a unique path with suffix that entails the version for example:

  import github.com/gorilla/mux/v2

- The reason for this is to allow us to upgrade version in a gradually measured way

Organizing major module versions

- Usually we will use the master branch for the main v1 of our module
- Every additional major revision can be added to a different branch
- We need to remember to add tag to this new created branch to mark the new version
- Also we should update the module name with the suffix for it's new version for example github.com/example/v3

```
module github.com/gameserver/v3
go 1.14
```

- We can use more than one major revision of a module in our project provided that we've added an alias in the import section

```
import (
      "rsc.io/quote"
   quotev3 "rsc.io/quote/v3"
)
```

Importing unversioned modules

- What happen when we need to use a module that do not have any major.minor.patch semver ?

- The simple answer is that the go tooling will fix it in a way that we will be able to import as much as we did before with semver

- The long answer is that the go tooling will create a prerelease semver based on the latest commit hash and timestamp for example if we import golang.org/x/tools we will have in our go.mod file:

```
golang.org/x/tools v0.0.0-20200622192924-4fd1c64487bf
```

Requesting a Specific Module

- The module system allows us to request a specific module by using the @ sign for example $ go get github.com/gorilla/mux@v1.6.1

- We could also ask for a version with the following :
  - Version prefix : just use for example @v1 and it will fetch the latest version with this major revision
  - @latest : will fetch the latest version from the repo
  - Specific commit : for example @c55772
  - @master : will fetch the head of master , the latest commit
  - Comparison operator: @>=1.5.0 , we are asking for a version greater or equal to 1.5.0,always remember that the closest match wins

Go mod additional commands

- $ go mod why github.com/rsc.io/quote this command build a dependency tree in order to show us the modules that use it and all the modules required by it

- $ go mod graph will show you a tree of all the modules in your app and their dependent modules

- If you need to point to the local version of a dependency in Go rather than the one over the web, use the edit option with **replace** argument

Go mod additional commands

- $ go mod download github.com/rsc.io/quote is used to download the module into the build cache so it will be available for later builds

- go mod vendor this command will create a vendor folder and will hold all the dependencies there, it might be useful for organizations that do not want production build server to retrieve the dependency from the internet

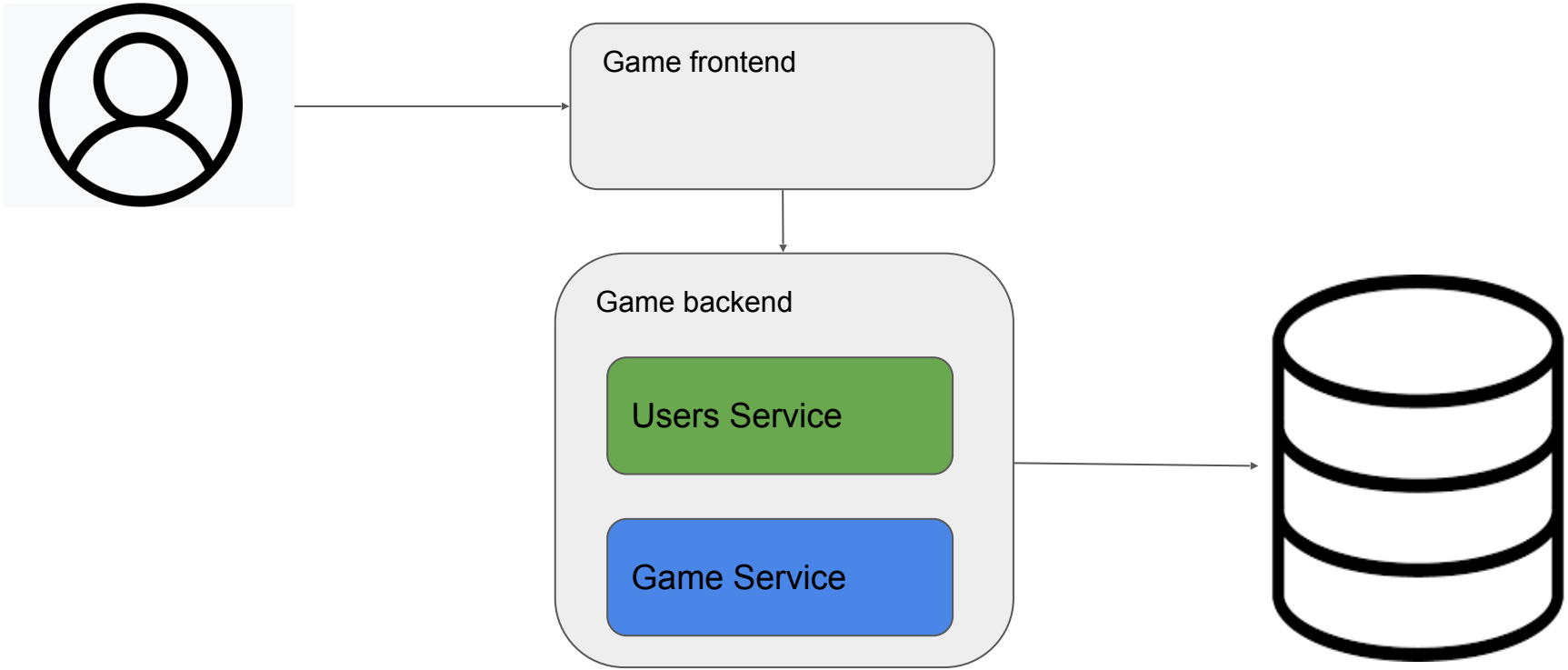
- To use the vendor directory you need to do $ go run -mod vendor . please note that all dependencies must be in the vendor directory , go will not mix it with the what is in the cache

Go Project Structure

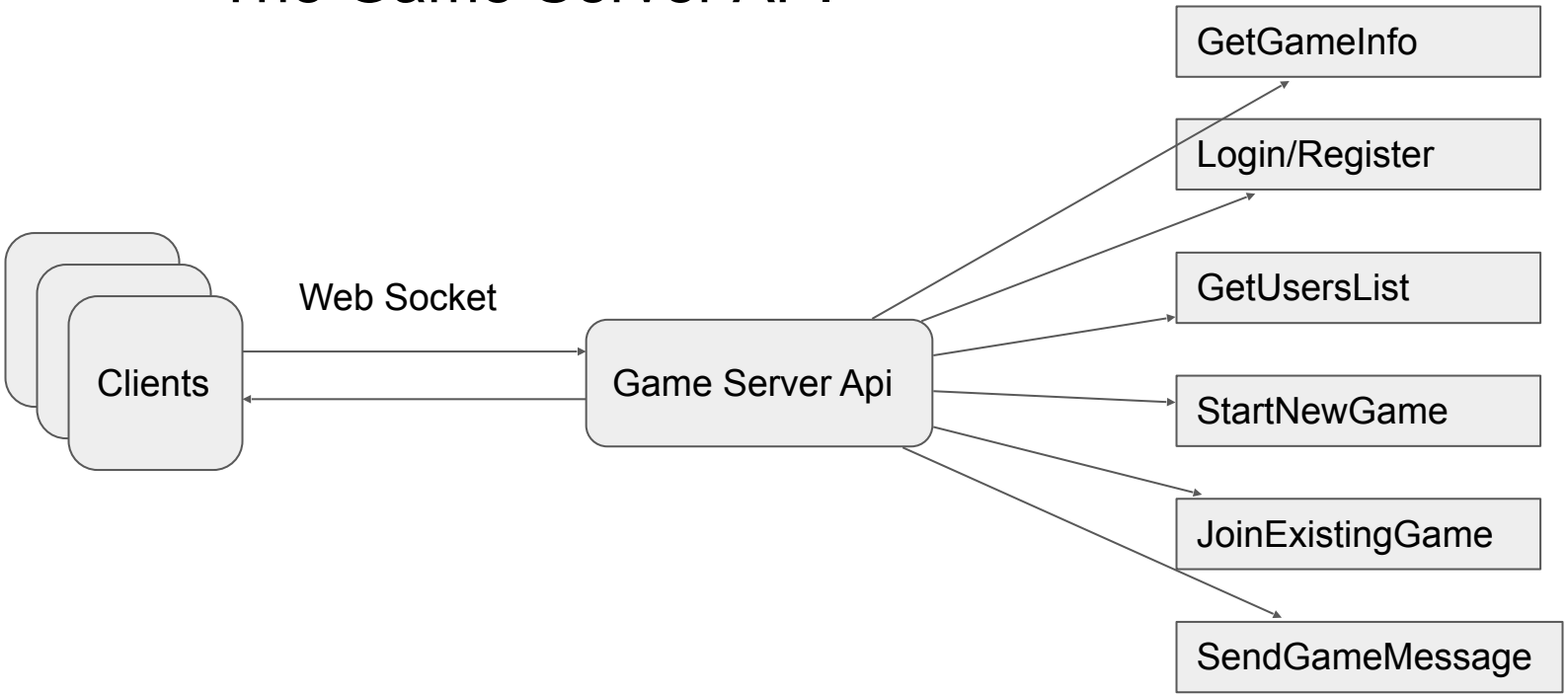
- As your project grows keep in mind that it'll be important to make sure your code is well structured ,otherwise you'll end up with a messy code with lots of hidden dependencies and global state.
- The following is just a recommendation for a basic layout for Go application projects, based on a set of common historical and emerging project layout patterns in the Go ecosystem
- GO Directories
    - **/cmd** - the main application go file should be placed in this directory, if you have other executables each should be placed in a sub directory for example /cmd/myutil
    - **/internal** - Private application and library code. This is the code you don't want others importing in their applications or libraries.
    - /**pkg** - Library code that's ok to use by external applications (e.g., `/pkg/mypubliclib`). Other projects will import these libraries expecting them to work, so think twice before you put something here :-)
    - /**vendor** - Application dependencies (managed manually or by your favorite dependency management tool like the new built-in `Go Modules` feature)
    - /**web** - Web application specific components: static web assets, server side templates and SPAs.
- And more … https://github.com/golang-standards/project-layout

Demo - Gameserver App

The Game Server App

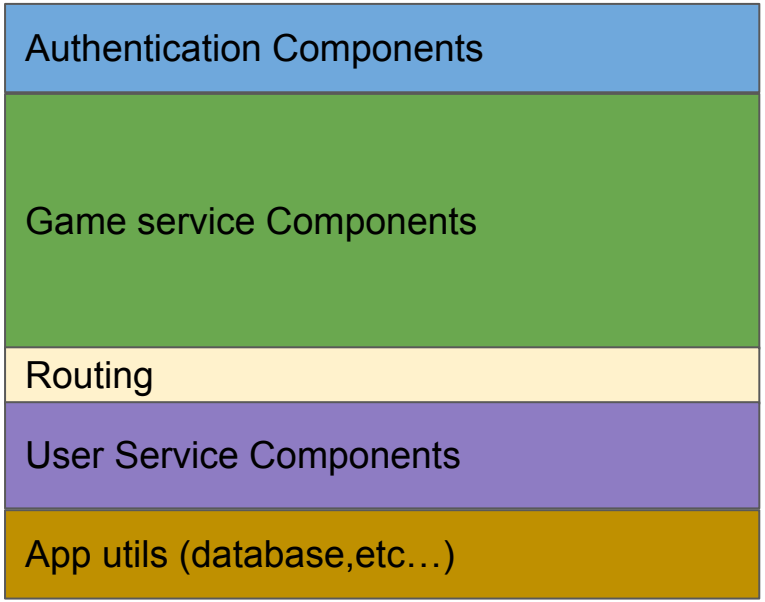


The Game Server API



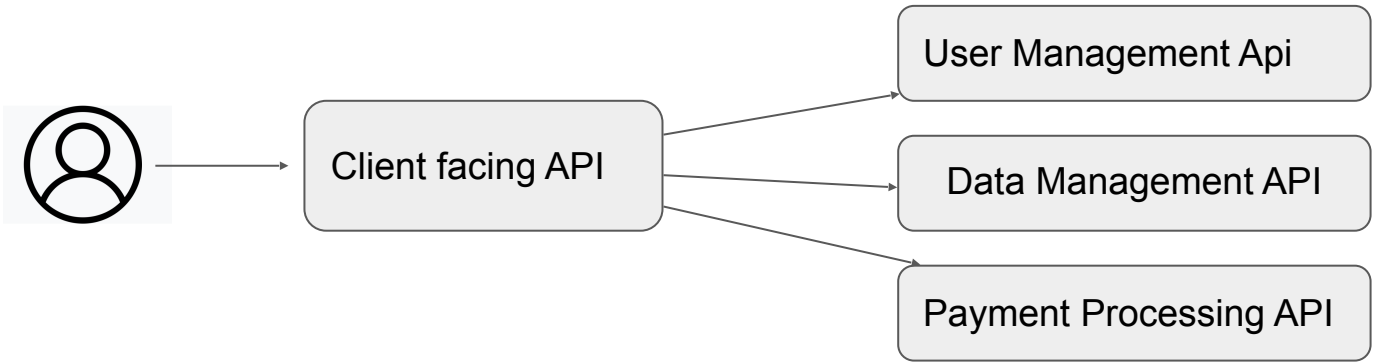
```
├── Dockerfile
├── README.md
├── cmd
│   ├── keys
│   │   ├── app.rsa
│   │   └── app.rsa.pub
│   └── main.go
├── deployments
│   ├── config.yaml
│   ├── database.yaml
│   └── gameserver.yaml
├── go.mod
├── go.sum
└── internal
    ├── auth
    │   ├── auth.go
    │   └── token.go
    ├── games
    │   ├── game.go
    │   ├── gamemanager.go
    │   ├── gamemessages.go
    │   ├── gamesession.go
    │   ├── gamesutils.go
    │   ├── player.go
    │   ├── service
    │   │   └── game.service.go
    │   └── utils
    ├── routes
    │   └── routes.go
    ├── users
    │   ├── service
    │   │   └── user.service.go
    │   └── user.go
    └── utils
        ├── db.go
        └── exception.go
```

Game Server Components



Building Web Services With GO

- Web services are are special type of programs exposing an API for consumers clients
- Web services enable a separation of concerns between different layer of our application
- While web services can provide data in a number of different formats, XML and JSON are the most common



Rest - Representational state transfer

- Restful services goes hand to hand with http , for all requests and response
- Restful api are stateless, the service doesn’t have to remember anything about the caller
- Restful services expose their api using uri’s called endpoints
- Each endpoint uniquely maps to a resource , enabling to retrieve or modify its state, for example :
    - example.com/authors - return all the Authors
    - example.com/author/1 - return a specific author

Rest - Requests

- The request http method dictate that action we wish to perform on the resource exposed by the api, most common are get, put, post and delete
- The request body is optional and most commonly contains json formatted data
- The requests can also include metadata in the for of http headers , this can be useful for example for sending authentication tokens,etc…

Rest Responses

- Every http response contains a status code , the status code is used by the restful web service to indicate whether the request was successfully processed

2xx: Success	Indicates that the client’s request was accepted successfully.
3xx: Redirection	Indicates that the client must take some additional action in order to complete their request.
4xx: Client Error	This category of error status codes points the finger at clients.
5xx: Server Error	The server takes responsibility for these error status codes.

- Response can also contain a Body with data containing the requested resource
- Response can also contain headers with metadata for example , the content length of the data ,cookies ,etc...

Handling Http Requests

The go standard library provides us the net/http package for all kind of http operations

Using the net/http package there are two main ways for handling http requests

1. Implementing the handler interface and passing it to the http.Handle function the handler interface implements the handle function to match a specific pattern
2. Using the http.handleFunc and passing it a function that is implemented for a specific request pattern

Handling Http Requests - Using Handler example

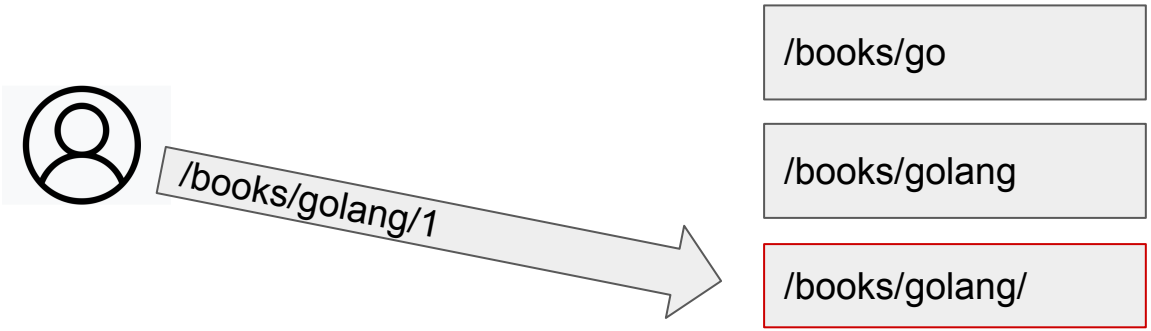
```
package main

import "net/http"

//MyHandler type
type MyHandler struct{}

func (handler MyHandler) ServeHTTP(rs http.ResponseWriter, rq *http.Request) {
   rs.Write([]byte("Hello world"))
}

func main() {

   var myhandler MyHandler
   http.Handle("/books", myhandler)
   http.ListenAndServe(":5000", nil)

}
```

Handling Http Requests - Using Handlerfunc

```
package main

import "net/http"

func main() {

   http.HandleFunc("/books", func(rs http.ResponseWriter, rq *http.Request) {
       rs.Write([]byte("Hello world from handler func"))
   })
   http.ListenAndServe(":5000", nil)

}
```

Http ServeMux

- A ServeMux is just an http handler that takes a url and match it’s pattern to registered handlers
- You have probably noticed that we’ve used `http.ListenAndServe(":5000", nil)` this indicates that we want to use the default http ServeMux(we can also use our own)
- We could also use the https version `http.ListenAndServeTLS(string,certfile,keyfile,string,handler)`
- It will always try to find the best match for the url from the list of registered patterns for example:



Working With Json

- The primary data format that is being used by web apps to communicate with restful services is probably json.
- Go provide us the easiest way for working with json data format by using the encoding/json standard library
- There are two main ways for working with json
  - Using the Marshal and Unmarshal functions of the json package
  - Using the Encoding and Decoding functions of the json package

Json.Marshall

- Json Marshal takes interface{} as parameter which means it can get any type,
- marshaller use reflection to understand the real type
- When marshalling structs only the exported types will be marshalled
- You can use string tags to require different names and options to marshall with
- json.MarshallIndent allows for pretty-printing of nested structures

```go
package main

import (
   "encoding/json"
   "fmt"
   "log"
)

// Student type
type Student struct {
   fullName string
   Name     string
   Age      int    `json:"age"`
   City     string `json:"city"`
}

func main() {

   student := Student{
       Name: "Dave",
       Age:  37,
       City: "Denver",
   }
   d, err := json.Marshal(&student)
   if err != nil {
       log.Fatalf("json.Marshal failed with '%s'\n", err)
   }
   fmt.Printf("Student in compact JSON: %s\n", string(d))

   d, err = json.MarshalIndent(student, "", "  ")
   if err != nil {
       log.Fatalf("json.MarshalIndent failed with '%s'\n", err)
   }
   fmt.Printf("student in pretty-printed JSON:\n%s\n", string(d))

}
```

Json Unmarshal

- When unmarshalling we have to pass a pointer to a struct, else we will get an error
- Notice that although we did not provide accurate mapping in the City element the json decoder is smart enough to unmarshall the city field
- Field Name shows that JSON decoder can also automatically decode into a pointer to a value.

```
package main

import (
   "encoding/json"
   "fmt"
   "log"
)

//Student type
type Student struct {
   Name  *string `json:"name"`
   Age   int     `json:"age"`
   City  string
   Major string
}

var jsonStr = `{
   "name": "Dave",
   "age": 37,
   "city": "denver"
}`

func main() {

   var student Student
   err := json.Unmarshal([]byte(jsonStr), &student)
   if err != nil {
       log.Fatalf("json.Unmarshal failed with '%s'\n", err)
   }
   fmt.Printf("Student struct parsed from JSON: %#v\n", student)
   fmt.Printf("Name: %#v\n", *student.Name)

}
```

Json Encode And Decode

We can decode and encode JSON data from a file on disk or,Network connection etc, actually we can decode from , any io.Reader.

```go
package main

import (
   "encoding/json"
   "fmt"
   "log"
   "os"
)
//Book type
type Book struct {
   Title     string `json:"title"`
   Subtitle  string `json:"subtitle"`
   Author    string `json:"author"`
   Published string `json:"published"`
}
func main() {
   file, err := os.Open("../books.json")
   defer file.Close()
   if err != nil {
       log.Fatalf("an error occured openinng the file")
   }
   var books []Book
   err = json.NewDecoder(file).Decode(&books)
   if err != nil {
       log.Fatalf("couldn't decode file into json")
   }
   for _, book := range books {
       fmt.Println(book.Title, " ", book.Author)
   }
   bookNewfile, err := os.Create("booksnew.json")
   defer file.Close()
   if err != nil {
       log.Fatalf("couldnt create file")
   }
   json.NewEncoder(bookNewfile).Encode(books)
}
```

Custom Json Marshaling and UnMarshaling

- Sometimes we need to control the format of the json data when ,marshaling and unmarshaling
- For example what is we need to change the way time is formated
- We can control the marshaling and unmarshaling by implementing the json.Marshaler interface by the requested type

```
type customTime time.Time

const customFormatUnmarshal = `"2006-01-02T15:04:05Z07:00"`
const customFormatMarshal = `"Monday, 02-Jan-06 15:04:05 MST"`

//Book type
type Book struct {
   Title     string     `json:"title"`
   Subtitle  string     `json:"subtitle"`
   Author    string     `json:"author"`
   Published customTime `json:"published"`
}

func (ct customTime) MarshalJSON() ([]byte, error) {
   t := time.Time(ct)
   s := t.Format(customFormatMarshal)
   return []byte(s), nil
}

func (ct *customTime) UnmarshalJSON(d []byte) error {
   t, err := time.Parse(customFormatUnmarshal, string(d))
   if err != nil {
       return err
   }
   *ct = customTime(t)
   return nil
}
```

Controlling json serialization

- We can hide specific fields from json structs by using a first lowercase letter in the field or tagging json field with the “-” letter for example

```
type Book struct {
    Title     string `json:"title"`
    Subtitle  string `json:"subtitle"`
    Author    string `json:"author"`
    Published string `json:"-"`
    isbn      string
}
```

- We can also omit fields when they are empty by tagging them as in the following example

```
type Book struct {
    Title     string `json:"title"`
    Subtitle  string `json:"subtitle,omitempty"`
    Author    string `json:"author"`
    Published string `json:"-"`
    isbn      string
}
```

So What method should we use for serializing json ?

The Simple rule is as follows:

- Use json.Decoder if your data is coming from an io.Reader stream, or you need to decode multiple values from a stream of data.

- Use json.Unmarshal if you already have the JSON data in memory.

Inroducing Http Request object

All http requests has a method property usually describing the restful operation context

The Headers is a map of strings that gives us access to the headers of the requests

And the Body which is of type io.ReadCloser is the actual data received which could be a json formatted data

Handling Http Requests

- The ServerMux just match a url pattern to a handler
- We could have one handler handling all the CRUD operations for a specific Entity
- When the client sends a request , we are responsible to handle it based on the parameters sent
- Please note that we need to extract the parameters passed in the url to be able to respond correctly, for example, for books/123 we will have to extract the “123” as the id of the book, and combined with request.HttpMethod take the right action

Lets see a simple restful service to handles books ...

MiddleWare

- Occasionally we need to add code that is intended for every incoming requests
- This code is not necessary tied to the logic in our application but serve as a more general code that is intended to reduce duplication and can help us with the following
- For example , check for user authentication, and access control ,add logging information for the requests, load user profile information etc…
- In Middleware we can execute functionality before or after the http handler are called

examining the http.HandlerFunc

The http.HandlerFunc is a function type :

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

it implements the interface HttpHandler

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

We can use this type as an http.HttpHandler adapter , this will allow us to use functions as handlers, by having those functions be of type HttpHandler

```
http.Handle("/books/", http.HandlerFunc(handleBook))
```

This happens due to the type conversion in the go language, we do a type conversion to the handleBook that has the same signature of HandlerFunc

Middleware example

1. In the following example We have myMiddleware function that takes httphandler and returns httphandler it basically wrapps the handler passed in

1. The returned function is converted to a handler (happens automatically by using the http.HandleFunc),
2. Inside the function implementation we call the ServeHttp method which is the one invoking the middleware chain

1. Remember that The implementation of the http.HandlerFunc calls the actual function

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

```
func myMiddleware(handler http.Handler) http.Handler {
    return http.HandlerFunc(func(rs http.ResponseWriter,
                                        rq *http.Request) {
        fmt.Println("called before")
        handler.ServeHTTP(rs, rq)
        fmt.Println("called after")
    })
}

func main() {

    booksHandler := http.HandlerFunc(handleBooks)
    http.Handle("/books", myMiddleware(booksHandler))
    http.HandleFunc("/books/", handleBook)

    if err := http.ListenAndServe(":5000", nil); err != nil {
        log.Fatalf(err.Error())
    }
}
```

Using the Default ServerMux , not the easiest way...

- As you saw before ,we pass to the http.ListenAndServe nill as a third parameter which is the default ServerMux
- The default ServerMux leaves us the responsibility of parsing the url parameters, and checking the Request method in the handlers
- We need to find a better way to save us the boilerplate code for each new service

Simplify routing with the gorilla/mux package

- It implements the `http.Handler` interface so it is compatible with the standard `http.ServeMux`.

```
func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", HomeHandler)
    r.HandleFunc("/products", ProductsHandler)
    r.HandleFunc("/articles", ArticlesHandler)
    http.Handle("/", r)
}
```

- Paths can have variables. They are defined using the format `{name}` or `{name:pattern}`. If a regular expression pattern is not defined, the matched variable will be anything until the next slash. For example:
- URL hosts, paths and query values can have variables with an optional regular expression.

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

The names are used to create a map of route variables which can be retrieved calling `mux.Vars()`:

```
func ArticlesCategoryHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "Category: %v\n", vars["category"])
}
```

Simplify routing with the gorilla/mux package

- Middleware can be easily added with a .Use() method similar to how the node express framework works with middleware

```
r := mux.NewRouter()
r.HandleFunc("/", handler)
amw := authenticationMiddleware{}
amw.Populate()
r.Use(amw.Middleware)
```

There are much more features and options at: https://github.com/gorilla/mux

Jwt Authentication

- JSON web tokens is currently one of the most popular ways to build authentication api
- JWT is a token system that allows us sending a token instead of sending the username and password with every request
- From jwt.io : *“Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.”*

For more information visit : jwt.io/introduction/

Lets see an example for a jwt middleware...

Demo - Jwt Authentication

Reading Configuration files And Environment variables

- It is considered best practice to store app configuration data and sensitive data outside of the application
- Sensitive data like DB connection credentials usually will be environment specific and will be set in the environment using env vars or files
- Other configuration data come very handy when we wish to affect the behaviour of our app based for example of the environment it is running on like development/production
- Also some apps should act in a dynamic way and change based on configuration, like for example nginx .conf file
- It is required to keep the app configuration outside of the code , else we will not be able to change the app behaviour after it is built

Go support libraries for configuration - 1. godotenv

- The easiest way to load the .env file is using godotenv package.
  godotenv provides a Load method to load the env files.

  // Load the .env file in the current directory
  godotenv.Load()

  // or
  godotenv.Load(".env")

```
func goDotEnvVariable(key string) string {

   // load .env file
   err := godotenv.Load(".env")
    if err != nil {
     log.Fatalf("Error loading .env file")
   }
    return os.Getenv(key)
}
```

- The problem with godot env that it does not load the
  environment variables automatically if they are not found in the env file

Go support libraries for configuration - 2. Viper

From viper :

“*Viper is a complete configuration solution for Go applications including 12-Factor apps. It is designed to work within an application, and can handle all types of configuration needs and formats. It supports:*

- *setting defaults*
- *reading from JSON, TOML, YAML, HCL, envfile and Java properties config files*
- *live watching and re-reading of config files (optional)*
- *reading from environment variables*
- *reading from remote config systems (etcd or Consul), and watching changes*
- *reading from command line flags*
- *reading from buffer*
- *setting explicit values*

Viper can be thought of as a registry for all of your applications configuration needs.”

Reading Configuration with viper

- viper.SetConfigName("config") → Provides the configuration file name.
- viper.SetConfigPath(".") → Provides the path in which viper needs to search for the configuration file.
- viper.AutomaticEnv() → Tells viper to look at the Environment Variables.
- viper.ReadInConfig() → Reads all the configuration variables.
- viper.Get("variable_name") → Returns the value of the "variable name" variable from environment configurations first and if it is not available, it reads from the configuration file.
- viper.SetDefault("database.dbname", "test_db") → Sets a default value

```go
package main

import (
    "fmt"
    "github.com/spf13/viper"
    corev1 "k8s.io/api/core/v1"
)
func main() {
    // Set the file name of the configurations file
    viper.SetConfigName("config")
    // Set the path to look for the configurations file
    viper.AddConfigPath(".")
    // Enable VIPER to read Environment Variables
    viper.AutomaticEnv()
    viper.SetConfigType("yml")
    var configuration corev1.Secret

    if err := viper.ReadInConfig(); err != nil {
        fmt.Printf("Error reading config file, %s", err)
    }
    // Set undefined variables
    viper.SetDefault("database.dbname", "test_db")
    err := viper.Unmarshal(&configuration)
    if err != nil {
        fmt.Printf("Unable to decode into struct, %v", err)
    }
    // Reading variables using the model
    fmt.Println("Reading variables using the model..")
    fmt.Println("username are\t", configuration.StringData["username"])
    fmt.Println("password is\t\t", configuration.StringData["password"])
    // Reading variables without using the model
    fmt.Println("\nReading variables without using the model..")
    fmt.Println("Database is\t", viper.GetString("database.dbname"))
    fmt.Println("EXAMPLE_PATH is\t", viper.GetString("EXAMPLE_PATH"))
    fmt.Println("EXAMPLE_VAR is\t", viper.GetString("EXAMPLE_VAR"))
}
```

Persisting data into a database

- The go standard library provides us the sql package for interacting with a database
- The open function enables us to open a connection to a database , this function is passed a the driver for the database and the datasource , which is the connection string to the db

```
db, err := sql.Open("mysql", "username:password@tcp(127.0.0.1:3306)/mydb")
```

- The function returns a DB type object which managed the connections to the database within a connection pool
- The DB will open and close connections on demand and will do it on a thread safe way
- Please note that we also have the same method with same signature except that it returns a single connection “Conn type”

Connecting to a Database

- Let's start by running a db using a docker image

  $ docker run -p 3306:3306 -p 33060:33060 --name mysqldb -v ~/mysql:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=pass -d mysq

- Please note that beside the sql import package we also need to import the relevant database driver
- On the following example we use mysql driver
- Also note that we are using the "_" to allow the driver to be imported although there is no direct use of it from our code
- See https://golang.org/s/sqldrivers for a list of drivers.

```go
package main
import (
    "database/sql"
    "log"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:pass@(127.0.0.1)/game_db")
    if err != nil {
        log.Fatalf("couldn't open database connection")
    }
}
```

Query the database

- The sql package exports the Query function

  func (db *DB) Query(query string, args ...interface{}) (*Rows, error)

- It takes a string for the sql and a bunch of parameters that can optionally be passed to the query
- It returns a Rows type which allows us to iterate by using it’s next method
- During iteration on the rows we need to call scan in order to get the data into our go types

Query database example

Please note that we didn’t use the “*”

In the select statement , it is important to match the the fields to the number of values returned from the query in the order we get them

```
package main

import (
   "database/sql"
   "fmt"
   "log"

   _ "github.com/go-sql-driver/mysql"
)

type User struct {
   id         int
   first_name string
   last_name  string
   email      string
   password   string
}

var users []User = make([]User, 0)

func main() {
   db, err := sql.Open("mysql", "root:pass@(127.0.0.1)/game_db")
   if err != nil {
       log.Fatalf("couldnt open database connection")
   }

   rows, err := db.Query("select id,first_name , last_name, email , password  from users")

   for rows.Next() {
       var user User
       rows.Scan(&user.id, &user.first_name, &user.last_name, &user.email, &user.password)
       users = append(users, user)
   }
   fmt.Println(users)
}
```

Query With Parameters

We can also use QueryRow to fetch a single row , if the result of the query returns multiple rows , only the first one will be returned

```
package main

import (
   "database/sql"
   "fmt"
   "log"

   _ "github.com/go-sql-driver/mysql"
)
type User struct {
   id         int
   first_name string
   last_name  string
   email      string
   password   string
}

func main() {
   db, err := sql.Open("mysql", "root:pass@(127.0.0.1)/game_db")
   if err != nil {
       log.Fatalf("couldnt open database connection")
   }

   name := "dave"
   row := db.QueryRow("select id,first_name , last_name, email , password  from users where
first_name =?", name)

   var user User
   err = row.Scan(&user.id, &user.first_name, &user.last_name, &user.email, &user.password)
   if err == sql.ErrNoRows {
       log.Fatalf("couldnt find what we are looking for")
   } else if err != nil {
       log.Fatalf("couldnt find what we are looking for", err.Error())
   }
   fmt.Println(user)
}
```

Executing Database Commands

- DB.Exec method enables us to execute commands like insert update and delete ,on the database func (db *DB) Exec(query string, args ...interface{}) (Result, error)
- This function return a Result interface that has two methods
- LastInsertId(...) that is used to indicate the last inserted row id
- RowsAffected(...) that returns the number of rows that were changed as a result for example of an update, insert or delete operation

```
type Result interface {
    // LastInsertId returns the integer generated by the database
    // in response to a command. Typically this will be from an
    // "auto increment" column when inserting a new row. Not all
    // databases support this feature, and the syntax of such
    // statements varies.
    LastInsertId() (int64, error)

    // RowsAffected returns the number of rows affected by an
    // update, insert, or delete. Not every database or database
    // driver may support this.
    RowsAffected() (int64, error)
}
```

sql.Exec example

```go
package main

import (
   "database/sql"
   "fmt"
   "log"

   _ "github.com/go-sql-driver/mysql"
)
type User struct {
   id         int
   first_name string
   last_name  string
   email      string
   password   string
}
var users []User = make([]User, 0)

func main() {
   db, err := sql.Open("mysql", "root:pass@(127.0.0.1)/game_db")
   if err != nil {
       log.Fatalf("couldnt open database connection")
   }
   id := 1
   result, err := db.Exec("update users set last_name =? where id = ?", "axel", id)

   if num, err := result.RowsAffected(); err != nil {
       log.Fatalf("couldnt update database ", err.Error())
   } else {
       fmt.Println("numner of rows affected is ", num)
   }
   result, err = db.Exec("insert into users(first_name,last_name,email,password)values(?,?,?,?)", "jon", "smith", "son@gmail.com", "passpaass")
   if err != nil {
       log.Fatalf("couldnt insert into database ", err.Error())
   } else {
       if id, e := result.LastInsertId(); e != nil {
           fmt.Println("no rows affected")
       } else {
           fmt.Println("new row is ", id)
       }
   }
}
```

Database Connection management

- The DB type knows how to handle multiple connection to the database by opening and closing, and reusing connections based on the traffic
- The DB manage a pool of connections
- We can configure the number of connections in the pool for optimizing the number of concurrent connections from clients
    - db.SetConnMaxLifetime() - maximum amount of time the connection will be open 0 value means connections will never be closed
    - db.SetMaxIdleConns() - determines how many connection stay in the pool without being used
    - db.SetMaxOpenConns() - the number of open connections at any given time

when we exceeds the number of connections the new request is going to block, and the client will get a connection timeout … how can we handle this?

Accessing Database with context

- The prefered way is to use contexts for running operations against the database
- context allows us to control operations that might take too long to complete,
- So one way we can handle it is create a context object with a timeout that once exceeded we can can free the connection for other client to use
- In the db.Sql package there are equivalent functions to those we saw (query,exec,etc..) that take context as a first parameter

Accessing Database with context - sample

In order to use context , we need to start by creating one, for that we can use the context.Background function followed by a call to context.WithTimeout to get a context that is bound to a time

Please note that you have to remember to call defer cancel() so that the context will be closed and there will be no memory leak

```go
package main

import (
   "context"
   "database/sql"
   "fmt"
   "log"
   "os"
   "time"

   _ "github.com/go-sql-driver/mysql"
)

type User struct {
   id         int
   first_name string
   last_name  string
   email      string
   password   string
}

var users []User = make([]User, 0)

func main() {
   db, err := sql.Open("mysql", "root:pass@(127.0.0.1)/game_db")
   defer db.Close()
   if err != nil {
       log.Fatalf("couldnt open database connection")
   }
   ctx := context.Background()
   ctx,cancel := context.WithTimeout(ctx,time.Second * 15)
   defer cancel()

   rows, err := db.QueryContext(ctx,"select id,first_name , last_name, email , password  from users")
   defer rows.Close()

   for rows.Next() {
       var user User
       rows.Scan(&user.id, &user.first_name, &user.last_name, &user.email, &user.password)
       users = append(users, user)
   }
   fmt.Println(users)
}
```

Running operations with context

- So what is a Context ? from the formal definition in the docs :

  “*A Context carries deadlines, cancellation signals, and other request-scoped values across API boundaries and goroutines*”

- context.Context is a simple object that is meant to be transferred between a caller to a receiver, in order to allow more control on the requested operation
- Most use `context` with downstream operations, like making an HTTP call, or fetching data from a database, or while performing async operations with go-routines
- Another highly useful feature of `context` is it’s ability to cancel, or halt an operation mid-way
- For example a caller can decide to cancel a long running Http request and the receiver is responsible to abandon the operation once cancelled
- A context can also be set with a predefined time out that the receiver that gets the context should check periodically to decide whether to abandon the operation or not

Creating a Context

When calling Api's that requires a context, if you don't have one you can create one easily by calling context.Background() , or context,TODO() functions

context.Background() serves as the root of all context which will be derived from it

This empty context has no functionality at all and we can add functionality by deriving a new context from it using the following values

A derived context is can be created in 4 ways

- Passing request-scoped values - using **WithValue()** function of context package
- With cancellation signals - using **WithCancel()** function of context package
- With deadlines - using **WithDeadine()** function of context package
- With timeouts - using **WithTimeout()** function of context package

It's critically important that any cancel function returned from a With function is executed before that function returns. This is why the idiom is to use the defer keyword right after the With call, as you see on line 26. Not doing this will cause memory leaks in your program.

When to use contexts

- To pass data to downstream. Eg. a HTTP request creates a **request_id, request_user** which needs to be passed around to all downstream functions for distributed tracing.

- When you want to halt the operation in the midway – A HTTP request should be stopped because the client disconnected

- When you want to halt the operation within a specified time from start i.e with timeout – Eg- a HTTP request should be completed in 2 sec or else should be aborted.

- When you want to halt an operation before a certain time – Eg. A cron is running that needs to be aborted in 5 mins if not completed.

Using Context to for canceling http request

The following example issues an http request , if the request takes more then 5 seconds to complete it is being canceled

Server.go

```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("before sleep")
    time.Sleep(10 * time.Second)
    fmt.Println("after sleep")

    fmt.Fprintf(w, "Hi")
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":9191", nil))
}
```

client.go

```go
package main

import (
    "context"
    "fmt"
    "net/http"
    "time"
)

func main() {
    ctx, cancel := context.WithCancel(context.Background())

    go func() {
        fmt.Println("before request")
        client := &http.Client{Timeout: 30 * time.Second}
        req, err := http.NewRequest("GET", "http://127.0.0.1:9191", nil)
        if err != nil {
            panic(err)
        }
        req = req.WithContext(ctx)
        _, err = client.Do(req)
        if err != nil {
            panic(err)
        }
        fmt.Println("will not reach here")
    }()

    time.Sleep(5 * time.Second)
    cancel()
    fmt.Println("finished")
}
```

Using Context With Timeout

We can use context to give an operation a deadline so that after the specified amount of time the operation will be aborted

To do this we listen for incoming messages on the chanel that is returned from ctx.Done()

“The returned context's Done channel is closed when the deadline expires, when the returned cancel function is called, or when the parent context's Done channel is closed, whichever happens first”

```go
package main

import (
   "context"
   "fmt"
   "time"
)

func simulateLongOperation(ctx context.Context) (time.Time,error) {

   select {
   case <-ctx.Done():
       return time.Time{}, ctx.Err()
   case t := <-time.After(time.Millisecond * 100 ):
       return t, nil
   }
   return time.Time{},nil
}

func main() {
   ctx, _ := context.WithTimeout(context.Background(), time.Millisecond*20)
   t,err := simulateLongOperation(ctx)
   if err != nil{
       fmt.Println("Called simulateLongOperation() with 20ms ",  err.Error())
   }else{
       fmt.Println("operation was fiinished after 100 ms at",t)
   }
}
```

Passing values with context

Sometimes you need to pass data to downstream. For example : HTTP request that creates a **msgId** which needs to be passed around to all downstream functions.

```go
package main

import (
   "context"
   "net/http"
   "github.com/google/uuid"
)

func main() {
   helloWorldHandler := http.HandlerFunc(HelloWorld)
   http.Handle("/welcome", inejctMsgID(helloWorldHandler))
   http.ListenAndServe(":8080", nil)
}

//HelloWorld hellow world handler
func HelloWorld(w http.ResponseWriter, r *http.Request) {
   msgID := ""
   if m := r.Context().Value("msgId"); m != nil {
       if value, ok := m.(string); ok {
           msgID = value
       }
   }
   w.Header().Add("msgId", msgID)
   w.Write([]byte("Hello, world"))
}

func inejctMsgID(next http.Handler) http.Handler {
   return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
       msgID := uuid.New().String()
       ctx := context.WithValue(r.Context(), "msgId", msgID)
       req := r.WithContext(ctx)
       next.ServeHTTP(w, req)

   })
}
```

Testing http service

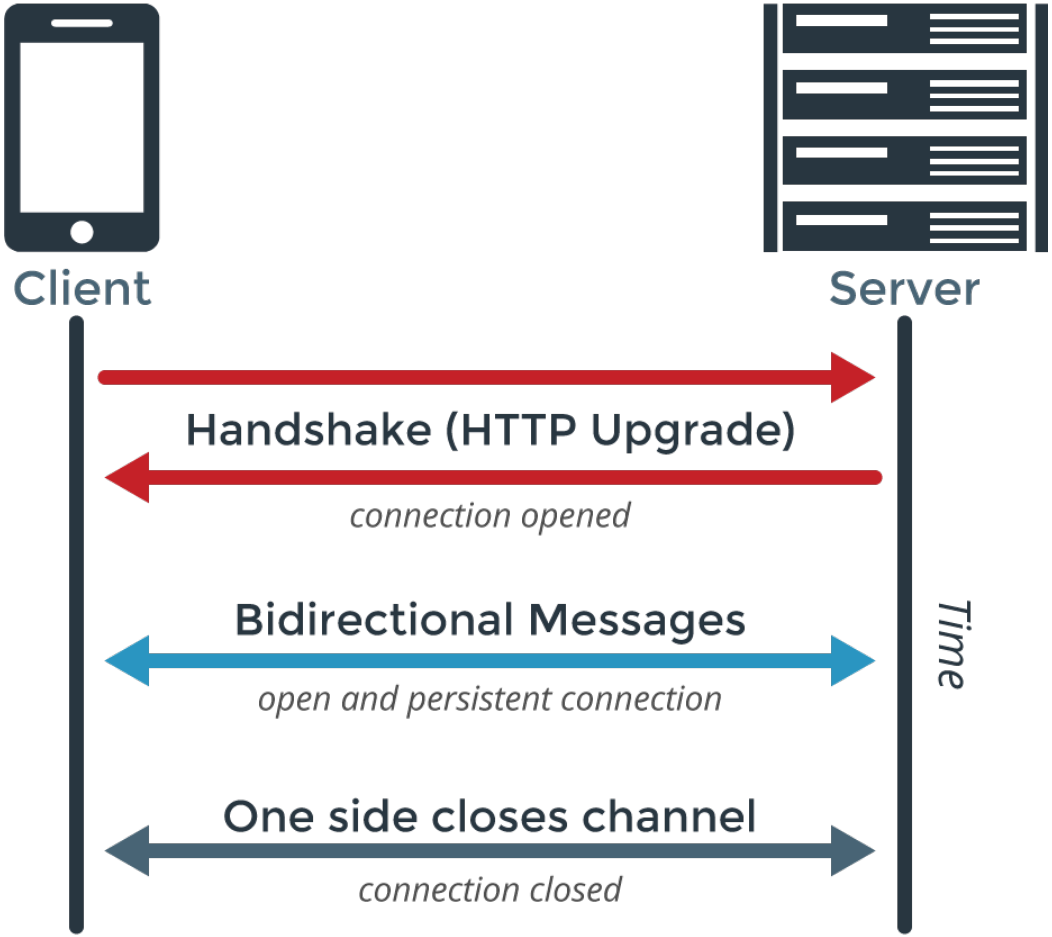
- Go provide us out of the box mock for http in httptest package to help us test our http server

```
func TestBooksIndex(t *testing.T) {
    rec := httptest.NewRecorder()
    req, _ := http.NewRequest("GET", "/books", nil)

    env := Env{db: &mockDB{}}
    http.HandlerFunc(env.booksIndex).ServeHTTP(rec, req)

    expected := "978-1503261969, Emma, Jayne Austen, £9.44\n978-1505255607, The Time Machine, H.
G. Wells, £5.99\n"
    if expected != rec.Body.String() {
        t.Errorf("\n...expected = %v\n...obtained = %v", expected, rec.Body.String())
    }
}
```

Web Sockets

- Https is a stateless protocol where a request from a client is sent to the server and the server responds to it
- One option to use when a web client needs to get updates periodically from the server , is to use a technique called pooling ,
    - this is not a real time approach...data can stay out of sync
    - It will create extra load on our http server
- Web sockets are designed to solve exactly those issues
- WebSockets offer us duplex communication from a non-trusted source to a server that we own across a tcp socket connection
- The `WebSocket` protocol, described in the specification RFC 6455 provides a way to exchange data between browser and server via a persistent connection.

Web Sockets


Client
Server
Handshake (HTTP Upgrade)
connection opened
Bidirectional Messages
open and persistent connection
One side closes channel
connection closed
Time

Web Socket workflow

1. The client sends a get request to the web server with
    a. header set to “Connection:upgrade”
    b. Another header telling the server to what the client wants to upgrade “Upgrade:websocket”
    c. Sec-websocket-key:key a key that is being exchanged between client and server to make sure the communication will be secured
2. The server responds with a status code of 101 indicating switching protocol to the client
    a. The server also responds with the same “Connection:upgrade” and “Upgrade:websocket” information
    b. The server also modies the Sec-websocket-key:key send by the client to allow the client to verify communication
3. The client and server start to communicate over websockets

Using gorilla/websocket package

Updater enables us to upgrade the connection to websocket, restrict connection from a specific origin , and define the Received and Send Buffer size (in the following example we jusr use the defaults)

```
var addr = flag.String("addr", "localhost:8080", "http service address")

var upgrader = websocket.Upgrader{} // use default options

func echo(w http.ResponseWriter, r *http.Request) {
   c, err := upgrader.Upgrade(w, r, nil)
   if err != nil {
       log.Print("upgrade:", err)
       return
   }
   defer c.Close()
   for {
       mt, message, err := c.ReadMessage()
       if err != nil {
           log.Println("read:", err)
           break
       }
       log.Printf("recv: %s", message)
       err = c.WriteMessage(mt, message)
       if err != nil {
           log.Println("write:", err)
           break
       }
   }
}

func home(w http.ResponseWriter, r *http.Request) {
   homeTemplate.Execute(w, "ws://"+r.Host+"/echo")
}

func main() {
   flag.Parse()
   log.SetFlags(0)
   http.HandleFunc("/echo", echo)
   http.HandleFunc("/", home)
   log.Fatal(http.ListenAndServe(*addr, nil))
}
```

Profiling go programs

- The go standard library has two libraries that support profiling

  net/http/pprof
  runtime/pprof

- Pprof is a tool for visualization and analysis of profiling data.
  useful for identifying where your application is spending its time (CPU and memory).

- You can install it using:
  $ go get github.com/google/pprof

  A Profile is a collection of stack traces showing the call sequences that led to instances of a particular event, such as allocation. Packages can create and maintain their own profiles; the most common use is for tracking resources that must be explicitly closed, such as files or network connections. – pkg/runtime/pprof

Profiling go programs - ReadMemoryStats

- We can use ReadMemoryStats from the runtime package to easily get the current stats of memory
- The TottalAlloc shows the total memory that was accumulated
- The heap allocation shows the amount of memory in the point in time where we call to ReadMemStats

  https://golang.org/pkg/runtime/#MemStats

```go
func printMemoryStats(status string){
    fmt.Println(status)
    runtime.ReadMemStats(&mem)
    log.Println(mem.Alloc)
    log.Println(mem.TotalAlloc)
    log.Println(mem.HeapAlloc)
    log.Println(mem.HeapSys)
}
// hugeMap allocates 100 megabytes
func hugeMap() *map[string][]byte {
    s := make(map[string][]byte)
    s["first"] = make([]byte, 100000000)
    s["second"] = make([]byte, 100000000)
    return &s
}
func main() {

    var mem runtime.MemStats

    printMemoryStats("start")

    for i := 0; i < 10; i++ {
        s := hugeMap()
        if s == nil {
            log.Println("oh noes")
        }
    }
    printMemoryStats("finish")
}
```

Profiling go programs - CPU

- We can use the pprof tool by creating a .profile file for analysis by instrumenting our code
- The only thing we need to do in order to instrument our code is calling the StartCPUProfile method of the runtime library
- And finally call the StopCPUProfile
- Calling StartCPUProfile will create a .profile file that we then give to the pprof to analyze
- For example:

```
go build -o cpuprofiling && time ./cpuprofiling > cpu.profile
$ go tool pprof cpu.profile
```

```
var factVal uint64 = 1
var i int = 1
var n int

func factorial(n int) uint64 {
   if n < 0 {
       fmt.Print("cant use Factorial of negative number")
   } else {
       for i := 1; i <= n; i++ {
           factVal *= uint64(i)
       }
   }
   return factVal
}
func main() {
   pprof.StartCPUProfile(os.Stdout)
   defer pprof.StopCPUProfile()

   for i := 0; i < 10; i++ {
       factorial(10000000)
   }
}
```

Profiling go programs -Memory

- To do memory profiling we just need to change the call from StartCPUProfile to WriteHeapProfile
- Please make sure to call the WriteHeapProfile after memory has been allocated
- Please note that we have only one function to call that just create a snapshot at the given point in time, rather having two functions start and stop that suggest an ongoing process like cpu
- Building and running is the same process like cpu profiling

```
func hugeMap() *map[string][]byte {
    s := make(map[string][]byte)
    s["first"] = make([]byte, 100000000)
    s["second"] = make([]byte, 100000000)
    return &s
}
func main() {

    for i := 0; i < 10; i++ {
        s := hugeMap()
        if s == nil {
            log.Println("oh noes")
        }
    }
    pprof.WriteHeapProfile(os.Stdout)
}
```

```
$ go build -o memoryprofiling && time memoryprofiling > cpu.profile
$ go tool pprof cpu.profile
```

Profiling using remote http server

- In order for our app to support remote profiling using an http server, we need to enable it by importing the "net/http/pprof"
- We also need to pull up a webserver ,the webserver is needed cause the pprof expose some endponts through it
- If there is already a webserver in our app , and provided that we use a different ServerMultiplexer than the default , we will need to add the endoints to it, as shown in the example
- Now you can access profiling information http://localhost:5000/debug/pprof/

```
package main
import (
   "net/http"
   "net/http/pprof"
)
func myendpoint(w http.ResponseWriter, r *http.Request)
{
   w.Write([]byte("simple web server"))
}
func main() {
   r := http.NewServeMux()
   r.HandleFunc("/", myendpoint)

   r.HandleFunc("/debug/pprof/", pprof.Index)
   r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
   r.HandleFunc("/debug/pprof/profile", pprof.Profile)
   r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
   r.HandleFunc("/debug/pprof/trace", pprof.Trace)

   http.ListenAndServe(":5000", r)
}
```

What are those profiling endpoints?

Actually all the below links are just links to .profiles that in order to process we will need to use a tool lilke pprof

- block: stack traces that led to blocking on synchronization primitives
- goroutine: stack traces of all current goroutines
- heap: a sampling of all heap allocations
- mutex: stack traces of holders of contended mutexes
- threadcreate: stack traces that led to the creation of new OS threads

Those are special links that are dynamically generated

- profile: a 30 second cpu profile that is created dynamically and returned as a file
- Trace: a 5 second trace hat is created dynamically and returned as a file

So in order to use those we will need to execute from a different shell:
go tool pprof http://localhost:6060/debug/pprof/<.profile>

Where <.profile> is the name of the subject we wish to get profiling information on for example:
go tool pprof http://localhost:6060/debug/pprof/goroutine

Visualizing profiling data

- We can generate an image for the data that was analysed in the format od svg,pdf,png and gif

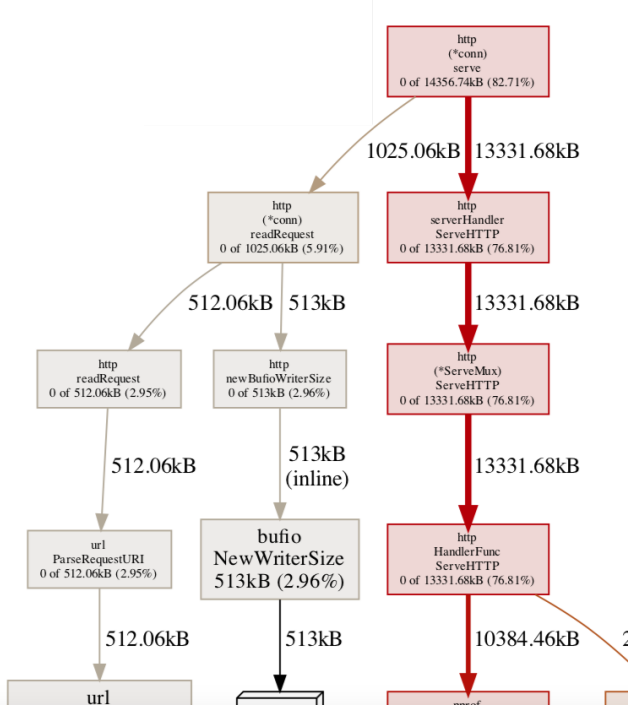
- Make sure that you have graphviz installed on your syste you cab use “brew install) https://enterprise-architecture.downloads?id=208

- Then for example execute :

  go tool pprof -alloc_space -pdf http://localhost:5000/debug/pprof/heap

- We can also use pprof webui tool by running

  $ go tool pprof -http localhost:8080 ../cpuprofiling/cpu.profile



Tracing

- go tool trace visualizes all the runtime events over a run of your Go program,
- go tool trace is better suited at finding out what your program is doing *over time*, not in aggregate

1. Creation, start and end of goroutines
2. Events that block/unblock goroutines (syscalls, channels, locks)
3. Network I/O related events
4. Syscalls
5. Garbage collection

In order to support tracing for your webserver, you will need to add following endpoint:
r.HandleFunc("/debug/pprof/trace", pprof.Trace)

For utilising tracing in a non http app all we need to do is import "runtime/trace" and then add calls to the trace.Start and trace.Stop

Then we need to compile our app generate the trace data, and open it with the trace tool like so:

$ **go** run .
$ **go** tool trace trace.out

Building Go Plugin

- Plugins allow developers to build loosely coupled modular programs
- Plugins are compiled as a shared object libraries and that can be loaded at runtime
- A Go plugin is a package compiled using the `-buildmode=plugin` build flag to produce a shared object (`.so`) library file
- A plugin package must be identified as `main`.
- Exported package functions and variables become shared library symbols

Building Go Plugin

On the right , For example you can see that the sort plugin is just an exported function name , using it is simple as :

1. Opening the file using plugin package `plugin.Open("./plugins/bubblesort.so")`

1. Looking for the exported function

```
symbol, err := p.Lookup("Sort")
```

1. Executing the function

```
sortFunc, ok := symbol.(func([]int) *[]int)
sorted := sortFunc(numbers)
```

```
package main

func Sort(items []int) *[]int {
    if len(items) < 2 {
        return &items
    }
    tmp := 0
    for i := 0; i < len(items); i++ {
        for j := 0; j < len(items)-1; j++ {
            if items[j] > items[j+1] {
                tmp = items[j]
                items[j] = items[j+1]
                items[j+1] = tmp
            }
        }
    }
    return &items
}
```

Building Go program for multiple platform

- when developing software that needs to run on multiple operating systems ,it is a common practice to build your final binary for many different platforms, to maximize your program’s performance.

- It can be difficult when the platform you are using for development is different from the platform you want to deploy your program to

- In the past building for Mac or for windows , forced you to set up build machines for those operating systems

GOOS & GOARCH

- Go solves that issue directly in the go build tool by adding support for multiple platforms.
- using the env variables and build tags you can control which operating system your binary will be built for, and also you can toggle code that will run only on a specific operating system
- To see all the platform go can compile for run $ go tool dist list
- To see the current architecture you are running on run go env GOOS GOARCH

GOOS & GOARCH

- the following sample use file path separator as it is platform dependent to demonstrate compiling to different platforms

- When running this program, you will receive different output depending on which platform you are using

- On windows we will get a\b\c , vs on linux or Mac we will get a/b/c

- when calling filepath.Join() :
  the Go tool chain automatically detects your machine's GOOS and GOARCH and uses this information to use the code snippet with the right build tags and file separator

```
package main

import (
 "fmt"
 "path/filepath"
)

func main() {
 s := filepath.Join("a", "b", "c")
 fmt.Println(s)
}
```

GOOS & GOARCH

Note that the path.go file is with the path separator for linux based os, and we prevent this compiling on windows by adding inversion sign For ! windows : **path.go**

path.go

```
// +build !windows

package main

const PathSeparator = "/"
```

main.go

```
package main

import (
 "fmt"
 "strings"
)

func Join(parts ...string) string {
 return strings.Join(parts,
PathSeparator)
}

func main() {
 s := Join("a", "b", "c")
 fmt.Println(s)
}
```

If we will try to run this on windows we will get an error

Fixing the problem - compile on windows

- We can fix it by adding a file for windows as well : **windows.go**
- If go finds the GOOS and GOARCH env variables it will use them for compiling , if not it will use the current os and arch the app is running on
- We can change the compilation target by changing thuds env variables
  For example: **GOOS=windows go build**
- You will see now an app.exe file in your directory although you are on Mac or linux

```
// +build windows

package main

const PathSeparator = "\\"
```

Using GOOS **and** GOARCH **Filename Suffixes**

- the Go standard library makes heavy use of build tags to simplify code by separating out different platform implementations into different files

- Go also use suffixes for which platform the file was meant for, for example On the os/path_windows.go meant for windows

- When naming a .go file, you can add GOOS and GOARCH as suffixes to the file's name in that order, separating the values by underscores (_)

- For example for a windows os with arm architecture we could use a file name like **myfile_windows_arm64.go**

Fast Exercise

1. Update your program to use file suffixes instead of build tags to compile for unix and windows

Building Docker image for go - CGO_ENABLED=0

- Usually compiled go binaries do not have any c dependencies so we are not dependent on runtime libraries, and the go libraries we use are embedded in executable

- If we compile our app on linux the resulting binary will link against a few c standard libraries,this creates a dependency between our go app an the linux os

- Please note that if we compiles on windows or mac with the GOOS=linux env , we will not have this issue,cause the code will link statically

- On linux we can force the code to use static linking by using the CGO_ENABLED=0

Building Docker image for go

- The following docker file is a generic docker file that takes TARGETOS and TARGETARCH as args
- The CGO_ENABLED=0 force go to compile with static linking , so it is not dependent on the libraries on the hosting os
- Also GOOS and GOARCH are assigned with values to define the os and arch when building the app
- We can build image with the following command

  $ docker build -t dynamicgo --build-arg TARGETOS=windows --build-arg TARGETARCH=amd64 --target bin .

- The Scratch image is a special image directly build into the docker engine, and allow us build minimal image

```
ARG TARGETOS
ARG TARGETARCH
FROM golang:1.14.3-alpine AS build
WORKDIR /src
ENV CGO_ENABLED=0
COPY . .
RUN go mod download
RUN GOOS=${TARGETOS} GOARCH=${TARGETARCH} go build -o /
out/example .

FROM scratch AS bin-unix
COPY --from=build /out/example /

FROM bin-unix AS bin-linux
FROM bin-unix AS bin-darwin

FROM scratch AS bin-windows
COPY --from=build /out/example /example.exe

FROM bin-${TARGETOS} as bin
ENTRYPOINT ["/example"]
```

Kubernetes Recap

- Kubernetes Automates and monitors the lifecycle of a stateless application
- It can scale the app up or down and always make sure it keeps running
- Kubernetes Cluster Consists of computers called nodes
- The basic unit of work in kubernetes called pod, which is a group of one or more linux containers
- Kubernetes can be divided to planes
  - Control plane - the actual pods that the kubernetes core components runs on,exposing the api,etc...
  - Data Plane - everything else meaning the nodes and pods which the application runs on

- Kubernetes is made from one or more masters and a bunch of nodes (use to be called minions)
- Master Components holds a “control plane” which in charge of which work to run on which node
- (In short control plane is what monitor the cluster schedule the work , makes the changes, etc..)
- The nodes are what actually do the real work , they report back to the master and watch for changes

Kubernetes control plane and data plane

- Kubernetes is actually a collection of pods implementing the k8s api and the cluster orchestration logic what also referred as the kubernetes control plane

- The application/data plane is everything else meaning all the nodes that hosts all the pods that the application runs on

- The *controllers* of the control plane implement control loops that repeatedly compare the desired state of the cluster to its actual state

- When the state of the cluster changes , controllers take actions to match it back again to it’s desired state

Masters in more details

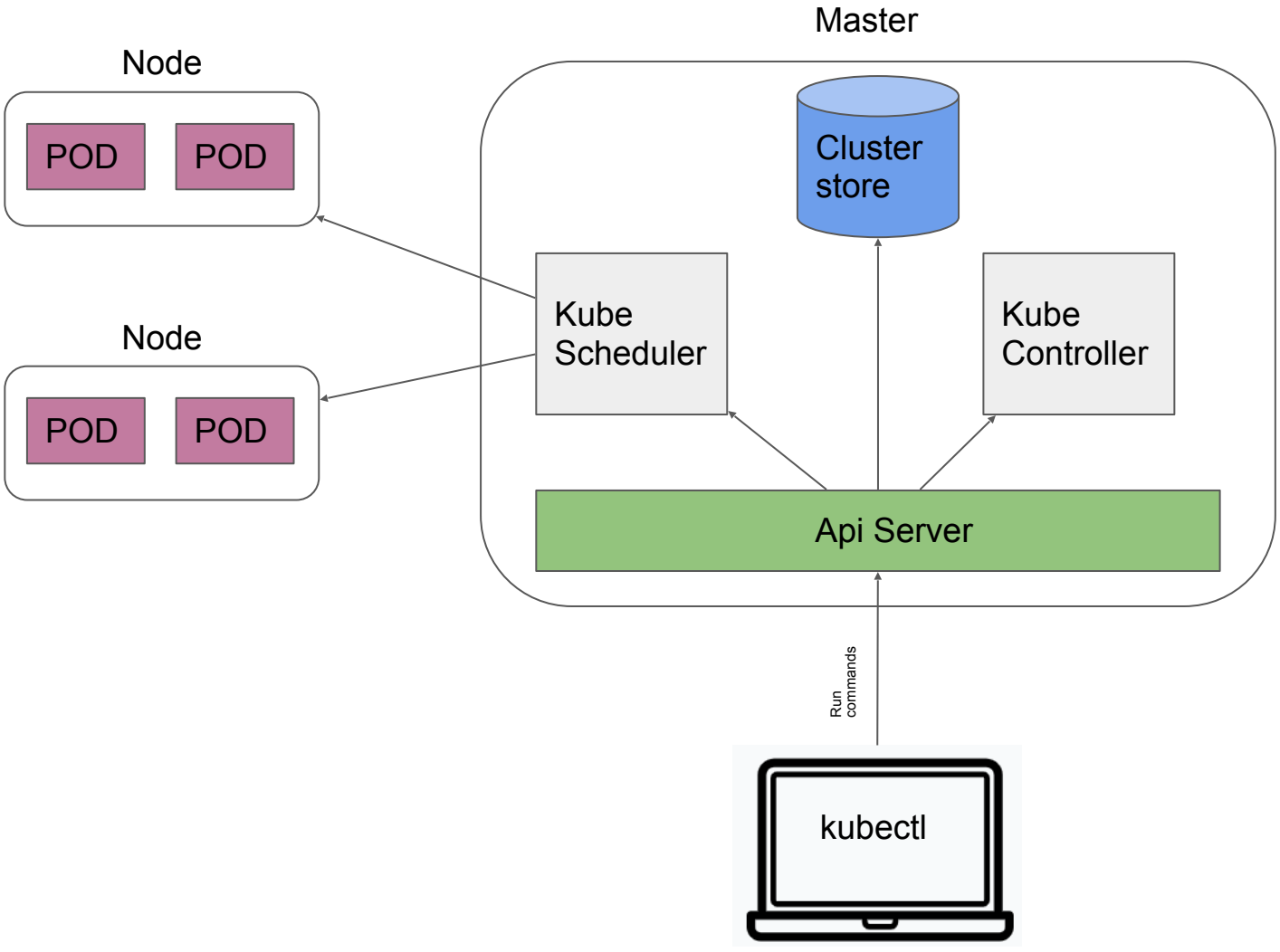
- The Master is composed of multiple components
  for simplicity, set up scripts typically start all master components on the same machine, and do not run user containers on this machine.

- **Kube-apiserver** which is the front facing interface into the master , it expose it’s interface via Rest api and consumes json or yaml

- **Etcd** is the cluster store it is an open source distributed key value database that uses to Keeps the state and config of the cluster
  - Just to make it clear we send a manifest file that defines our intents to the api server the api server validates it and save it into the cluster store

- **kube-scheduler**
  - Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.
  - Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

- **Kube-Controller-Manager** - Component on the master that runs controllers
  - Node Controller: Responsible for noticing and responding when nodes go down.
  - Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
  - Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).
  - Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.


Master
Node
POD
POD
Node
POD
POD
Cluster store
Kube Scheduler
Kube Controller
Api Server
Run commands
kubectl

- Kubernetes support a declarative model , hence we send the api a yaml file in a declarative form

- By Desired state it means, we specify in the yamls file our desired state and the cluster responsibility is to make sure it will happen

- We describe the desired state using yaml or json file that serves as a record of intent, but we do not specify how to get there(this is kubernetes responsibility to get us there)

- Things could change or go wrong over the lifetime of the cluster(node failing etc…) ,the kubernetes is responsible to always make sure that the desired state is kept.

- The kubernetes control plane controllers are always running in a loop and checking that the actual state of the cluster matches the desired state, so that if any error occurs they kick in and rectify the cluster

- Watch for the spec fields in the YAML files later!
- The *spec* describes *how we want the thing to be*
- Kubernetes will *reconcile* the current state with the spec

  (technically, this is done by a number of *controllers*)
- When we want to change some resource, we update the *spec*
- Kubernetes will then *converge* that resource

Kubernetes basic execution unit -> Pod

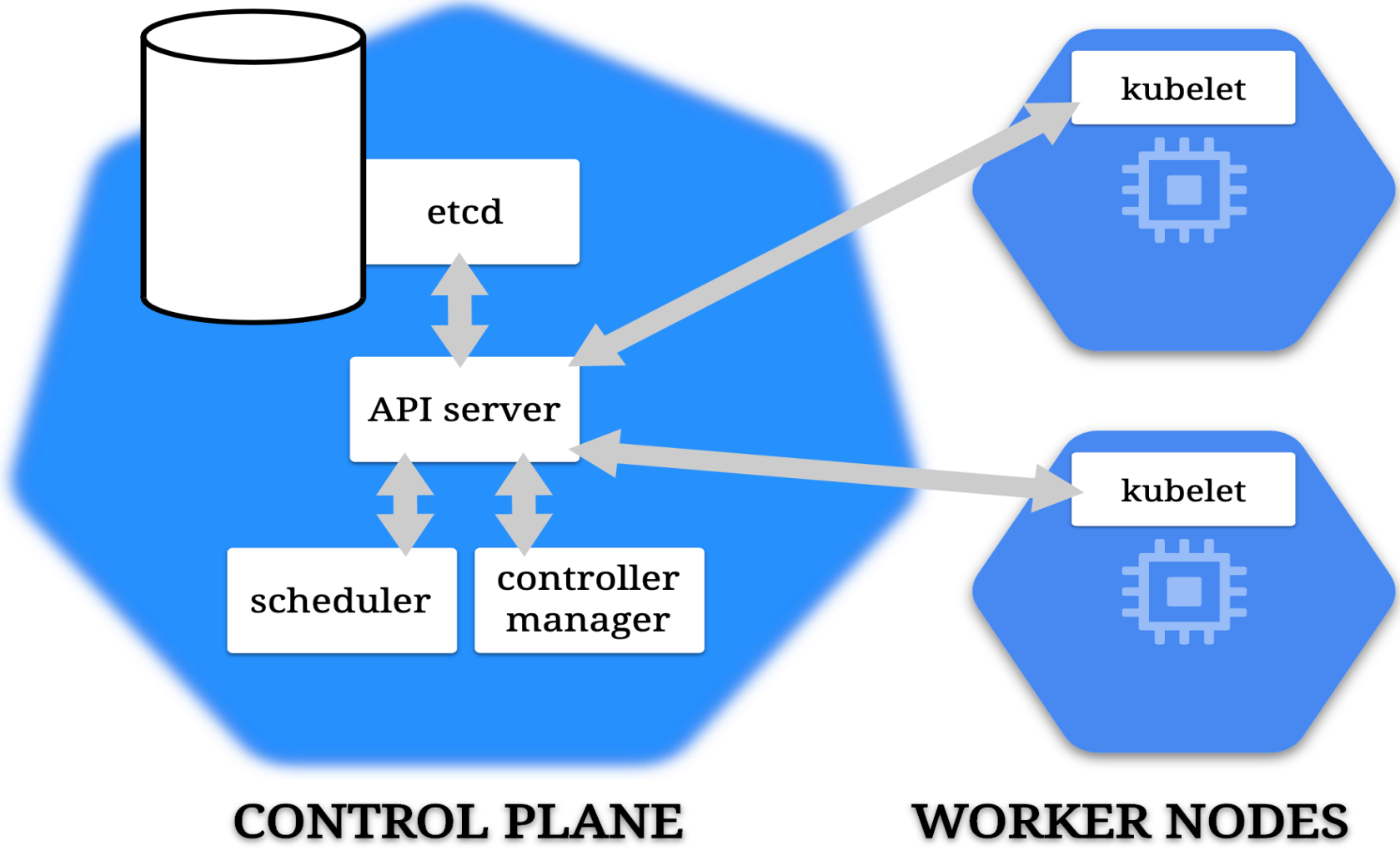
- A Pod is the basic execution and scaling unit in kubernetes,
- kubernetes runs containers but always inside pods
- It is like a sandbox to run containers in
- A pod is a group of containers:
  - running together (on the same node)
  - sharing resources (RAM, CPU; but also network, volumes)
- A Pod models an application-specific “logical host”

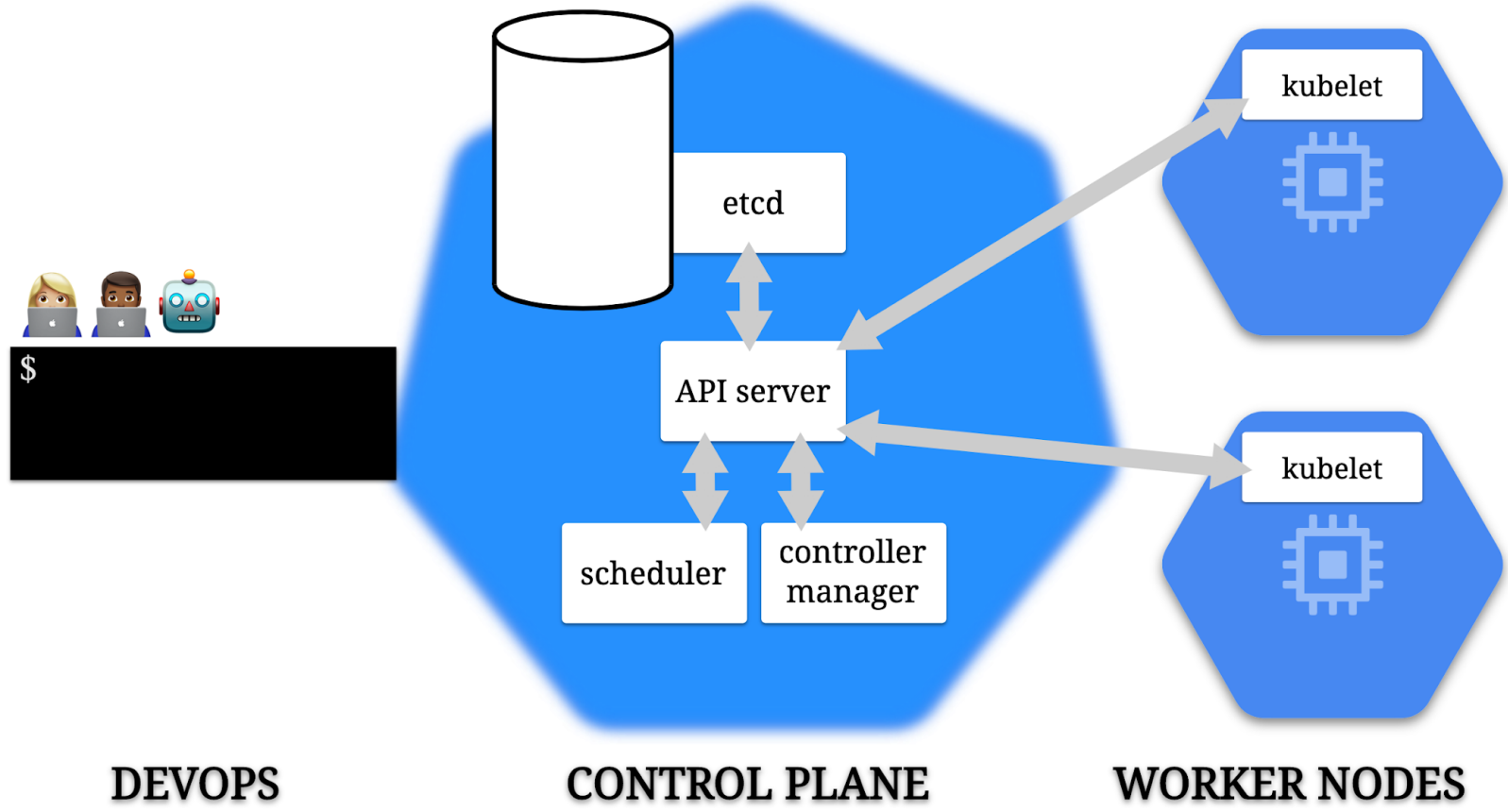
Pod state

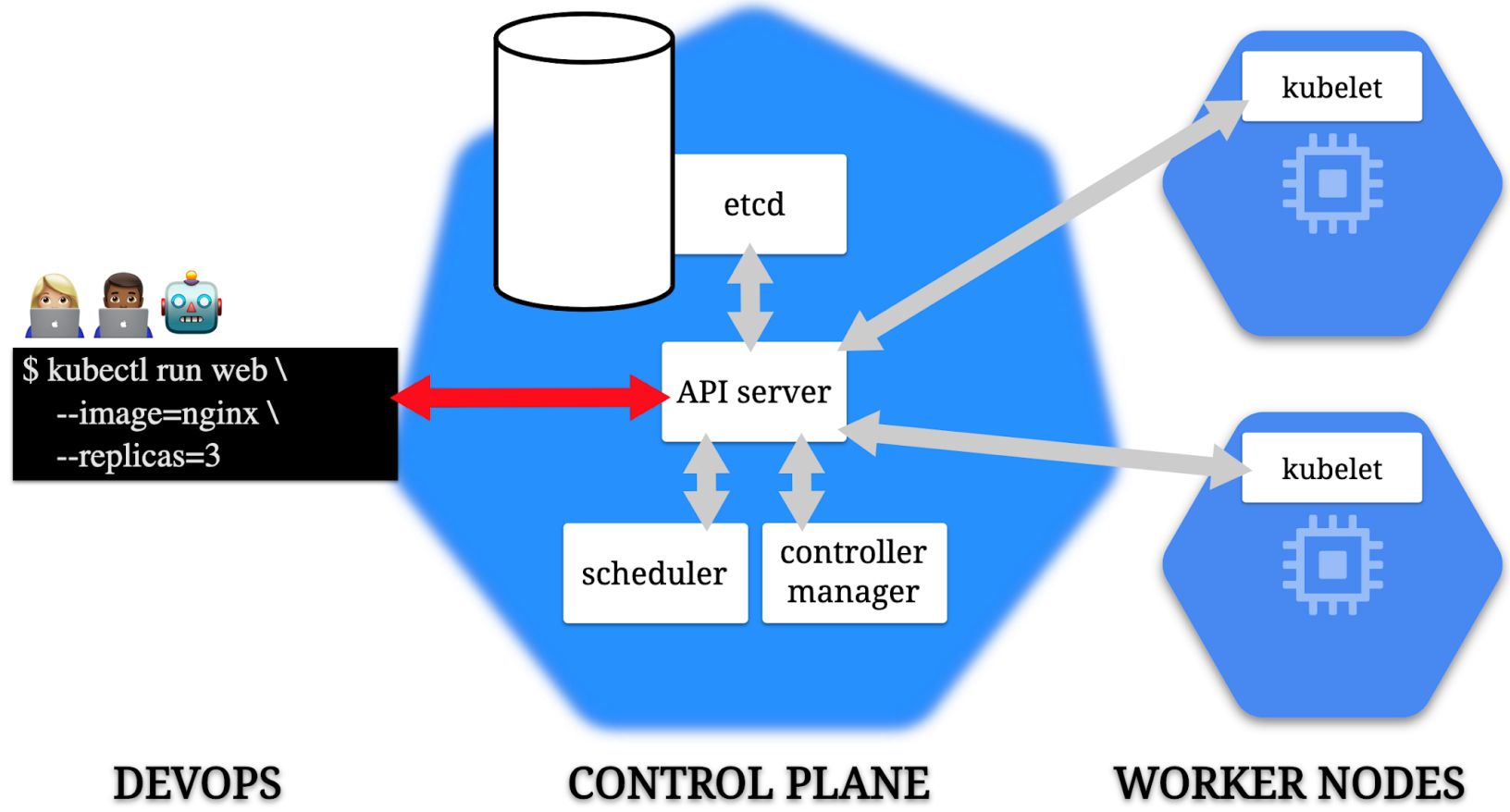
- Pods are considered to be relatively ephemeral (rather than durable) entities
- Pods do not hold state , so one a pod crash kubernetes will replace it with another
- Multiple instances of the same pod are called replicas

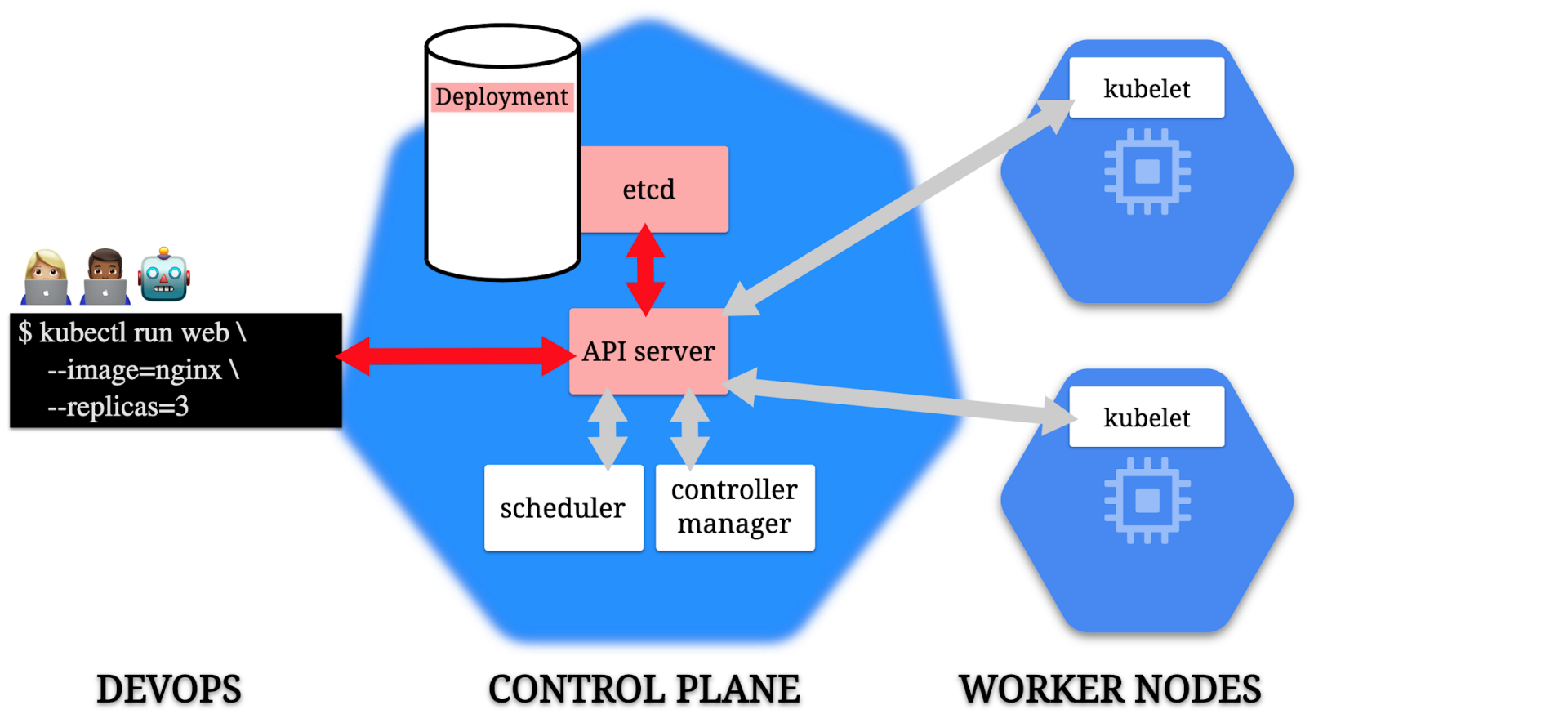
Deployments

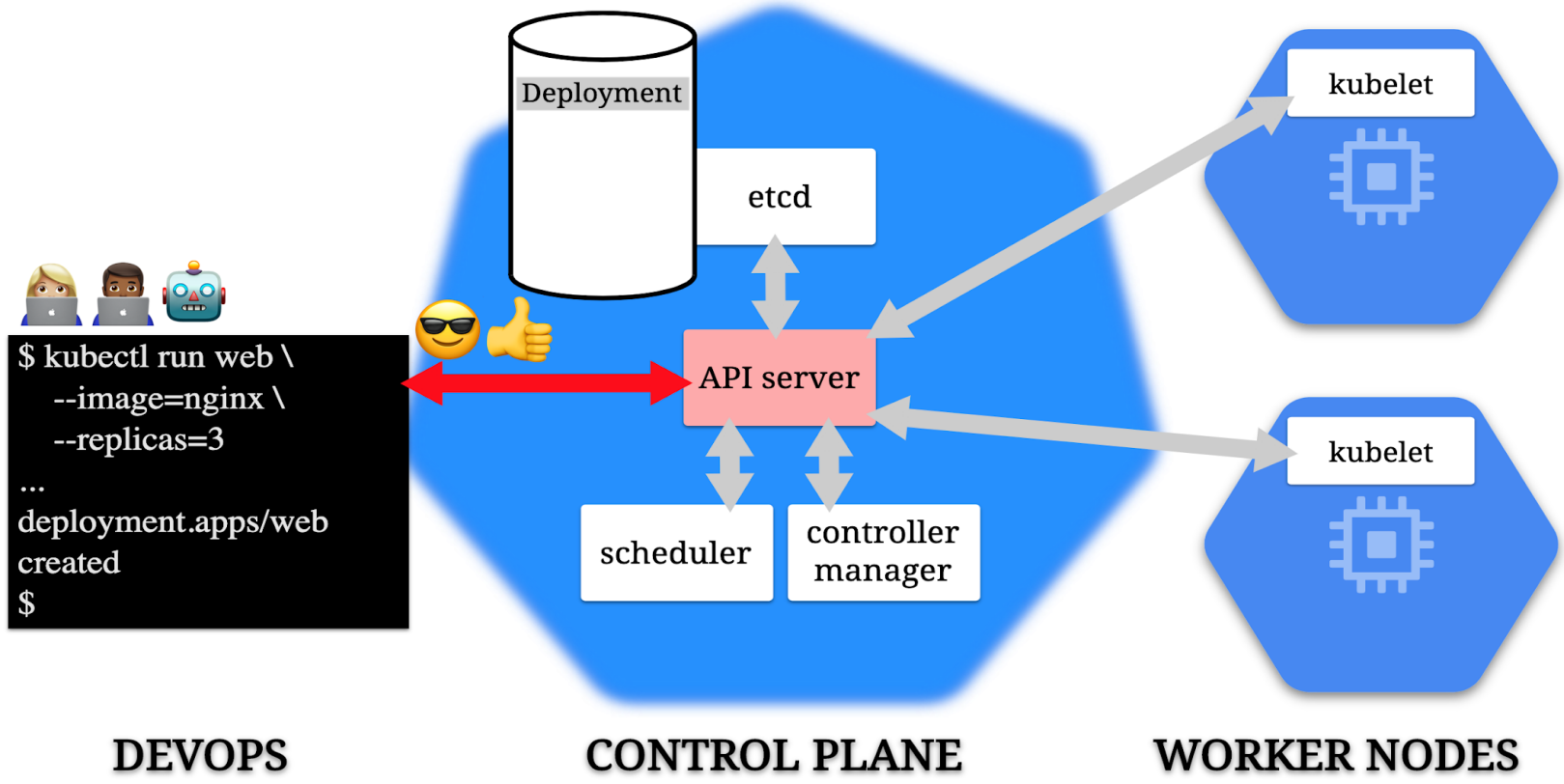
- A *Deployment* controller provides declarative updates for Pods and ReplicaSets.
- The Deployment Object gives us a better way of handling the scaling of PODs
- The advantage of using Deployment versus using a replicaset is having rolling updates support of the POD container versions out-of-the-box.
- Every time the application code changes, a new version of the application container is built, and then there is a need to update the Deployment manifest with the new version and tell K8s to apply the changes.
- K8s will then handle the rolling-out of this newer version, it will start terminating PODs with the old version as it spins up the new PODs with the updated container.
- This means that at some point we will have multiple versions of the same application running at the same time

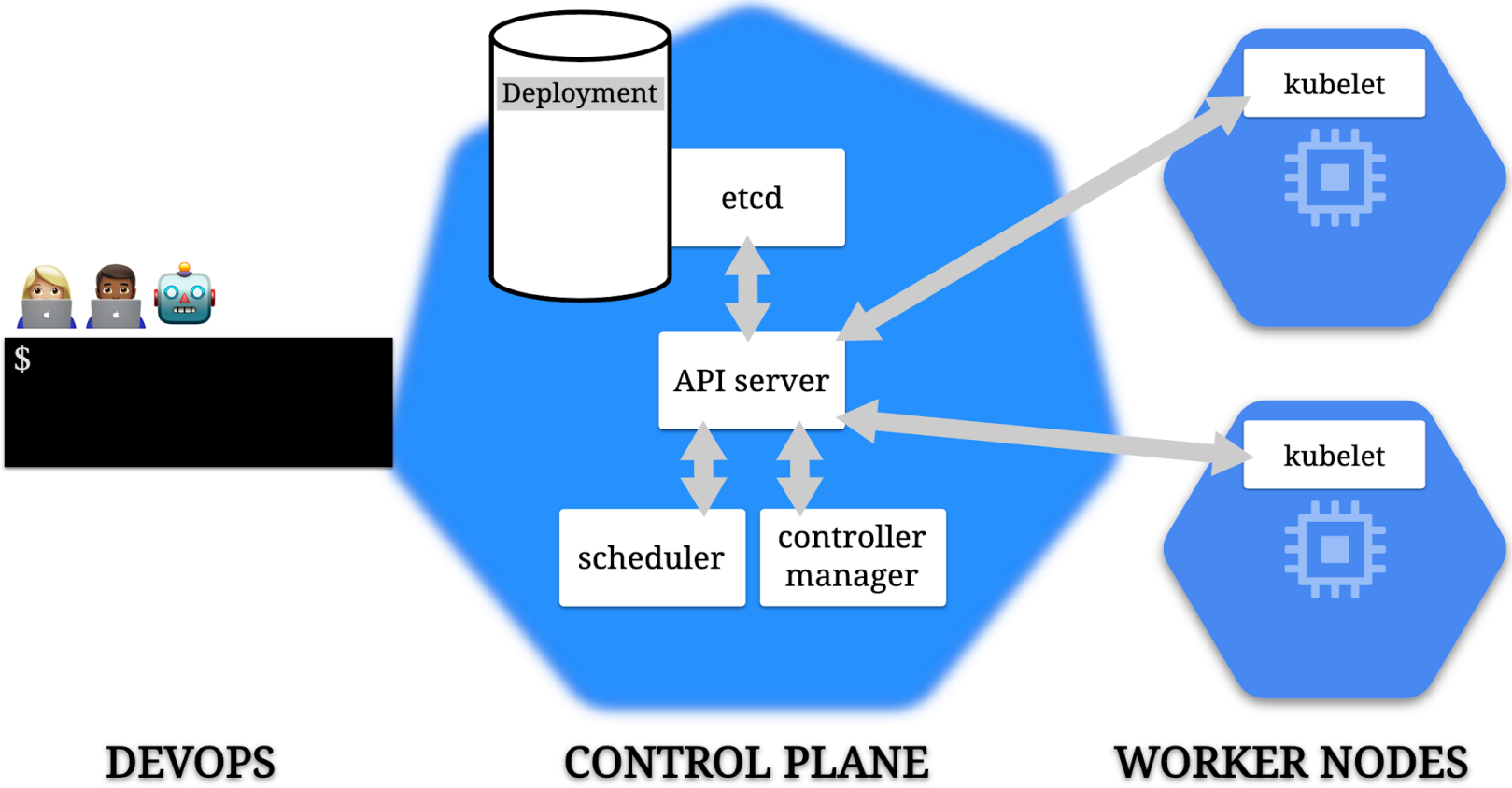

etcd
API server
scheduler
controller manager
kubelet
kubelet
CONTROL PLANE
WORKER NODES

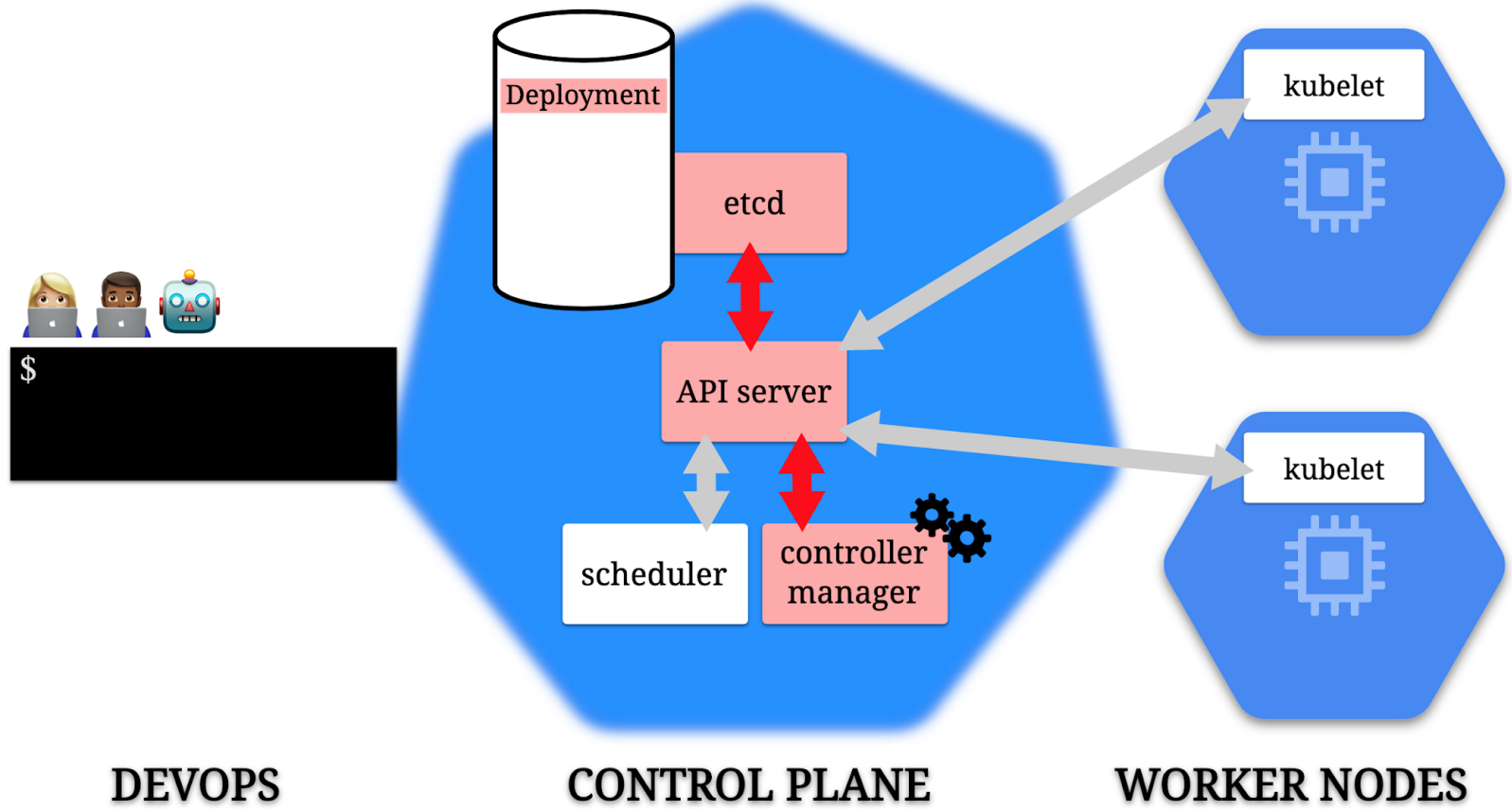

etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

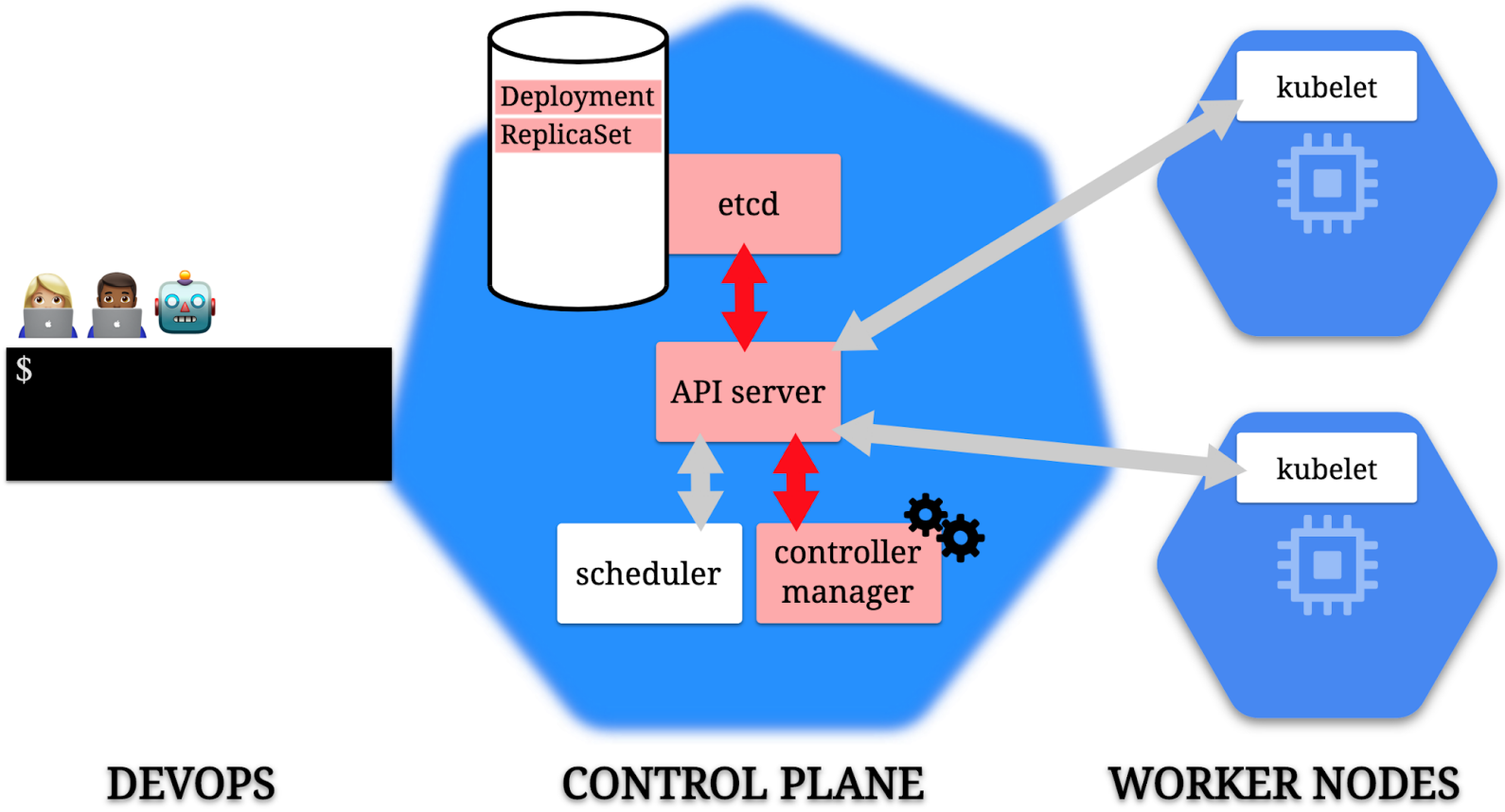

etcd
kubelet
$ kubectl run web \
--image=nginx \
--replicas=3
API server
kubelet
scheduler
controller
manager
DEVOPS
CONTROL PLANE
WORKER NODES

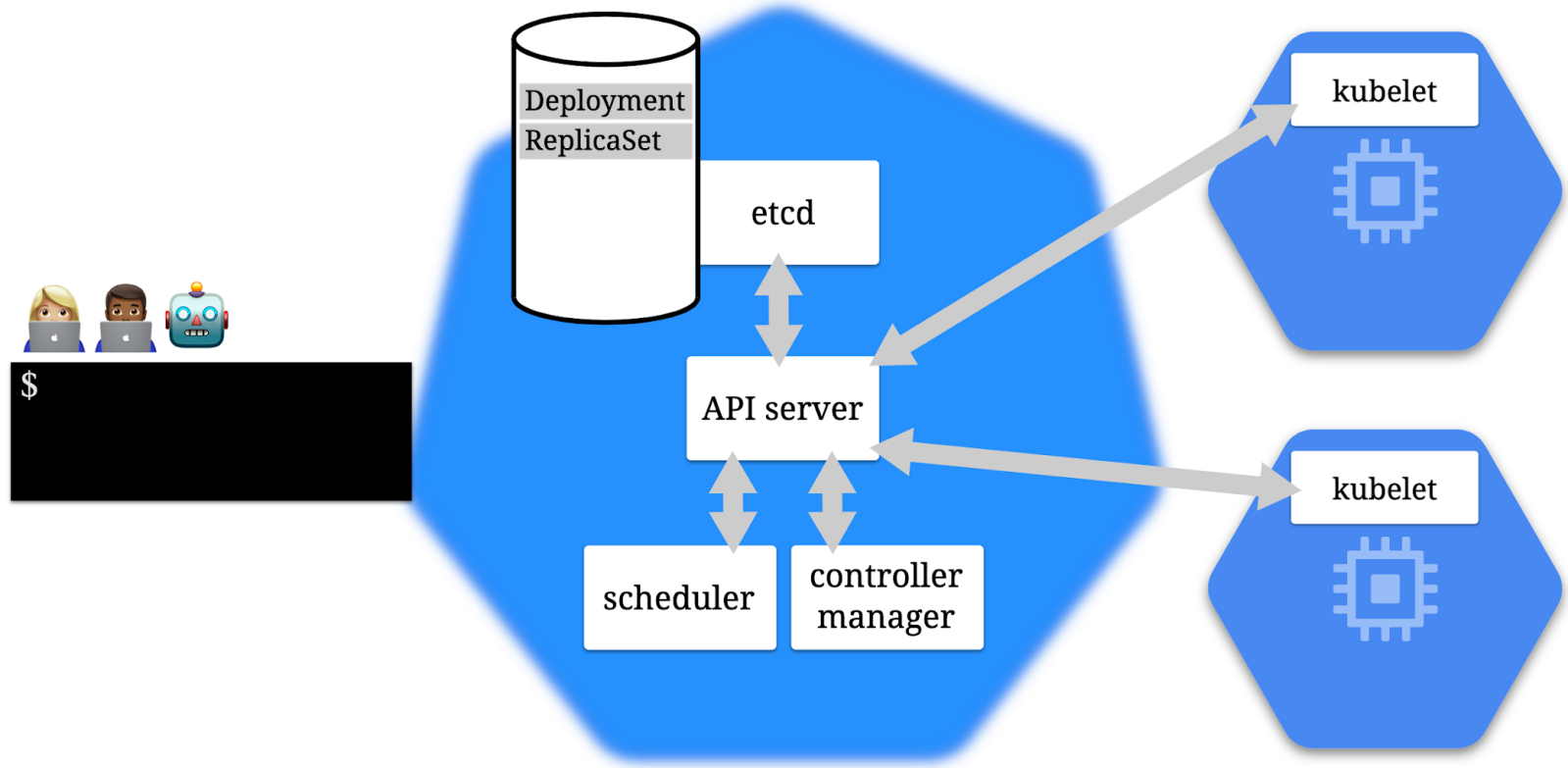

Deployment
etcd
kubelet
$ kubectl run web \
--image=nginx \
--replicas=3
API server
kubelet
scheduler
controller
manager
DEVOPS
CONTROL PLANE
WORKER NODES

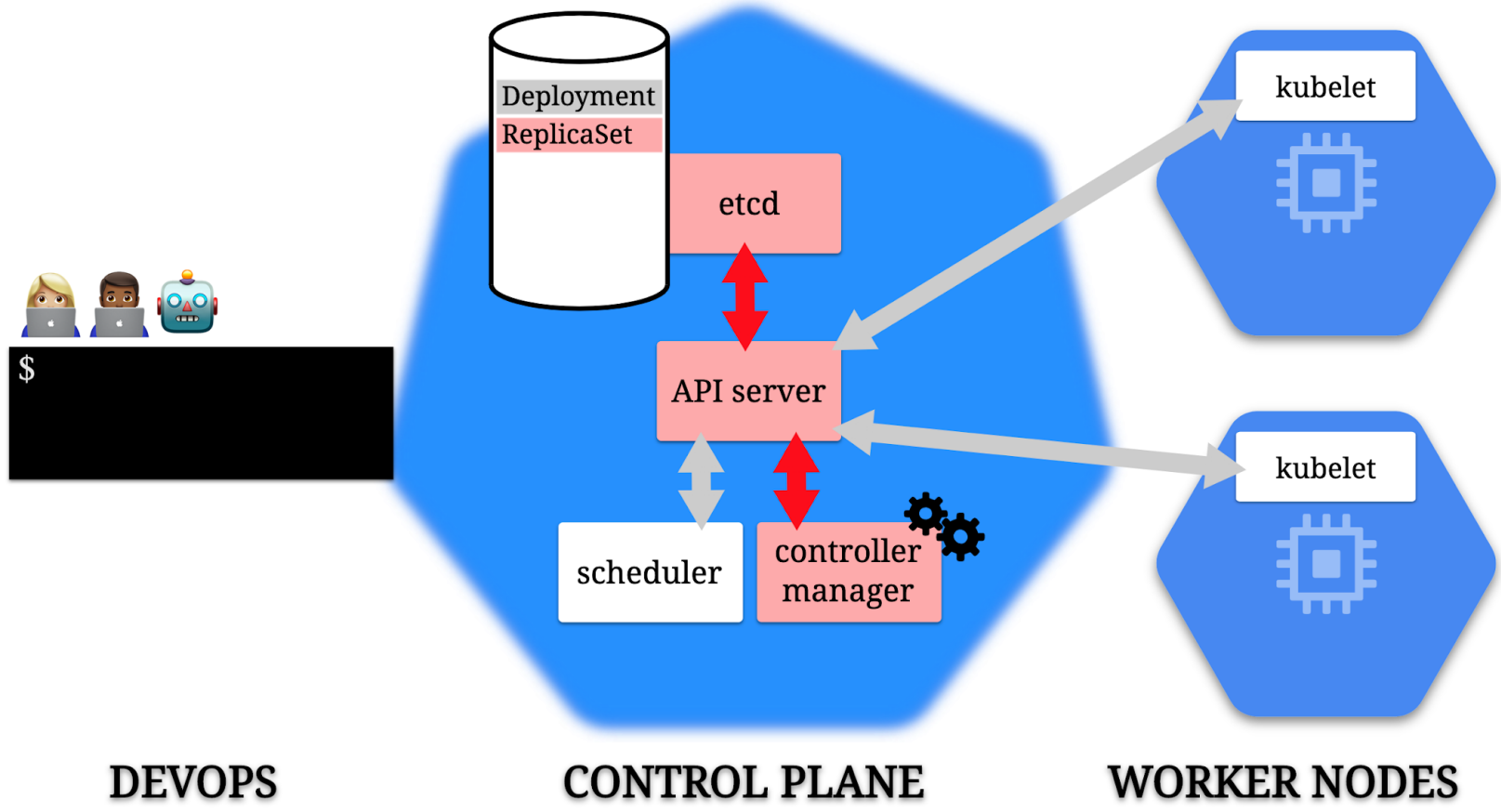

Deployment
etcd
API server
scheduler
controller manager
kubelet
kubelet
$ kubectl run web \
--image=nginx \
--replicas=3
...
deployment.apps/web created
$
DEVOPS
CONTROL PLANE
WORKER NODES

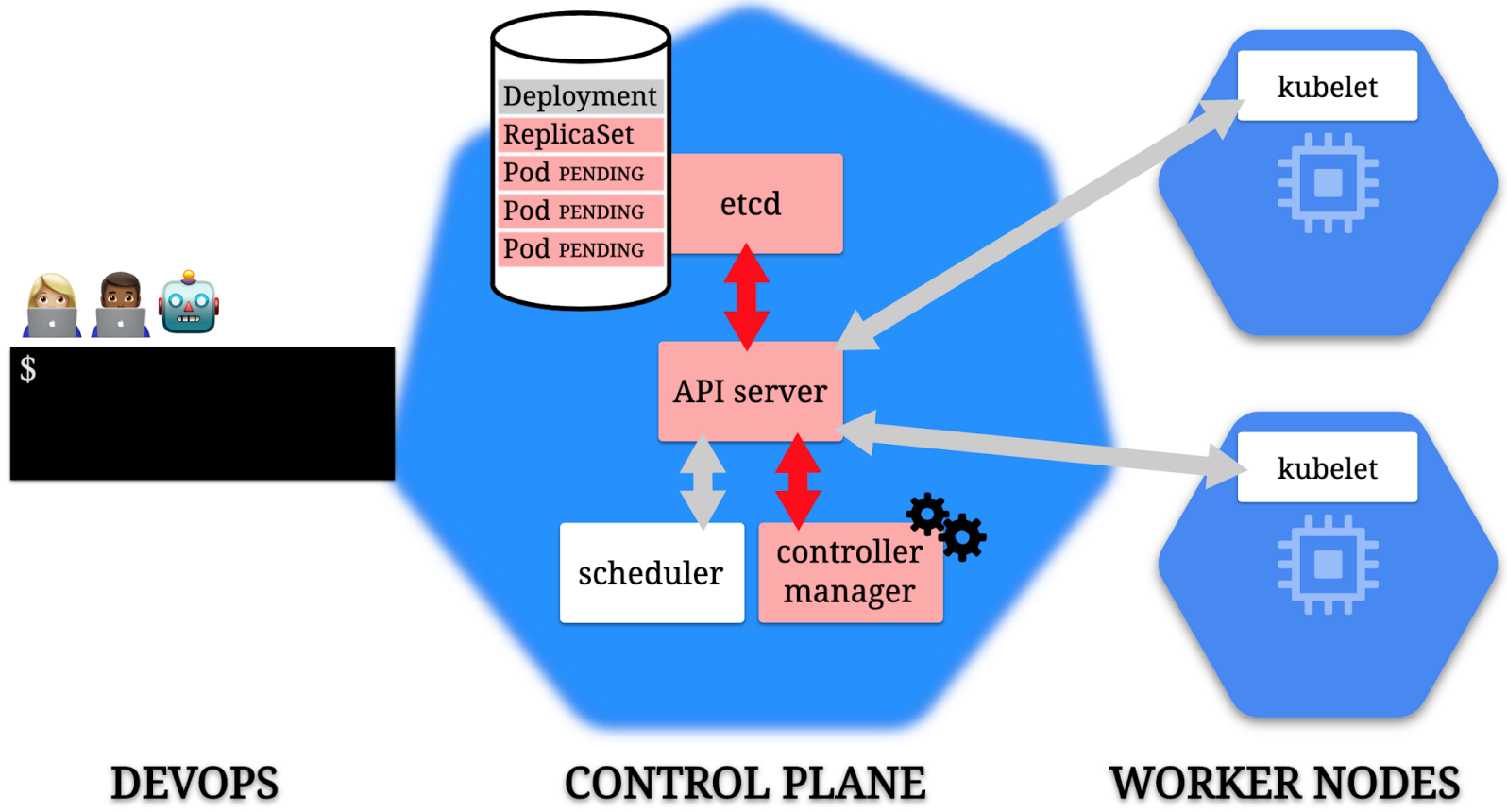

Deployment
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

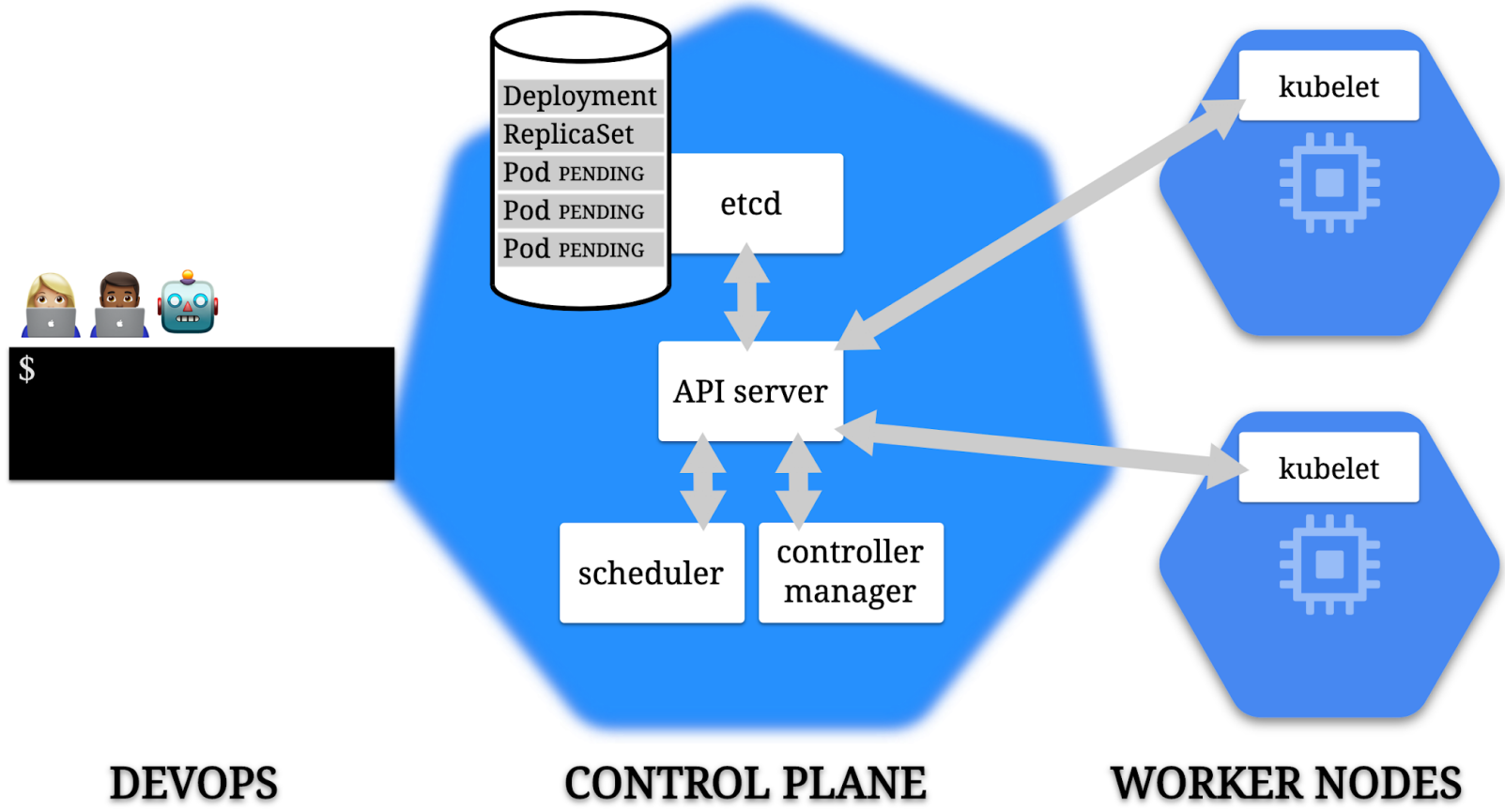

Deployment
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

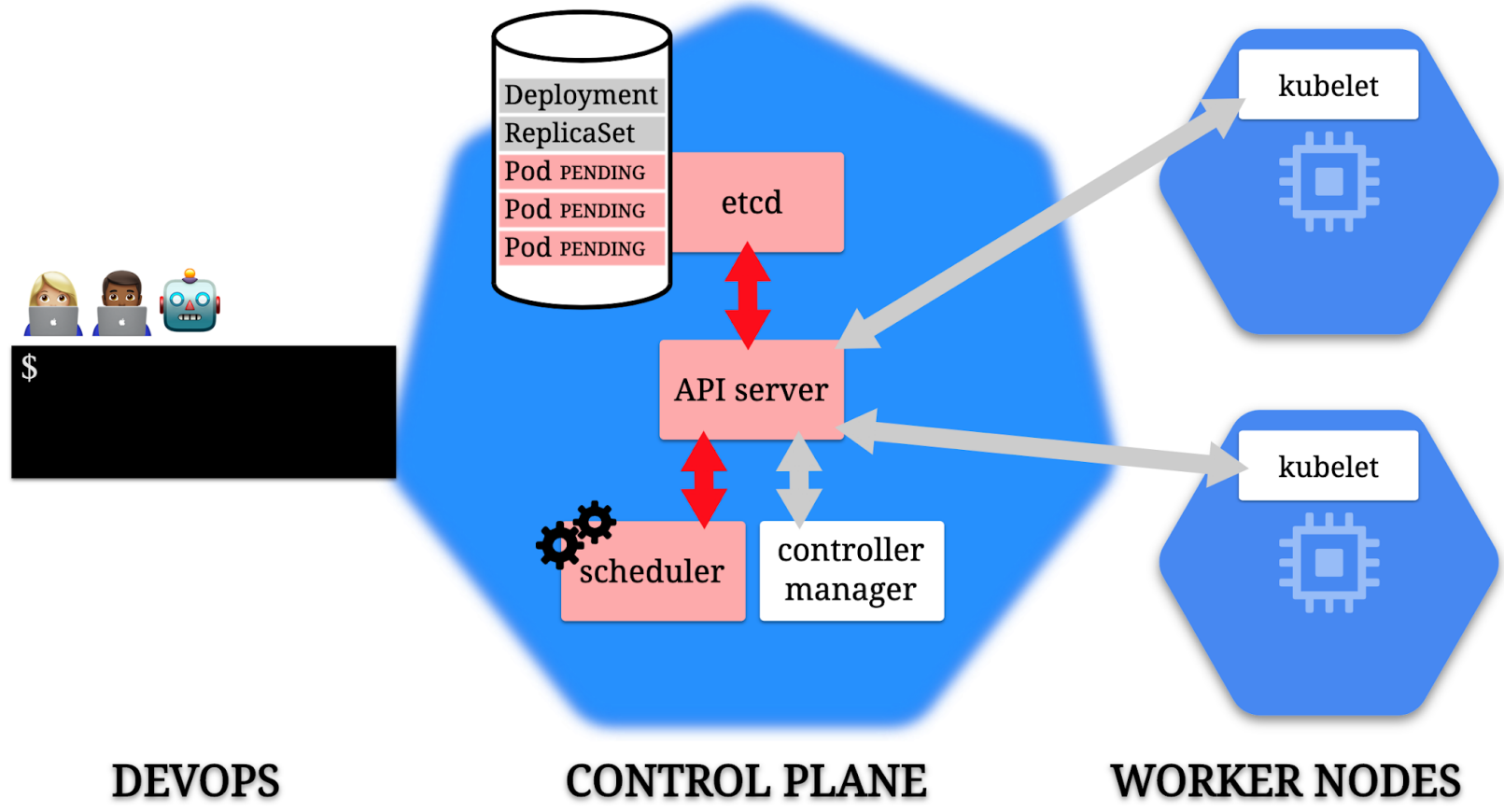

Deployment
ReplicaSet
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

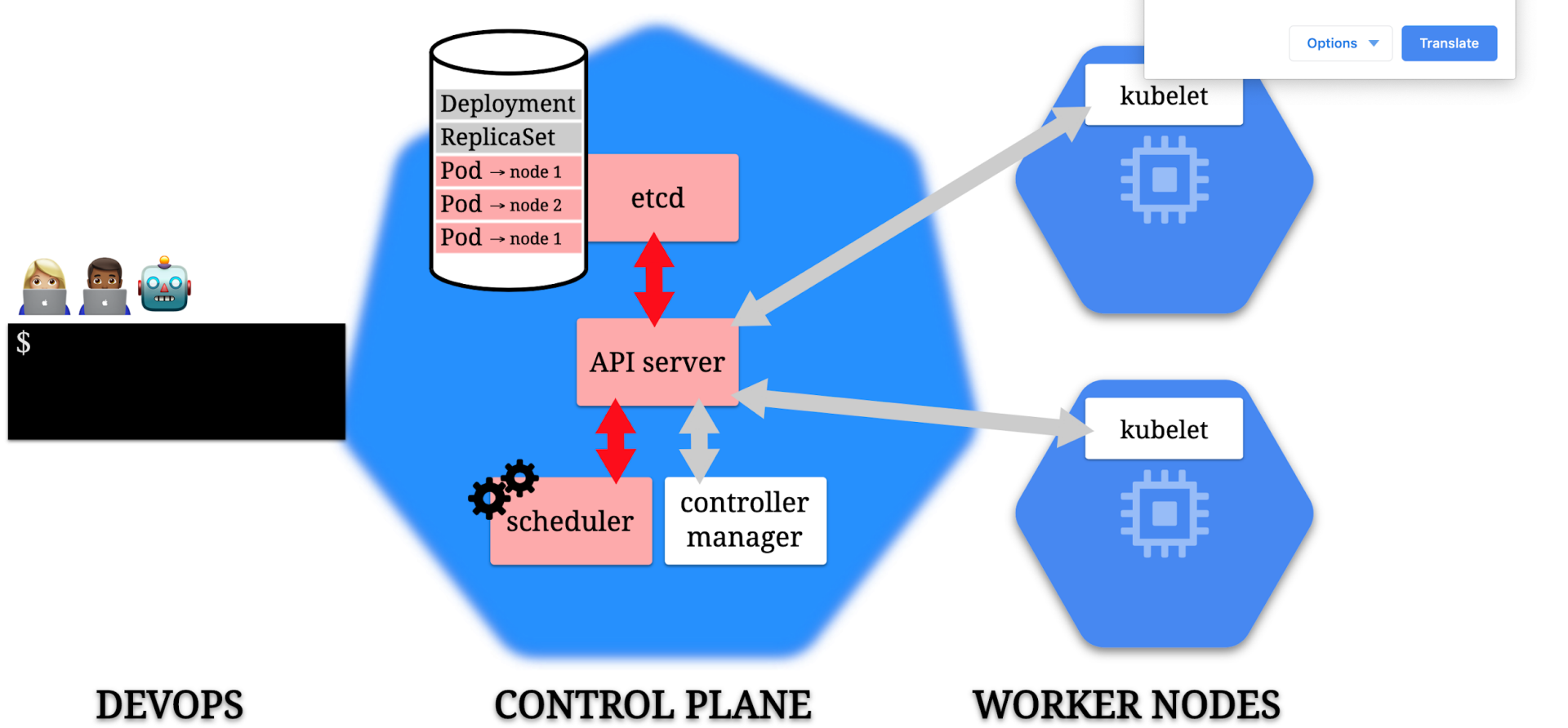

Deployment
ReplicaSet
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

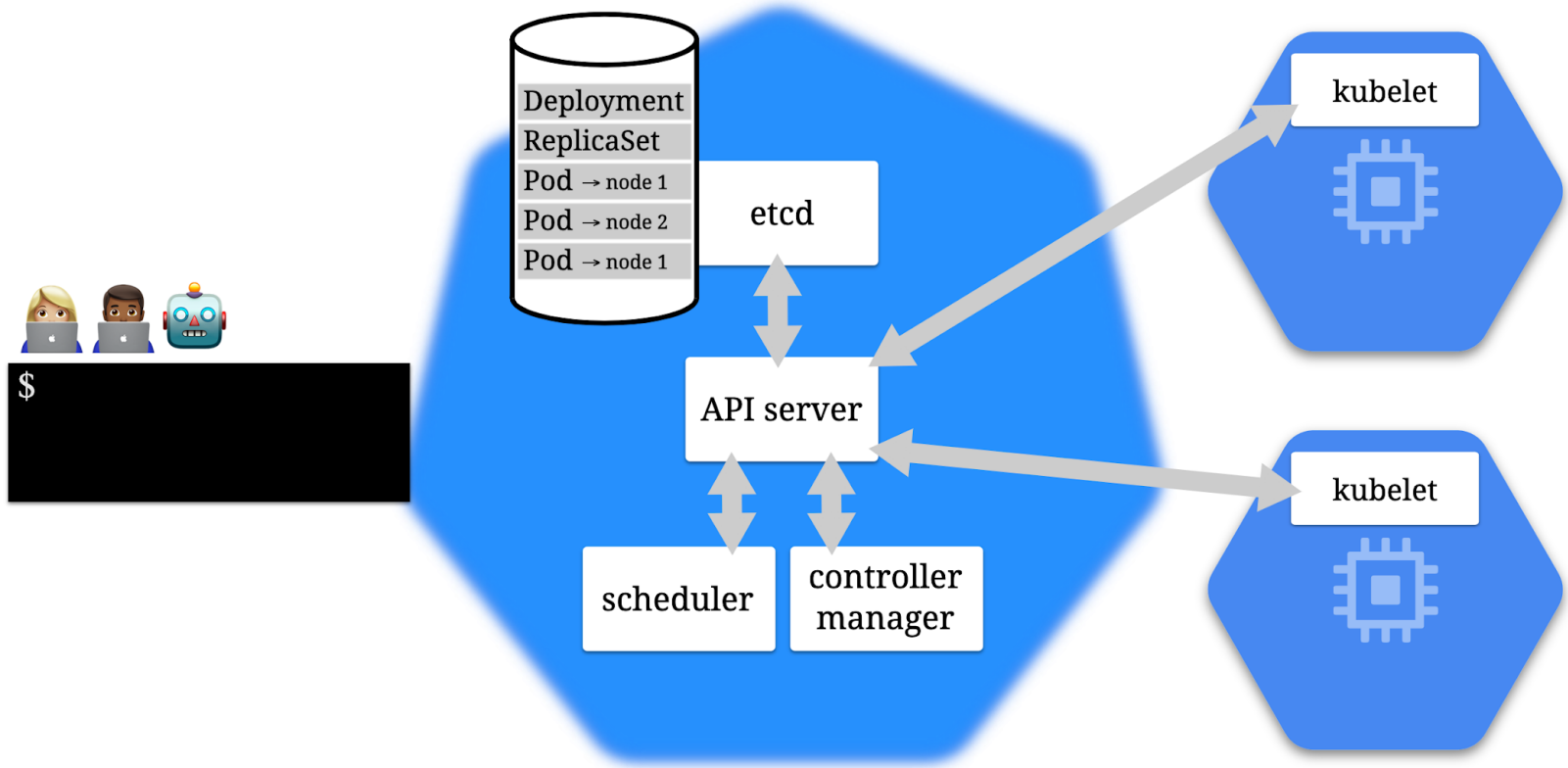

Deployment
ReplicaSet
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod PENDING
Pod PENDING
Pod PENDING
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod PENDING
Pod PENDING
Pod PENDING
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod PENDING
Pod PENDING
Pod PENDING
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

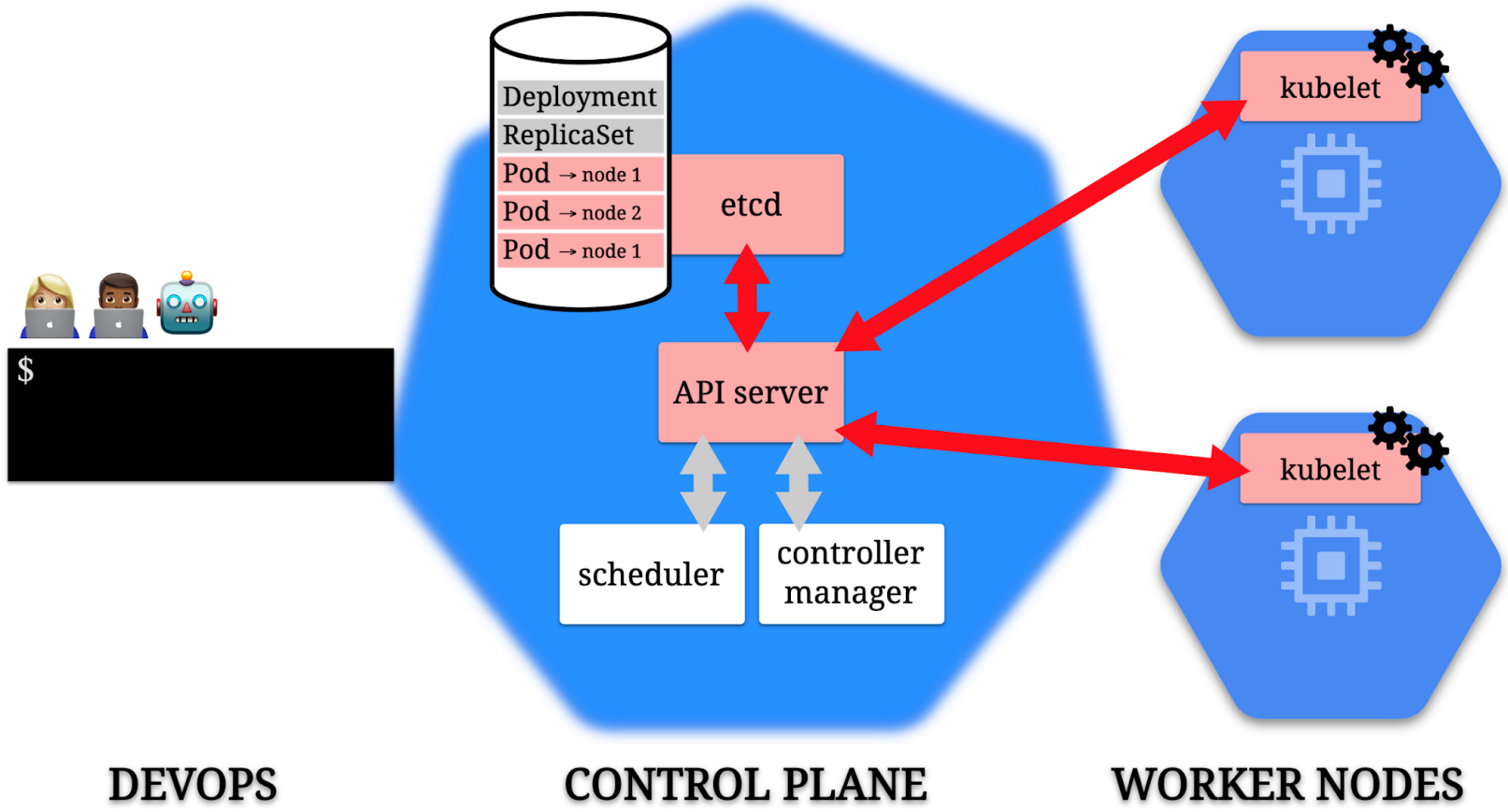

Deployment
ReplicaSet
Pod → node 1
Pod → node 2
Pod → node 1
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

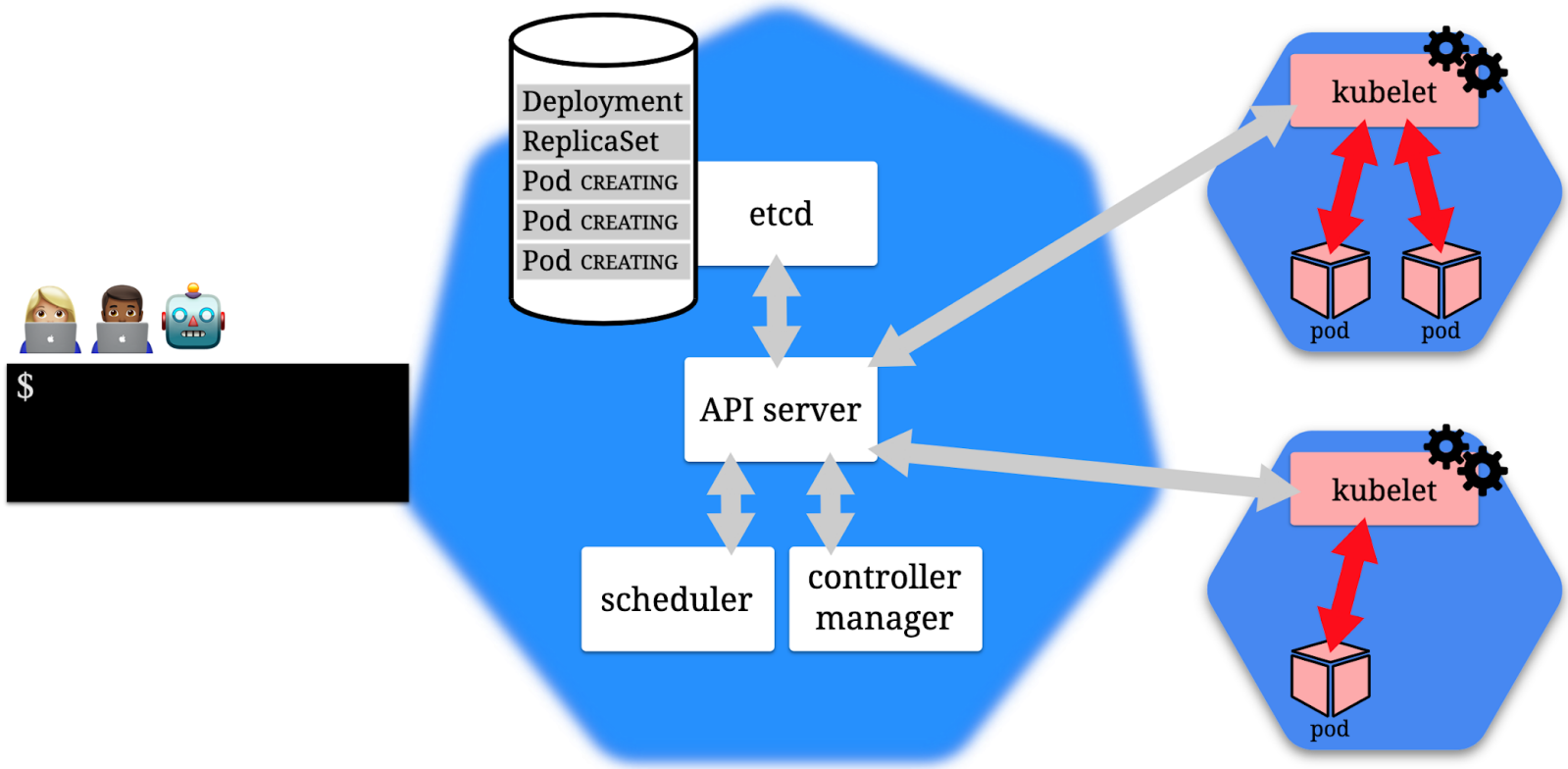


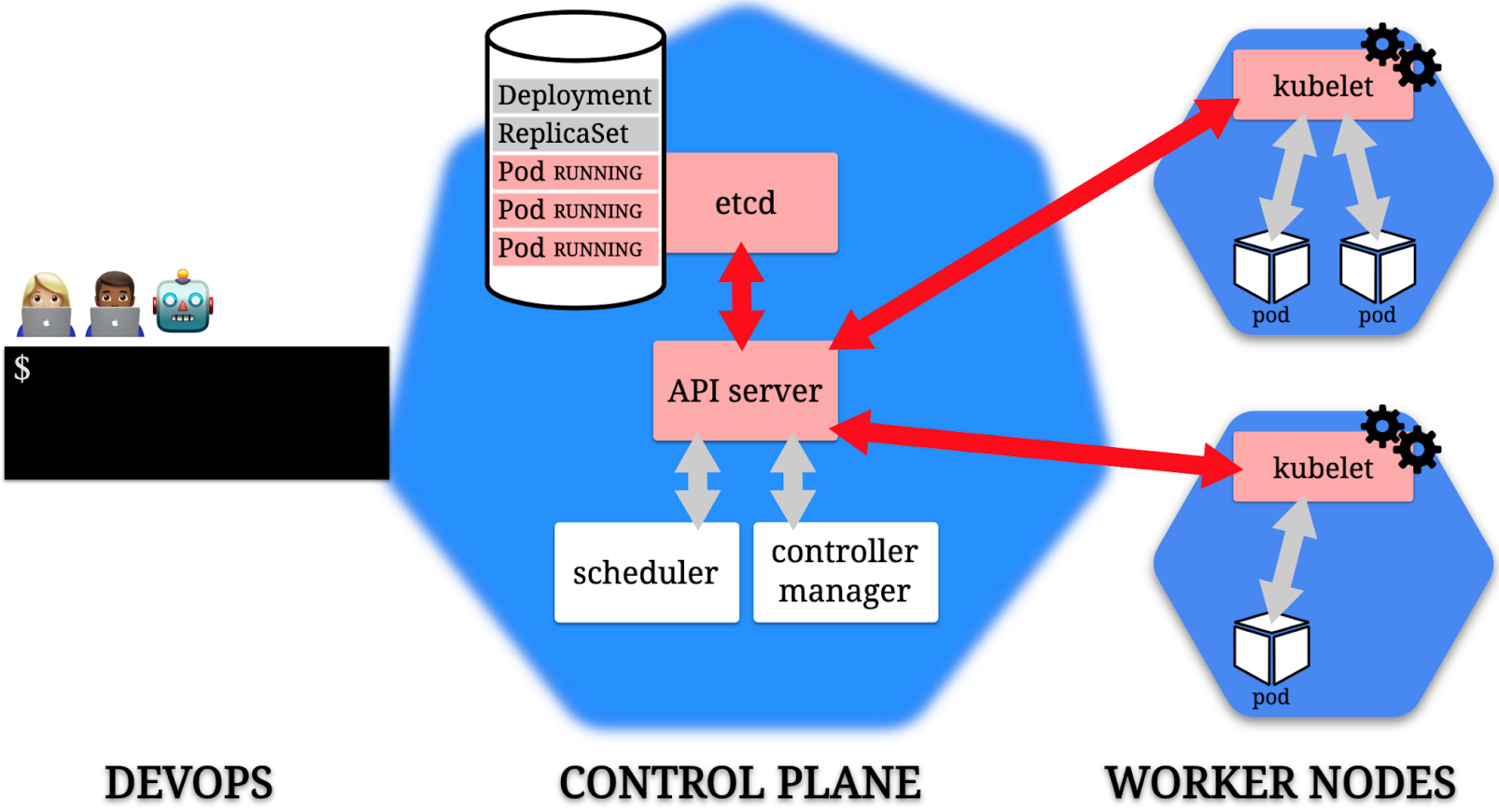
DEVOPS

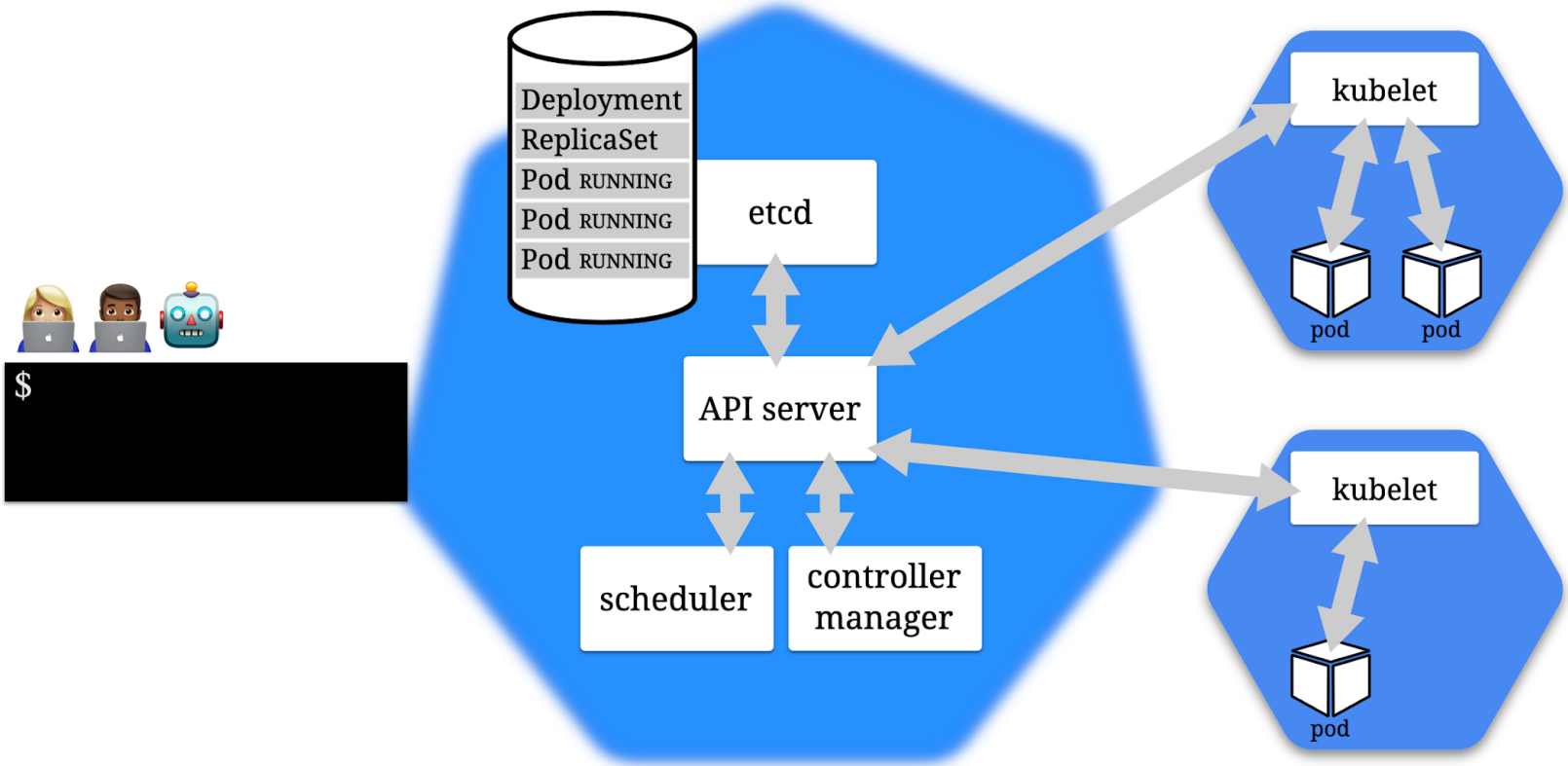
CONTROL PLANE

WORKER NODES


Deployment
ReplicaSet
Pod → node 1
Pod → node 2
Pod → node 1
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod CREATING
Pod CREATING
Pod CREATING
etcd
API server
scheduler
controller manager
kubelet
pod
pod
kubelet
pod
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod RUNNING
Pod RUNNING
Pod RUNNING
etcd
API server
scheduler
controller manager
kubelet
pod
pod
kubelet
pod
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod RUNNING
Pod RUNNING
Pod RUNNING
etcd
API server
scheduler
controller manager
kubelet
pod
pod
kubelet
pod
$
DEVOPS
CONTROL PLANE
WORKER NODES

Kubernetes Operators

- An Operator builds on Kubernetes abstractions to automate the entire lifecycle of the software it manages
- As a Software developer we can say that Operators provides the infrastructure services to dependent app
- Operators provide a clean and consistent way to distribute software on kubernetes clusters
- Operators Extends the Kubernetes control plane and api, adding an endpoint to the kubernetes api with a custom resource
- The control plane monitors and maintain the custom resource of the operator
- automating systems administration by writing software to run your software

Kubernetes operators

- An Operator is a way to package, run, and maintain a Kubernetes application
- An operator is not just a regular app deployed into kubernetes , it uses all the kubernetes facilities as it is a part of it.
- Operators are like core services that applications running on kubernetes depends on
- Deploying applications to kubernetes are made easy and consistent with operators
- Operators reduce support burdens by identifying and correcting application problems automatically

Kubernetes operators

- Operators are there to make sure that the cluster desired state is kept

- Operator can be sometimes considered as a control plane component

- Most Applications have state like configuration data , startup component, and data storage , and actually have their own notion of what a cluster means

- Kubernetes cannot know all about every stateful, complex, clustered application while also remaining general and simple

- An operator knows about the internal state of the application it manage

Kubernetes operators

- Operators are like an expert administrator.
- For example An Operator can manage a cluster of database servers,
- It knows the details of configuring and managing its application, and it can install a database cluster of a declared software version and number of members.
- An Operator continues to monitor its application as it runs, and can back up data, recover from failures, and upgrade the application over time, automatically.
- Cluster users employ kubectl and other standard tools to work with Operators and the applications they manage,

Kubernetes operators

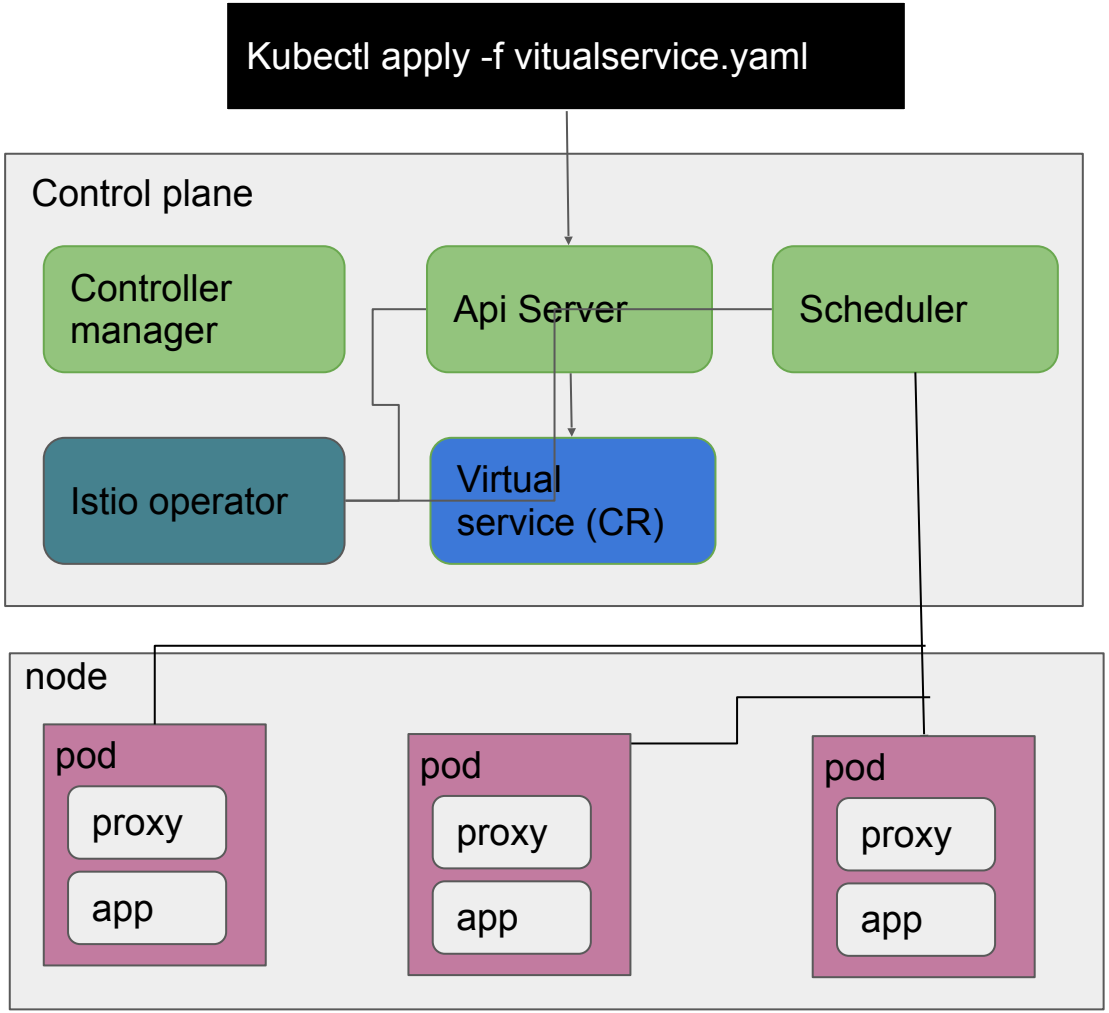
- Operators enables us to provide specific features for our software by extending kubernetes
- Operators are like the fondation services that remove the management overhead from administrators
- Deploying applications with an operator to manage it makes life easier in the matter of management and maintenance
- A lot of companies already adopted operators , you can check out Operatorhub.io

What Operators are made of

- Operator Extends the kubernetes control plane and api by creating new CR’s
- A CR is short for a custom resource and it is a structured data provided to the operator controler to watch
- CRD(custom resource definition) is the schema for the CR which actually ix a resource that extends the k8s api
- CRDS only extends the cluster which they are exists at
- So Simply speaking : operator adds the following:
  - a new endpoint to the api , called a CR (custom resource)
  - a control plane component that manage that resource

What does it mean creating an operator

1. creating a CRD
2. providing a program that runs in a loop watching CRs of that kind.
3. response to changes in the CR in a specific way to the application
    1. Scaling
    2. Upgrading
    3. Managing software


Kubectl apply -f vitualservice.yaml
Control plane
Controller manager
Api Server
Scheduler
Istio operator
Virtual service (CR)
node
pod
proxy
app
pod
proxy
app
pod
proxy
app

Etcd operator example

- Etcd is a lightweight distributed key value store database
- Adding new etcd member to the cluster is not trivial as just running a pod
- To manage etcd requires a big deal of knowledge by an admin , for example : How to join a new node to the cluster,backup and configuration , and upgrade.
- The etcd operator holds that complexity
- If one of an etcd member in the cluster fails,reconciliation is triggered, and the operator heals the cluster
- The etcd operator will have to take into account how to rejoin a member , whether it is a master or slave etc...

Ecd It is the underlying data store at the core of Kubernetes, and a key piece of several distributed applications

- you'll deploy the etcd Operator
- then have it create an etcd cluster according to your specifications.
- have the Operator recover from failures
- perform a version upgrade while the etcd API continues to service read and writes

- CR’s are acting like a native k8s resource,
- You define a CRD and k8s is responsible to expose it for you through the api
- Using the api you can read and set data of your cusom resource
- Usually you will interact with k8s api via kubectl
- Kubernetes use CR’s as a mechanism to expose new api objects
- When defining CRD for the CR’s you must follow the regular k8s conventions using .spec and .status etc…
- The Cr’s are being continually monitored by the operator custom controller that in turn can also create other api objects and more components to support it’s operation

operator scope

Kubernetes support namespaces in which only one resource name of a specific type can exist

You can restrict your operator for a namespace to allow separate teams to create instances of it, upgrade and test each of the instances without affecting the others

There are other situations that you can create operators with a cluster scope for to control the entire cluster , istio is a good example of it

Operators like any other k8s resources follows the RBAC rules they are assigned with, and whether they are namespace scoped or cluster scoped they need to have the right Role assigned (Role , or ClusterRole)

- There is a lot of complexity and boilerplate code involved in creating an operator

- Operator-sdk is a framework that allows us to easily create an operator in golang , and distribute our app, by automating almost everything

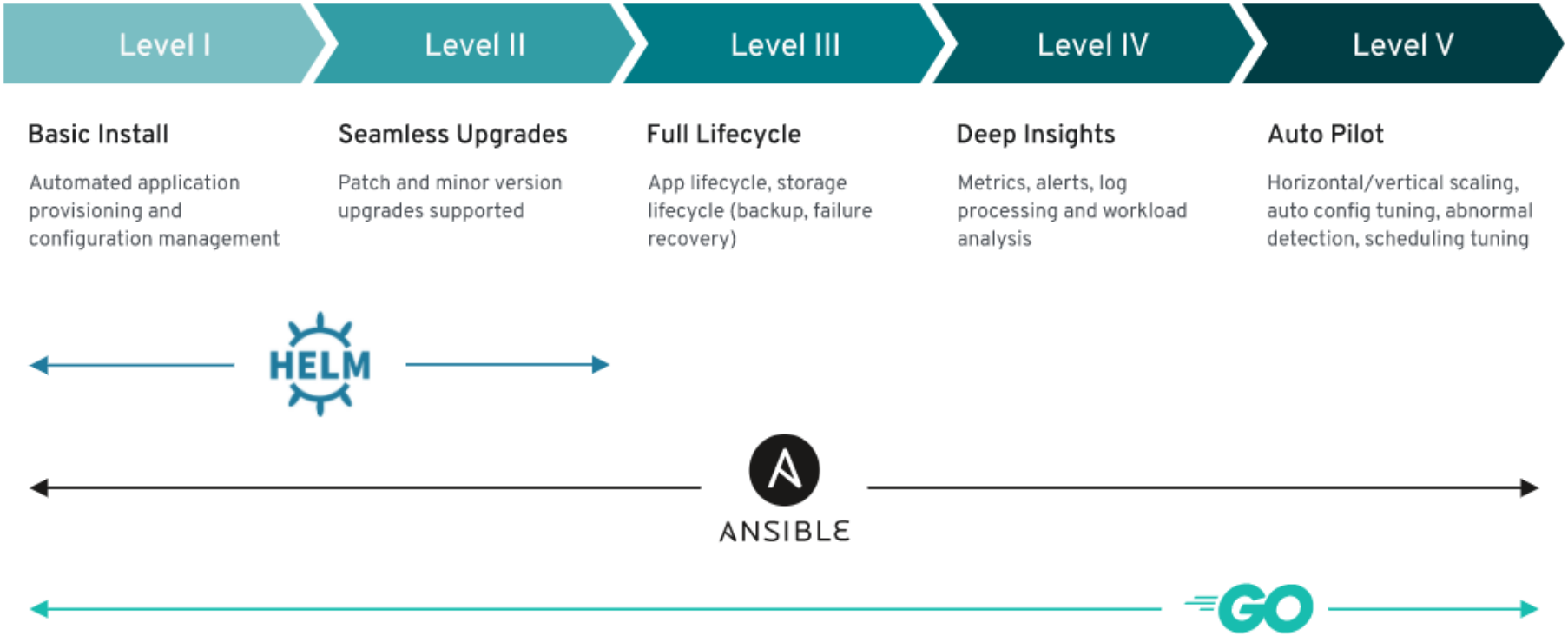
- There are other good ways to get started with developing an operator , kubebuilder is one of them

- The sdk use controller-runtime k8s libraries which are a set of libraries supporting creation of k8s controller

Why operator sdk

- operator -sdk provides us an additional important feature which is the OLM
- the sdk add OLM integration points in the code
- It also has support for creating operators using helm or ansible
- Support for metering

maturity model



Steps for building our operator

1. Create the CRD’s as the model for our app, that also provide the api to interact with

1. Create the controller - each CRD needs to have one controller that watch it for changes , and handles them accordingly

1. Build the Operator image and create the Kubernetes manifests to deploy the Operator and its RBAC components (service accounts, roles, etc.).

- Download from https://github.com/operator-framework/operator-sdk/releases

- Install :

```
$ wget https://github.com/operator-framework/operator-sdk/releases/download/v0.../...
$ sudo mv operator-sdk-v... /usr/local/bin/operator-sdk
$ sudo chmod +x /usr/local/bin/operator-sdk
```

bootstrap the operator

$ operator-sdk new gameserver --repo github.com/motiso/gameserver
//Point the repo argument to the repository you want to use for your Go module.

INFO[0000] Creating new Go operator 'gameserver'.
INFO[0000] Created go.mod
INFO[0000] Created tools.go
INFO[0000] Created cmd/manager/main.go
INFO[0000] Created build/Dockerfile
INFO[0000] Created build/bin/entrypoint
INFO[0000] Created build/bin/user_setup
INFO[0000] Created deploy/service_account.yaml
INFO[0000] Created deploy/role.yaml
INFO[0000] Created deploy/role_binding.yaml
INFO[0000] Created deploy/operator.yaml
INFO[0000] Created pkg/apis/apis.go
INFO[0000] Created pkg/controller/controller.go
INFO[0000] Created version/version.go
INFO[0000] Created .gitignore
INFO[0000] Validating project
INFO[0022] Project validation successful.
INFO[0022] Project creation complete.

Adding new CRD's to the operator

We can use the add api command to add new CRD's to the operator

$ operator-sdk add api --api-version=gameserver.com/v1 --kind=Gameserver

```
INFO[0000] Generating api version gameserver.com/v1 for kind Gameserver.
INFO[0000] Created pkg/apis/gameserver/group.go
INFO[0001] Created pkg/apis/gameserver/v1/gameserver_types.go
INFO[0002] Created pkg/apis/addtoscheme_gameserver_v1.go
INFO[0002] Created pkg/apis/gameserver/v1/register.go
INFO[0002] Created pkg/apis/gameserver/v1/doc.go
INFO[0002] Created deploy/crds/gameserver.com_v1_gameserver_cr.yaml
INFO[0002] Running deepcopy code-generation for Custom Resource group versions: [gameserver:[v1], ]
INFO[0010] Code-generation complete.
INFO[0010] Running CRD generator.
INFO[0011] CRD generation complete.
INFO[0011] API generation complete.
INFO[0011] API generation complete
```

The above Command generates a lot of files that are based on the kind and api version that we gave it

```
├── build
│   ├── Dockerfile
│   └── bin
│       ├── entrypoint
│       └── user_setup
├── cmd
│   └── manager
│       └── main.go
├── deploy
│   ├── crds
│   │   ├── gameserver.com_gameservers_crd.yaml
│   │   └── gameserver.com_v1_gameserver_cr.yaml
│   ├── operator.yaml
│   ├── role.yaml
│   ├── role_binding.yaml
│   └── service_account.yaml
├── go.mod
├── go.sum
├── pkg
│   ├── apis
│   │   ├── addtoscheme_gameserver_v1.go
│   │   ├── apis.go
│   │   └── gameserver
│   │       ├── group.go
│   │       └── v1
│   │           ├── doc.go
│   │           ├── gameserver_types.go
│   │           ├── register.go
│   │           └── zz_generated.deepcopy.go
│   └── controller
│       └── controller.go
├── tools.go
└── version
    └── version.go
```

Add the controller

- When creating the controller You need to use the same api-version and kind that you used before in order to scope the controller to a specific CRD

  ```
  $ operator-sdk add controller --api-version=gameserver.com/v1 --kind=Gameserver
  INFO[0000] Generating controller version gameserver.com/v1 for kind Gameserver.
  INFO[0000] Created pkg/controller/gameserver/gameserver_controller.go
  INFO[0000] Created pkg/controller/add_gameserver.go
  INFO[0000] Controller generation complete.
  ```

- The most interesting file in all the newly generated files are the one marked with red which is the controller file that is associated with the specific kind

- The controller is responsible for:
  - “reconciling” a specific resource
  - establish one or more “watches”

Let’s watch the Watch

```
// Watch for changes to primary resource Gameserver
    err = c.Watch(&source.Kind{Type: &gameserverv1.Gameserver{}}, &handler.EnqueueRequestForObject{})
    if err != nil {
        return err
    }
    err = c.Watch(&source.Kind{Type: &corev1.Pod{}}, &handler.EnqueueRequestForOwner{
        IsController: true,
        OwnerType:    &gameserverv1.Gameserver{},
    })
    if err != nil {
        return err
    }
```

Reconcile

The reconcile function main and only purpose , is make sure that the desired state is kept , and to update kubernetes system accordingly

Usually this function starts by retrieving the primary resource that triggered directly or indirectly(because one or more of it’s child changed) the reconcile request

```
func (r *ReconcileGameserver) Reconcile(request reconcile.Request) (reconcile.Result, error) {
   reqLogger := log.WithValues("Request.Namespace", request.Namespace, "Request.Name", request.Name)
   reqLogger.Info("Reconciling Gameserver")

   // Fetch the Gameserver instance
   instance := &gameserverv1.Gameserver{}
   err := r.client.Get(context.TODO(), request.NamespacedName, instance)
   if err != nil {
       if errors.IsNotFound(err) {
           // Request object not found, could have been deleted after reconcile request.
           // Owned objects are automatically garbage collected. For additional cleanup logic use finalizers.
           // Return and don't requeue
           return reconcile.Result{}, nil
       }
       // Error reading the object - requeue the request.
       return reconcile.Result{}, err
   }
```

Possible return values from the Reconcile function

return reconcile.Result{}, nil
The reconcile process finished with no errors and does not require another pass through the reconcile loop.

return reconcile.Result{}, err
The reconcile failed due to an error and Kubernetes should requeue it to try again.

return reconcile.Result{Requeue: true}, nil
The reconcile did not encounter an error, but Kubernetes should requeue it to run for another iteration.

return reconcile.Result{RequeueAfter: time.Second*5}, nil
Similar to the previous result, but this will wait for the specified amount of time before requeuing the request

Updating state by the controller

- When we wish to update kubernetes with the updated state we use the client.Client interface to call back to k8s with the updated state

- To do that we need to create the resource for example POD ,SECRET etc.. and then we need to instantiate that type and call back to k8s

- Kubernetes GO client API defines the specifications for how to instantiate the different k8s resources

```
// Define a new Pod object
   pod := newPodForCR(instance)
   // Set Gameserver instance as the owner and controller
   if err := controllerutil.SetControllerReference(instance, pod, r.scheme); err != nil {
       return reconcile.Result{}, err
   }
   // Check if this Pod already exists
   found := &corev1.Pod{}
   err = r.client.Get(context.TODO(), types.NamespacedName{Name: pod.Name, Namespace: pod.Namespace}, found)
   if err != nil && errors.IsNotFound(err) {
       reqLogger.Info("Creating a new Pod", "Pod.Namespace", pod.Namespace, "Pod.Name", pod.Name)
       err = r.client.Create(context.TODO(), pod)
       if err != nil {
           return reconcile.Result{}, err
       }
```

Running operator locally

- Deploy the Crd's **kubectl apply -f deploy/crds/*_crd.yaml**
- Start the operator in local mode - **operator-sdk run local --namespace default**
- Deploy an example resource (the sdk generates an example cr) **kubectl apply -f deploy/crds/*_cr.yaml**
- Stop a running operator using **control-c**

Thank you !