Golang Programming

Building RESTful APIs, Web Applications and Clients with Go

Where to Find The Code and Materials?

https://github.com/iproduct/coursego

Agenda for This Session

- Web Applications and Services
- URIs and URI templates
- Asynchronous JavaScript and XML (AJAX) & Fetch API
- Multimedia and Hypermedia – basic concepts
- Service Oriented Architecture (SOA),
- REpresentational State Transfer (REST)
- Hypermedia As The Engine Of Application State (HATEOAS)
- New Link HTTP header
- Richardson Maturity Model of Web Applications
- Cross Origin Resource Sharing
- Domain Driven Design - DDD
- Layered and Hexagonal Architectures

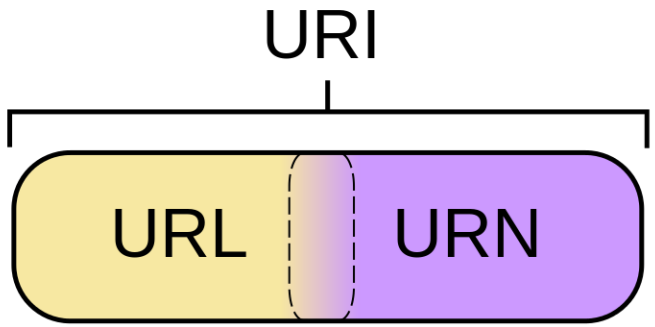
Web Applications and Services

Types of Web Applications

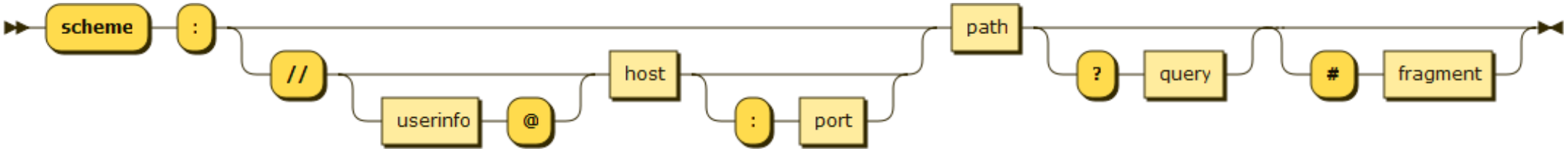
- Web Sites – presenting interactive UI
  - Static Web sites – show the same information for all visitors – can include hypertext, images, videos, navigation menus, etc.
  - Dynamic Web sites – change and tune the content according to the specific visitor
    - server-site – use server technologies for dynamic web content (page) generation (data comes drom DB)
    - client-side – use JavaScript and asynchronous data updates
- Web Services – managing (CRUD) data resources
  - Classical – SOAP + WSDL
  - RESTful – distributed hypermedia

URLs, IP Addresses, and Ports

- Uniform Resource Identifier - URI:
  - Uniform Resource Locator – URL
  - Uniform Resource Name – URN



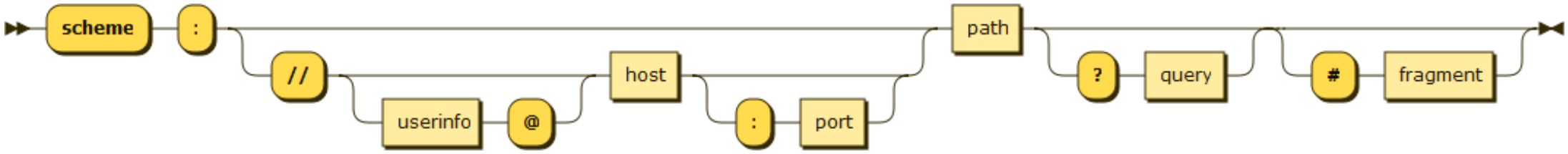
- Format:
  scheme://domain:port/path?query_string#fragment_id
- Schemas: http://, https://, file://, ftp://. news://, mailto:, telnet://



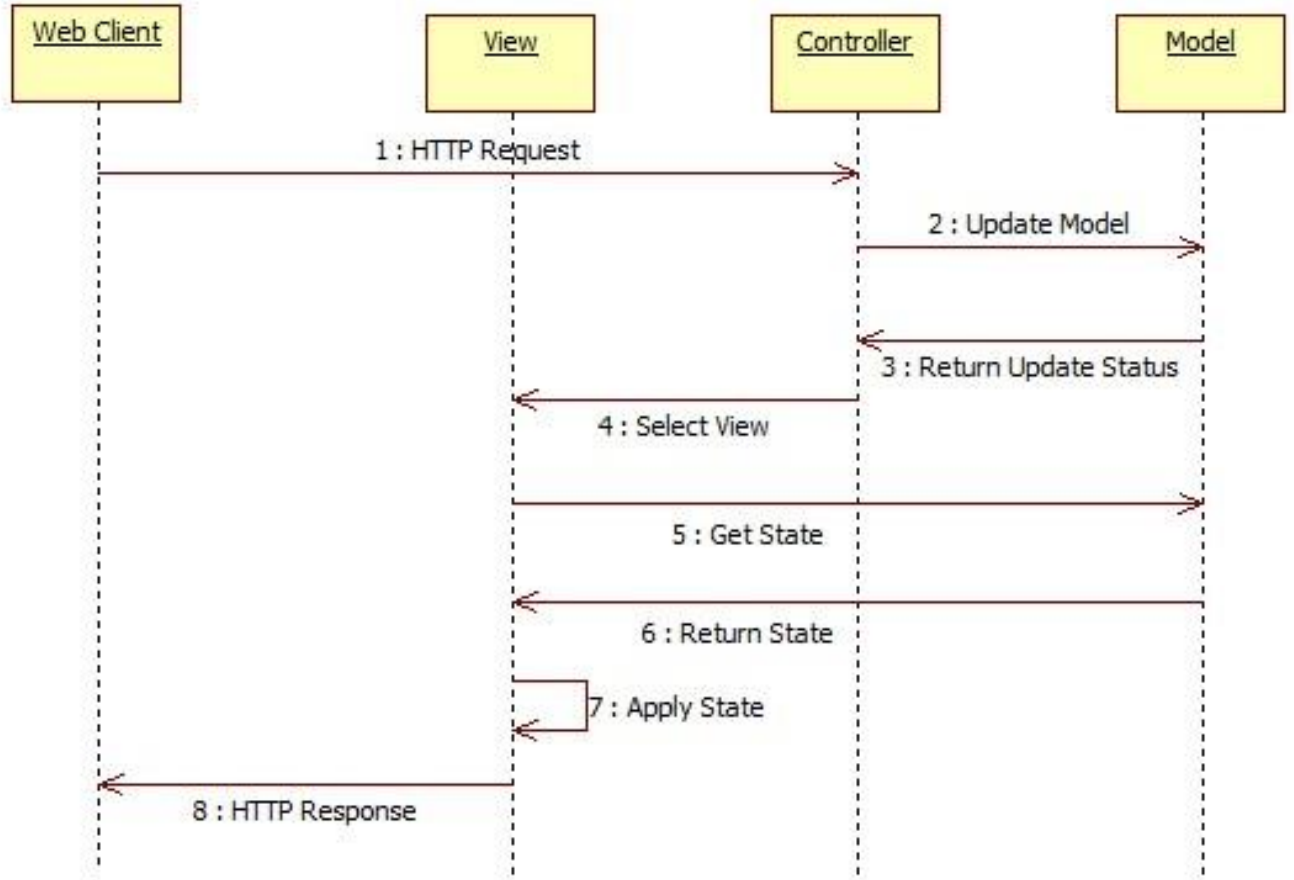
- Example:
  http://en.wikipedia.org/wiki/Uniform_resource_locator#History

Query Parameters and Fragment Ids

- URLs pointing to **dynamic resources** often include:
- **/{path_param}** – e.g.: /users/12/posts/3
- **?query_param=value** (query string) – e.g.: ? role=student&mode=edit
- **#fragment_identifier** – defines the fragment (part) of the resource we want to access, used by asynchronous javascript page loading (AJAX) applications to encode the local page state – e.g. #view=fitb&nameddest=Chapter3



Web MVC Interactions Sequence Diagram

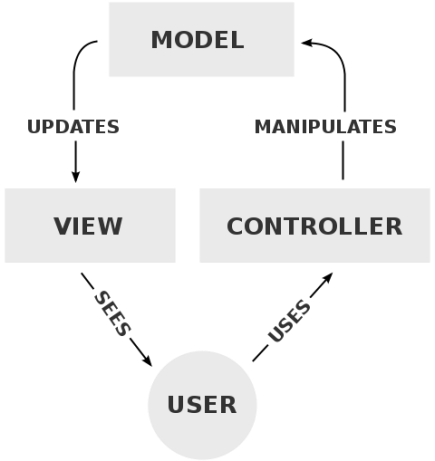


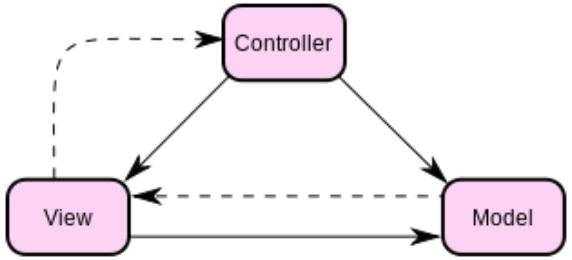
MVC Comes in Different Flavors

- What is the difference between following patterns:
- Model-View-Controller (MVC)
- Model-View-ViewModel (MVVM)
- Model-View-Presenter (MVP)
- http://csl.ensm-douai.fr/noury/uploads/20/ModelViewController.mp3

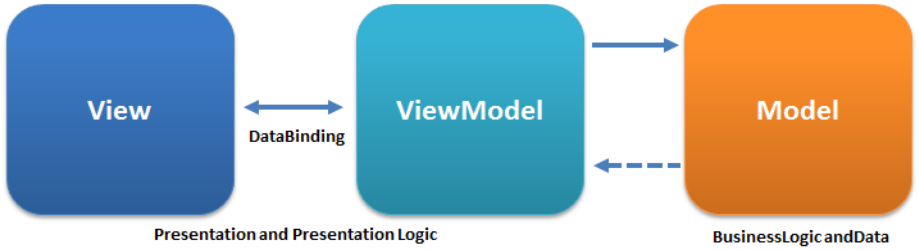
MVC Comes in Different Flavors

- MVC

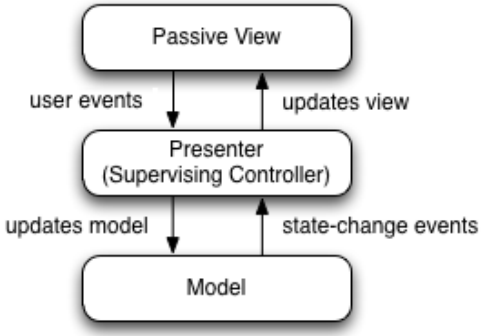




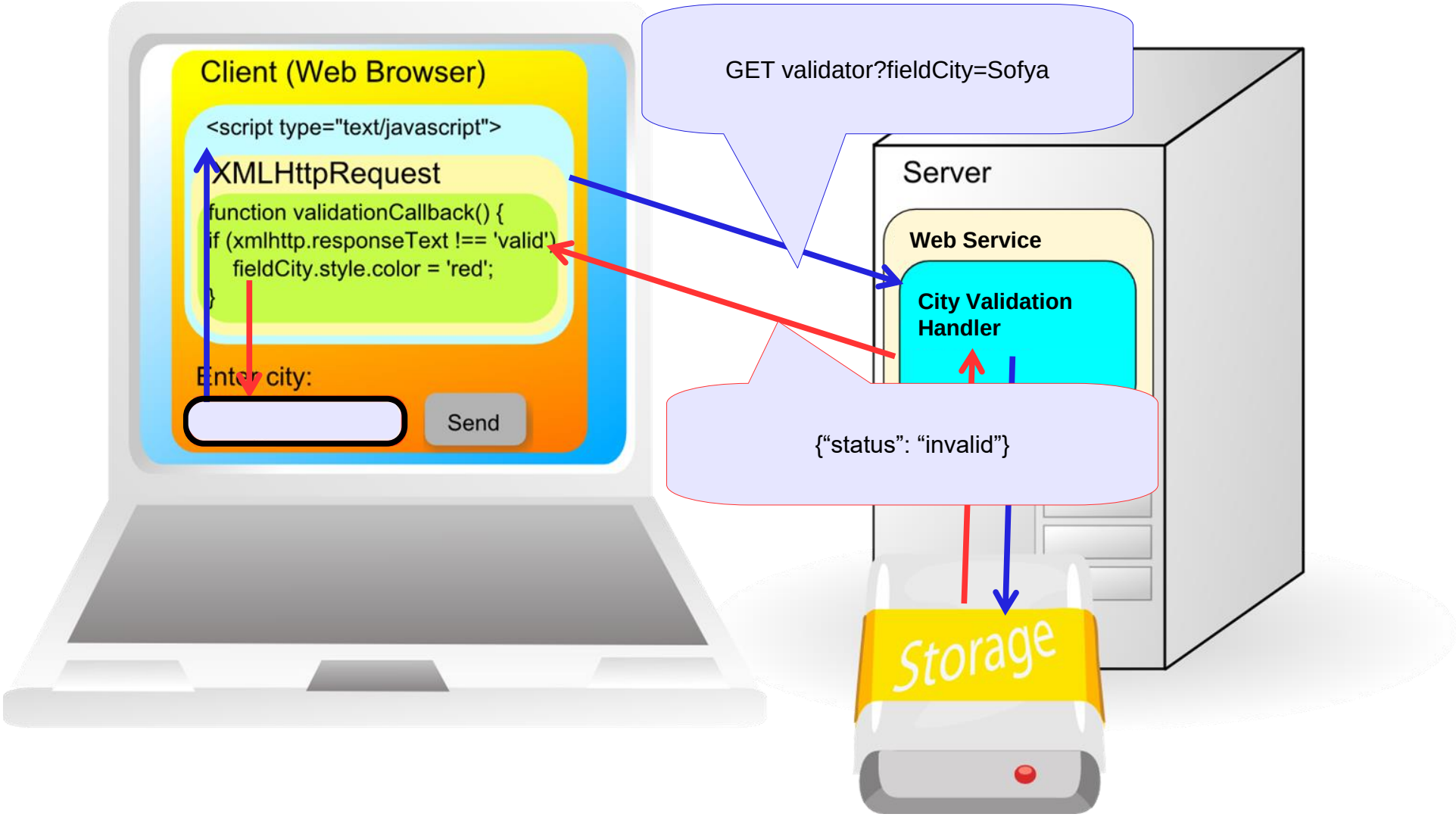
- MVVM



- MVP



Asysnchronous JavaScript and XML (AJAX)



Basic Structure of Asynchronous AJAX Request

```
if (window.XMLHttpRequest)   {// IE7+, Firefox, Safari, Chrome, Opera,
        xmlhttp=new XMLHttpRequest();
 }  else {// IE5, IE6
        xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
 }
 xmlhttp.onreadystatechange = function(){
        if (xmlhttp.readyState==4 && xmlhttp.status==200){
            callback(xmlhttp);
        }
 }
xmlhttp.open(method, url, true);
xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
xmlhttp.send(paramStr);
```

Callback function

isAsynchronous = true

XMLHttpRequest.readyState

Code	Description
1	after XMLHttpRequest.open() was called
2	HTTP response headers were successfully received
3	HTTP response content loading was started
4	HTTP response content was successfully received by the client

Fetch API

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API]

- The **Fetch API** provides an interface for fetching resources like XMLHttpRequest, but more powerful and flexible feature set.

- `Promise<Response> WorkerOrGlobalScope.fetch(input[, init])`
  - input - resource that you wish to fetch – url string or Request
  - init - custom settings that you want to apply to the request:
    - **method**: (e.g., GET, POST),
    - **headers,**
    - **body:** (Blob, BufferSource, FormData, URLSearchParams, or USVString),
    - **mode**: (cors, no-cors, or same-origin),
    - **credentials:**(omit, same-origin, or include. to automatically send cookies this option must be provided),
    - **cache**: (default, no-store, reload, no-cache, force-cache, or only-if-cached),
    - **redirect:** (follow, error or manual),
    - **referrer:** (default is client),
    - **referrerPolicy**: (no-referrer, no-referrer-when-downgrade, origin, origin-when-cross-origin, unsafe-url),
    - **integrity:** (subresource integrity value of request)

Web Page Design Recommendations

- Concentrate on users and their goals
- Implement intuitive navigation paths – design navigation before implementing the site
- Follow the web conventions and the "principle of least astonishment (POLA)":
  - Navigation system
  - Interface metaphors
  - Visual layout of elements
  - Color conventions
- Use a responsive web design CSS framework: Foundation, Blueprint, Bootstrap, Cascade Framework, Bulma, Undernet, Materialize
- More concrete recommendations you can find at Jakub Linowski's site: http://goodui.org

Interface http.Handler

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

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

```
func hello(w http.ResponseWriter, req *http.Request) {
        fmt.Fprintf(w, "hello\n")
}
func headers(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
        fmt.Fprintf(w, "Host = %q\nRemoteAddr = %q\n\n ", r.Host, r.RemoteAddr)
        for name, headers := range r.Header {
                for _, h := range headers {
                        fmt.Fprintf(w, "%v: %v\n", name, h)
                }
        }
}
func main() {
        http.HandleFunc("/hello", hello)
        http.HandleFunc("/headers", headers)
        http.ListenAndServe(":8080", nil)
}
```

Example: HTTP QR Link Generator

Example: HTTP - QR Link Generator - Code

```
import ( "flag“; "html/template“; "log“; "net/http")

var addr = flag.String("addr", ":8080", "http service address")
var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
        flag.Parse()
        http.Handle("/", http.HandlerFunc(QR))
        err := http.ListenAndServe(*addr, nil)
        if err != nil {
                log.Fatal("ListenAndServe:", err)
        }
}

// QR http handler function return the HTML template with QR code for the input link
func QR(w http.ResponseWriter, req *http.Request) {
        templ.Execute(w, req.FormValue("s"))
}
```

HTML Template [https://golang.org/pkg/html/template/]

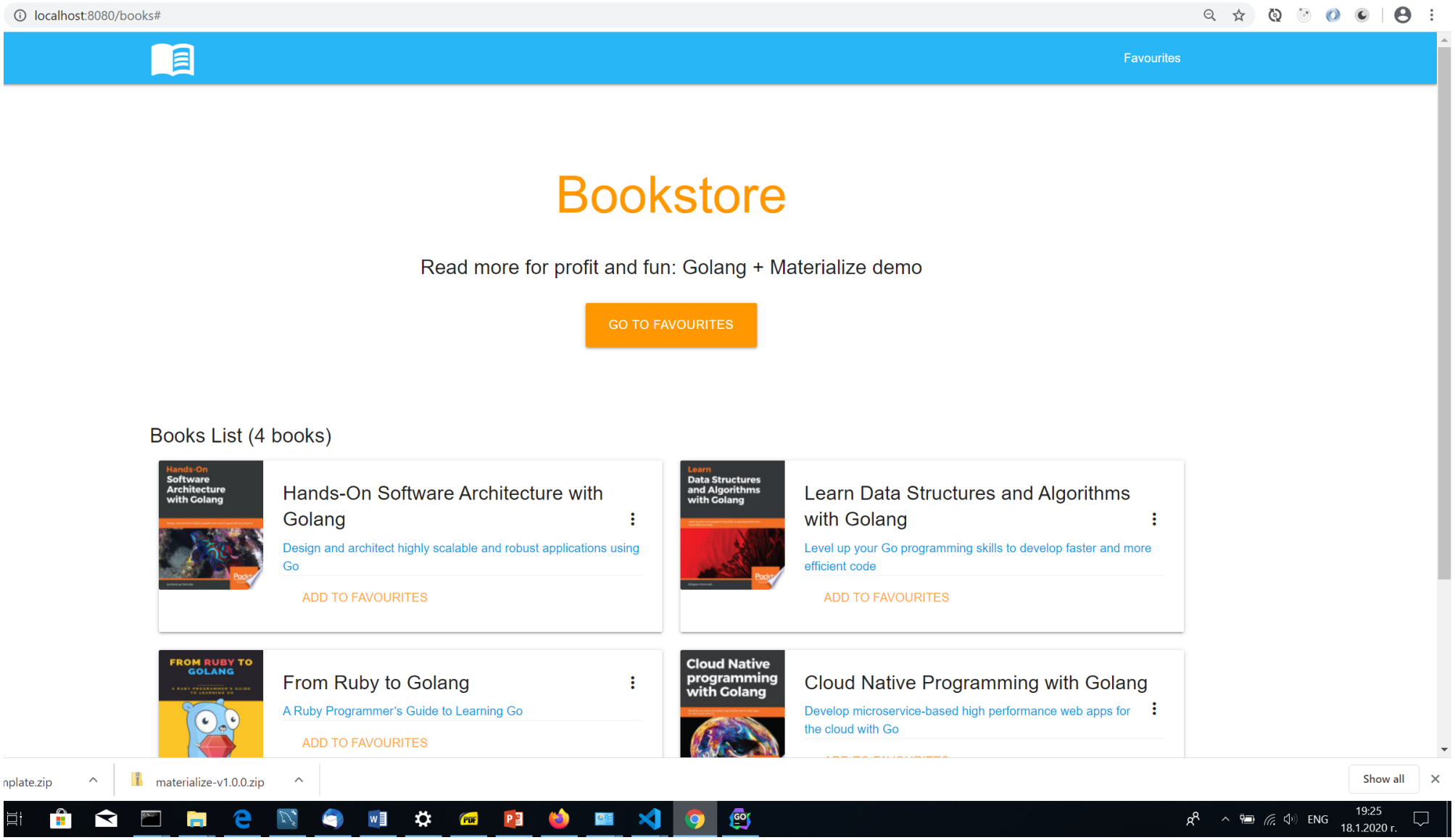
```
const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{.}}" />
<br>
{{.}}
<br>
<br>
{{end}}
<form action="/" name=f method="GET"><input maxLength=1024 size=70
name=s value="" title="Text to QR Encode"><input type=submit
value="Show QR" name=qr>
</form>
</body>
</html>
`
```

Text Templates [https://golang.org/pkg/text/template/]

```
import ("html/template"; "log"; "os")
const textTempl =
`{{len .}} issues:
{{range .}}----------------------------------------
ID: {{.ID}}
Title: {{.Title | printf "%.64s"}}
{{end}}`

func main() {
        tmpl := template.New("report")
        tmpl, err := tmpl.Parse(textTempl)
        if err != nil {
                log.Fatal("Error Parsing template: ", err)
                return
        }
        err1 := tmpl.Execute(os.Stdout, goBooks)
        if err1 != nil {
                log.Fatal("Error executing template: ", err1)
        }
}
```

```
3 books:

----------------------------------------

ID: fmd-DwAAQBAJ

Title: Hands-On Software Architecture with Golang

----------------------------------------

ID: o86PDwAAQBAJ

Title: Learn Data Structures with Golang

----------------------------------------

ID: xfPEDwAAQBAJ

Title: From Ruby to Golang
```

More Examples: Bookstore Web App



More Examples: Bookstore Web App

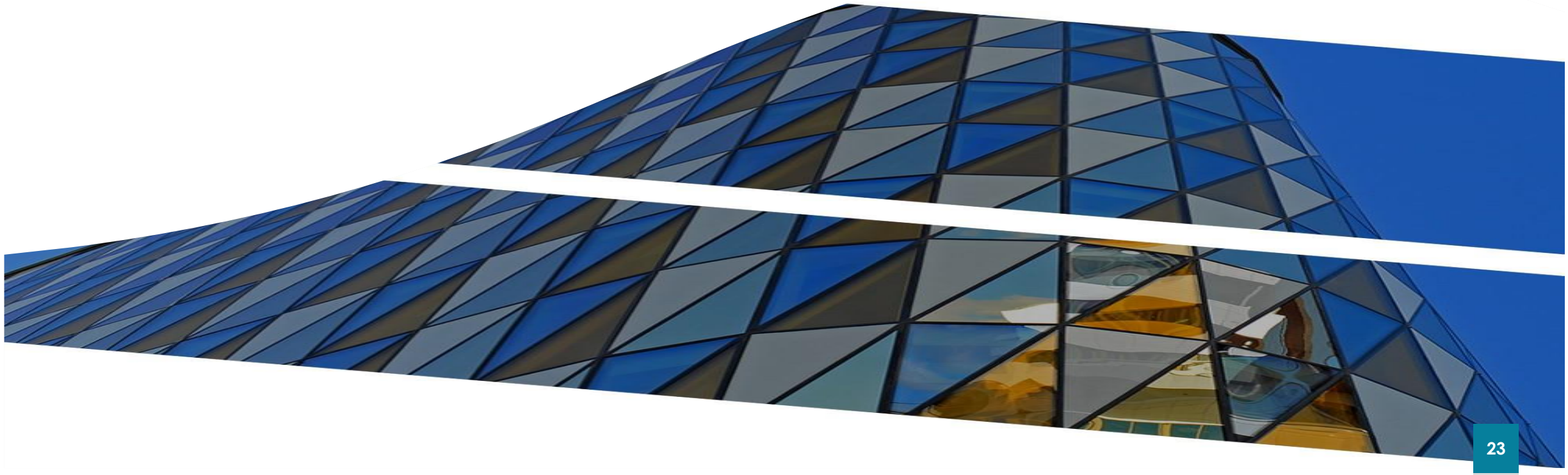
https://github.com/iproduct/coursego/tree/master/http/bookstore-simple

https://github.com/iproduct/coursego/tree/master/http/bookstore

html/template cheetsheet:
https://curtisvermeeren.github.io/2017/09/14/Golang-Templates-Cheatsheet

SOA & REST

Developing horizontally scalable web services



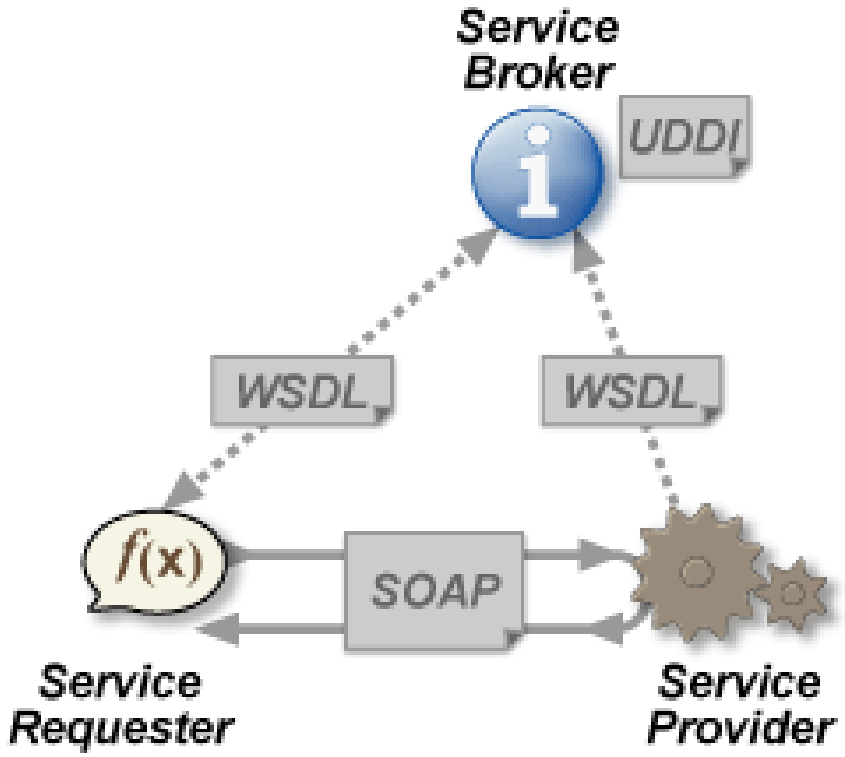
Service Oriented Architecture (SOA) – Definitions

Thomas Erl: SOA represents an open, agile, extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services. SOA can establish an abstraction of business logic and technology, resulting in a loose coupling between these domains. SOA is an evolution of past platforms, preserving successful characteristics of traditional architectures, and bringing with it distinct principles that foster service-orientation in support of a service-oriented enterprise. SOA is ideally standardized throughout an enterprise, but achieving this state requires a planned transition and the support of a still evolving technology set.

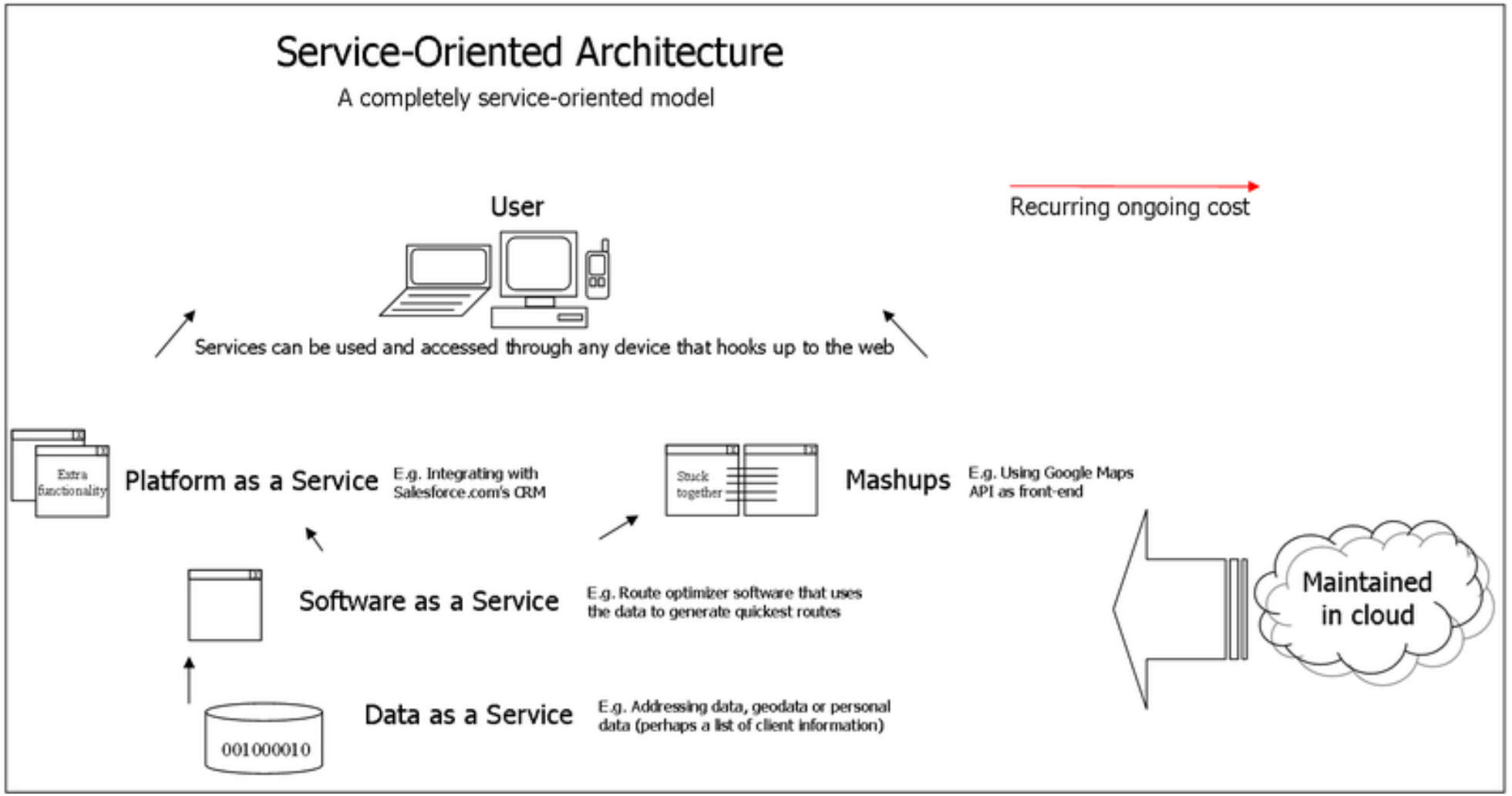
References: Erl, Thomas. serviceorientation.org – About the Principles, 2005–06

Classical Web Services - SOAP + WSDL

- Web Services are:
- components for building distributed applications in SOA architectural style
- communicate using open protocols
- are self-descriptive and self-content
- can be searched and found using UDDI or ebXML registries (and more recent specifications – WSIL & **Semantic Web Services**)



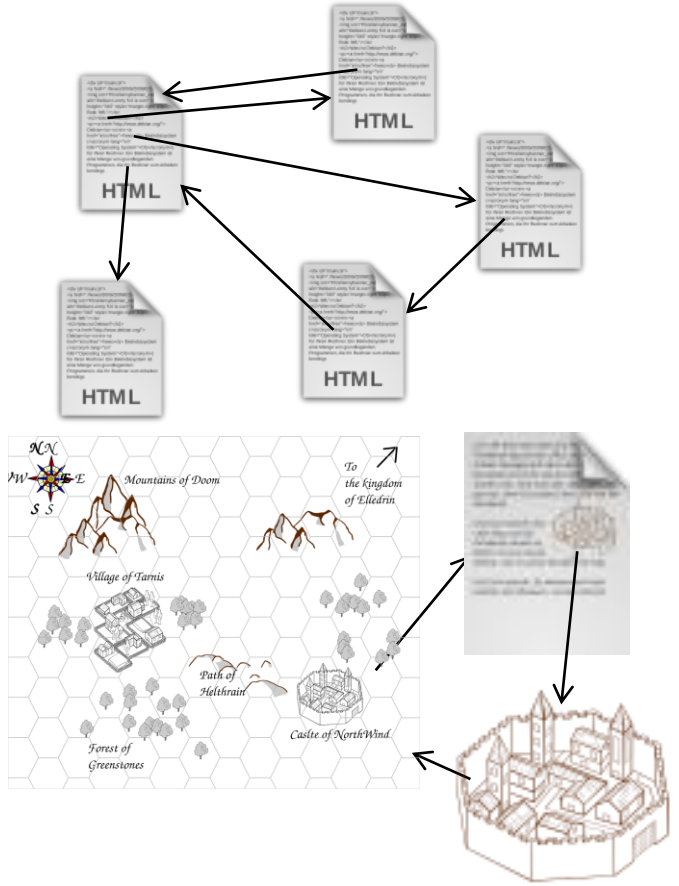
Service Oriented Architecture (SOA)



Source: http://en.wikipedia.org/wiki/File:SOA_Detailed_Diagram.png,
Author: JamesLWilliams2010, License: Creative Commons Attribution 3.0 Unported

Hypertext & Hypermedia

- Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text
- HTTP is the protocol to exchange or transfer hypertext
- Hypermedia - extension of the term hypertext, is a nonlinear medium of information which includes multimedia (text, graphics, audio, video, etc.) and hyperlinks of different media types (e.g. image or animation/video fragment can be linked to a detailed description).



REST Architecture

According to Roy Fielding [Architectural Styles and the Design of Network-based Software Architectures, 2000]:

- Client-Server
- Stateless
- Uniform Interface:
  - Identification of resources
  - Manipulation of resources through representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)
- Layered System
- Code on Demand (optional)

Representational State Transfer (REST)

- REpresentational State Transfer (REST) is an architecture for accessing distributed hypermedia web-services
- The resources are identified by URIs and are accessed and manipulated using an HTTP interface base methods (GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH)
- Information is exchanged using representations of these resources
- Lightweight alternative to SOAP+WSDL -> HTTP + Any representation format (e.g. JavaScript Object Notation – JSON)

HTTP Request Structure

GET */context/Servlet* **HTTP/1.1**

Host: *Client_Host_Name*

Header2: *Header2_Data*

…

HeaderN: *HeaderN_Data*

<Празен ред>

POST */context/Servlet* **HTTP/1.1**

Host: *Client_Host_Name*

Header2: *Header2_Data*

…

HeaderN: *HeaderN_Data*

<Празен ред>

POST_Data

HTTP Response Structure & JSON

HTTP/1.1 200 OK

Content-Type: application/json

Header2: Header2_Data

…

HeaderN: HeaderN_Data

<Празен ред>

```
[{  "id":1,
    "name":"Novelties in Java EE 7 ...",
    "description":"The presentation is ...",
    "created":"2014-05-10T12:37:59",
    "modified":"2014-05-10T13:50:02",
},
{  "id":2,
    "name":"Mobile Apps with HTML5 ...",
    "description":"Building Mobile ...",
    "created":"2014-05-10T12:40:01",
    "modified":"2014-05-10T12:40:01",
}]
```

Representational State Transfer (REST)

- Identification of resources – URIs
- Representation of resources – e.g. HTML, XML, JSON, etc.
- Manipulation of resources – through these representations
- Self-descriptive messages - Internet media type (MIME type) provides enough information to describe how to process the message. Responses also explicitly indicate their cacheability.
- Hypermedia as the engine of application state (aka **HATEOAS**)
- Application contracts are expressed as **media types** and [semantic] link relations (**rel** attribute - RFC5988, "Web Linking")

Multipurpose Internet Mail Extensions (MIME)

- Different types of media are represented using different text/binary encoding formats – for example:
  - Text -> plain, html, xml ...
  - Image (Graphics) -> gif, png, jpeg, svg ...
  - Audio & Video -> mp3, ogg, webm ...
- Multipurpose Internet Mail Extensions (MIME) allows the client to recognize how to handle/present the particular multimedia asset/node:
  - Ex.: Content-Type: text/plain

Media Type

Media SubType (format)

- More examples for standard MIME types: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types
- Vendor specific media (MIME) types: application/vnd.*+json/xml

Hypermedia As The Engine Of Application State (HATEOAS) – New Link Header (RFC 5988) Example

```
Content-Length →1656

Content-Type →application/json

Link →<http://localhost:8080/polling/resources/polls/629>; rel="prev";
  type="application/json"; title="Previous poll",
  <http://localhost:8080/polling/resources/polls/632>; rel="next";
  type="application/json"; title="Next poll",
  <http://localhost:8080/polling/resources/polls>; rel="collection";
  type="application/json"; title="Polls collection",
  <http://localhost:8080/polling/resources/polls>; rel="collection up";
  type="application/json"; title="Self link",
  <http://localhost:8080/polling/resources/polls/630>; rel="self"
```

Advantages of REST

- Scalability of component interactions – through layering the client server-communication and enabling load-balancing, shared caching, security policy enforcement;
- Generality of interfaces – allowing simplicity, reliability, security and improved visibility by intermediaries, easy configuration, robustness, and greater efficiency by fully utilizing the capabilities of HTTP protocol;
- Independent development and evolution of components – dynamic evolvability of services, without breaking existing clients.
- Fault tolerant, Recoverable, Secure, Loosely coupled

Richardson's Maturity Model of Web Services

According to **Leonard Richardson** [Talk at QCon, 2008 - http://www.crummy.com/writing/speaking/2008-QCon/act3.html]:

- Level 0 – POX: Single URI (XML-RPC, SOAP)
- Level 1 – Resources: Many URIs, Single Verb (URI Tunneling)
- Level 2 – HTTP Verbs: Many URIs, Many Verbs (CRUD – e.g Amazon S3)
- Level 3 – Hypermedia Links Control the Application State = HATEOAS (Hypertext As The Engine Of Application State) === **truely** RESTful Services

Simple Example: URLs + HTTP Methods

Uniform Resource Locator (URL)	GET	PUT	POST	DELETE
Collection, such as http://api.example.com/comments/	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	Delete the entire collection.
Individual element, such as http://api.example.com/comments/11	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it does not exist, create it.	Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Source: https://en.wikipedia.org/wiki/Representational_state_transfer

Web Application Description Language (WADL)

- XML-based file format providing machine-readable description of HTTP-based web application resources – typically RESTful web services
- WADL is a W3C Member Submission
  - Multiple resources
  - Inter-connections between resources
  - HTTP methods that can be applied accessing each resource
  - Expected inputs, outputs and their data-type formats
  - XML Schema data-type formats for representing the RESTful resources
- But WADL resource description is **static**

Cross-Origin Resource Sharing (CORS)

- Enables execution of cross-domain requests for web resources by allowing the server to decide which scripts (from which domains – Origin) are allowed to receive/manipulate such resources, as well as which HTTP Methods (GET, POST, PUT, DELTE, etc.) are allowed.

- In order to implement this mechanism, when the HTTP methods differs from simple GET, a prefight OPTIONS request is issued by the web browser, in response to which the server returns the allowed methods, custom headers, etc. for the requested web resource and script Origin.

HTTP Headers with CORS Simple Requests

- HTTP GET simple request

GET /crossDomainResource/ HTTP/1.1

Referer: http://sample.com/crossDomainMashup/

Origin: http://sample.com

- HTTP GET response

Access-Control-Allow-Origin: http://sample.com

Content-Type: application/xml

…

HTTP Headers with CORS POST/PUT/DELETE Requests

- HTTP OPTIONS preflight request

OPTIONS /crossDomainPOSTResource/ HTTP/1.1

Origin: http://sample.com

Access-Control-Request-Method: POST

Access-Control-Request-Headers: MYHEADER

- HTTP response

HTTP/1.1 200 OK

Access-Control-Allow-Origin: http://sample.com

Access-Control-Allow-Methods: POST, GET, OPTIONS

Access-Control-Allow-Headers: MYHEADER

Access-Control-Max-Age: 864000

RESTful Patterns and Best Practices

According to **Cesare Pautasso** [http://www.jopera.org/files/SOA2009-REST-Patterns.pdf]:

- Uniform Contract
- Content Negotiation
- Entity Endpoint
- Endpoint Redirection
- Distributed Response Caching
- Entity Linking
- Idempotent Capability

REST Antipatterns and Worst Practices

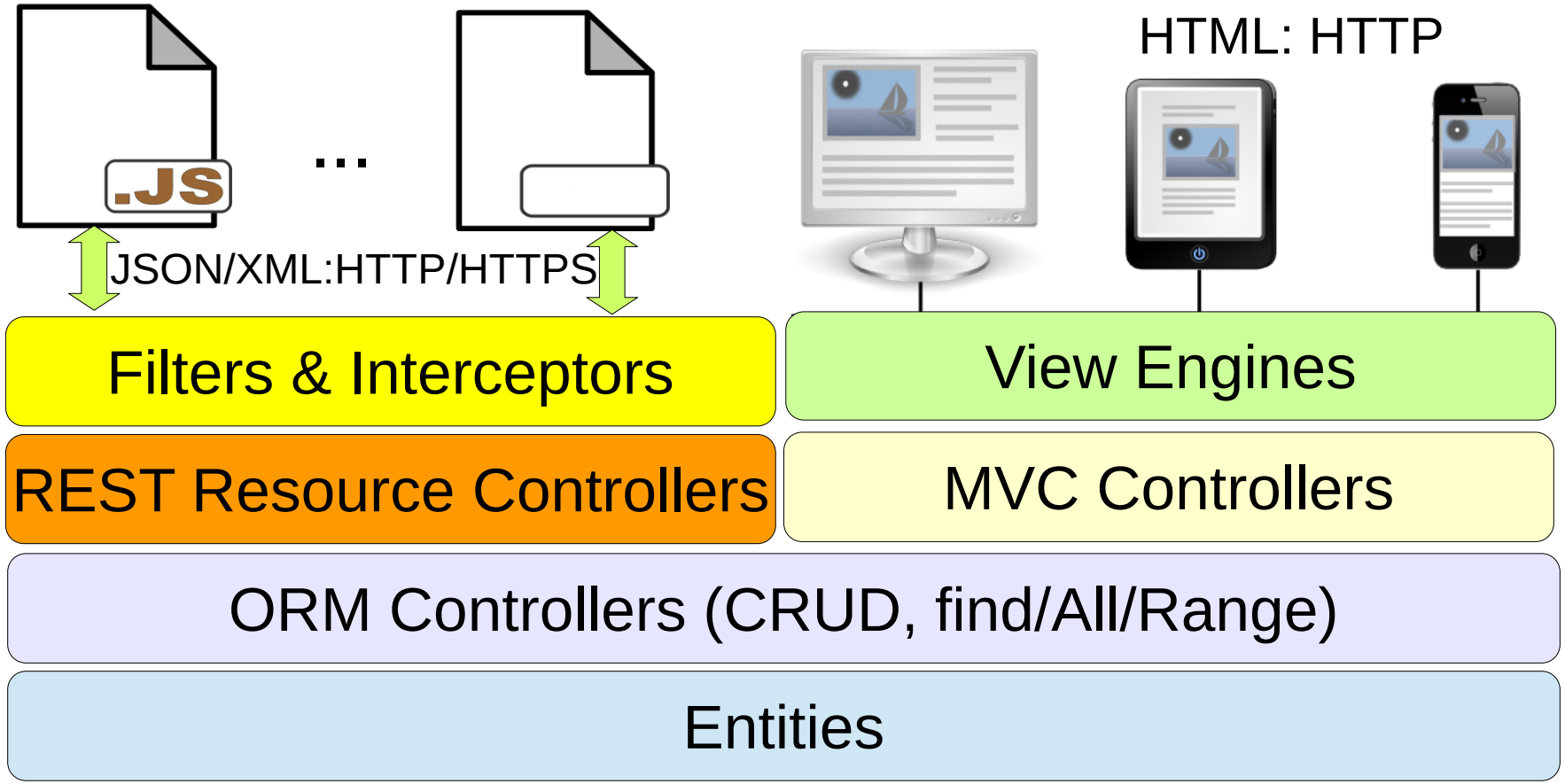
According to **Jacob Kaplan-Moss** [http://jacobian.org/writing/rest-worst-practices/]:

- Conflating models and resources
- Hardcoded authentication
- Resource-specific output formats
- Hardcoded output formats
- Weak HTTP method support (e.g. tunell everything through GET/POST)
- Improper use of links
- Couple the REST API to the application

Web Service Architectures

Coping with the Complexity – Domain Driven Design

N-Tier Web Architectures



Domain Driven Design (DDD)

We need tools to cope with the complexity inherent in most real-world design problems.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

Domain Driven Design (DDD) – back to basics: domain objects, data and logic.

Described by Eric Evans in his book:
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

Domain Driven Design (DDD)

Main concepts:

- Entities, value objects and modules
- Aggregates and Aggregate Roots [Haywood]:
  value < entity < aggregate < module < BC
- Repositories, Factories and Services:
  application services <-> **domain services**
- Separating interface from implementation

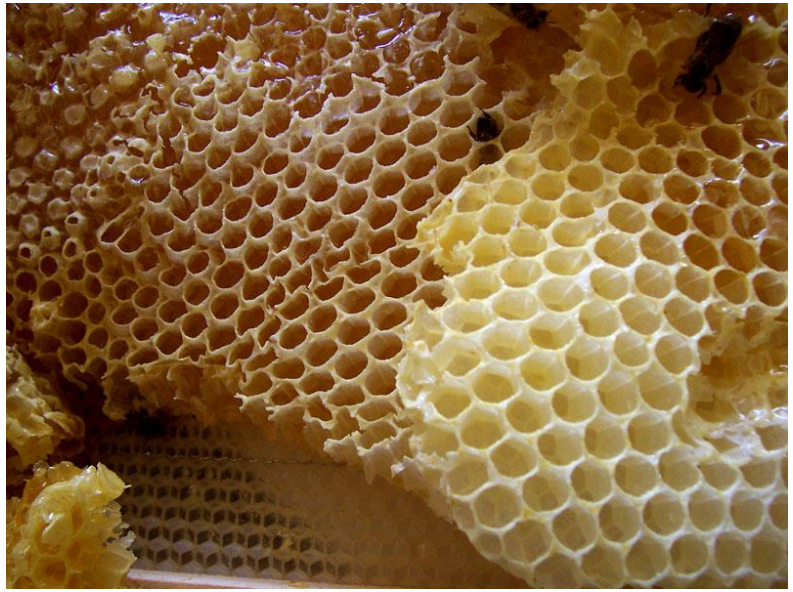
Domain Driven Design (DDD) & Microservices

- Ubiquitous language and Bounded Contexts
- DDD Application Layers:
- Infrastructure, Domain, Application, Presentation
- Hexagonal architecture :

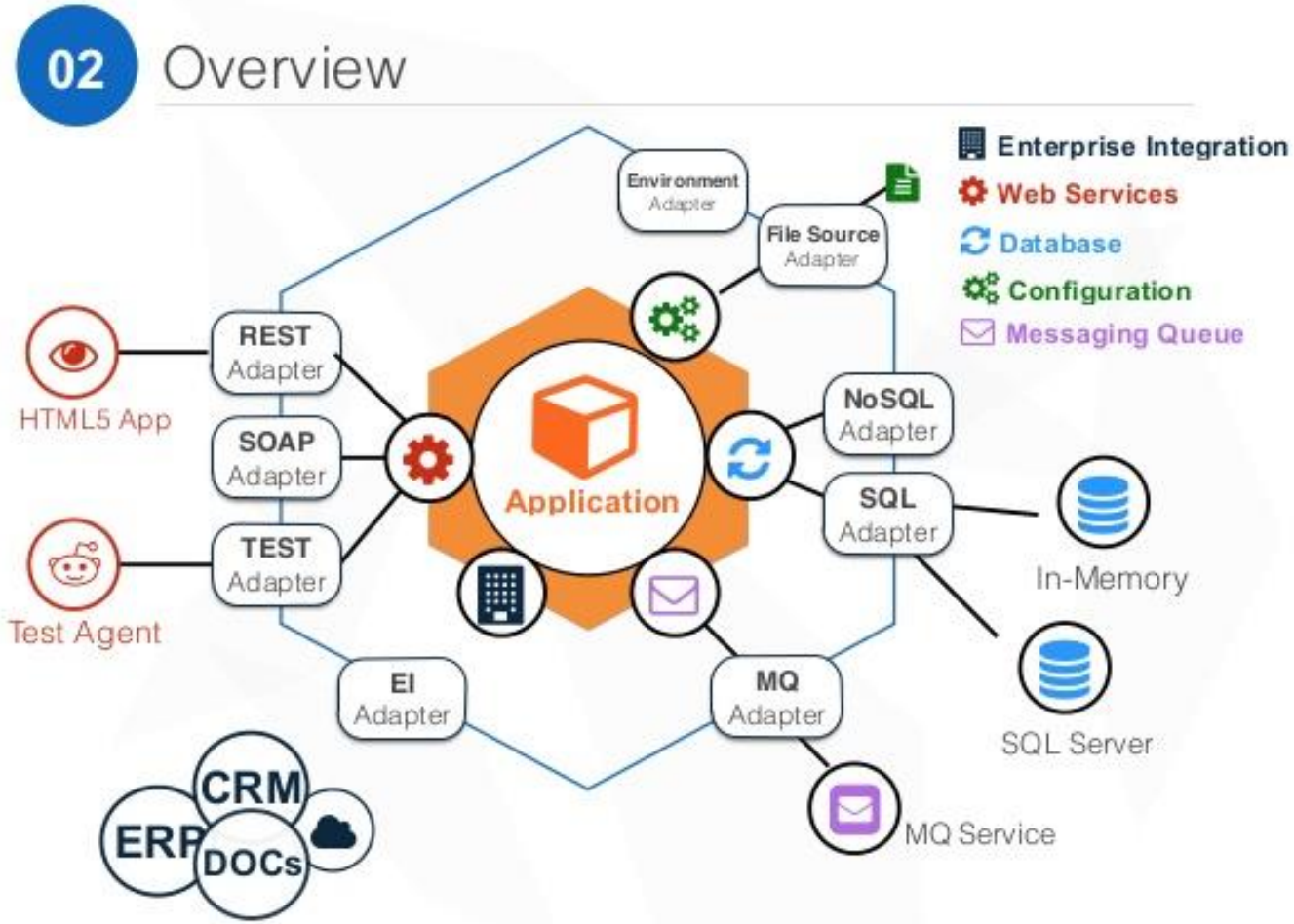
  OUTSIDE <-> transformer <->

  (application <-> domain)

  [A. Cockburn]



Hexagonal Architecture



Source: https://www.slideshare.net/fabricioepa/hexagonal-architecture-for-java-applications

Hexagonal Architecture Design Principles

- Allows an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.
- As events arrive from the outside world at a port, a technology-specific adapter converts it into a procedure call or message and passes it to the application
- Application sends messages through ports to adapters, which signal data to the receiver (human or automated)
- The application has a semantically sound interaction with all the adapters, without actually knowing the nature of the things on the other side of the adapters

Source: http://alistair.cockburn.us/Hexagonal+architecture

JSON Marshalling and Unmarshalling

```
// Structs --> JSON
data, err := json.Marshal(goBooks)
if err != nil {
        log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)

// Prettier formatting
data, err = json.MarshalIndent(goBooks, "", "     ")
if err != nil {
        log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)

// JSON -> structs
var books []Book
if err := json.Unmarshal(data, &books); err != nil {
        log.Fatalf("JSON unmarshaling failed: %s", err)
}
fmt.Println("AFTER UNMARSHAL:\n", books)
```

How to Unmarshal HTTP Request JSON Body

```
1) defer r.Body.Close()
   buf := new(bytes.Buffer)
   buf.ReadFrom(r.Body)
   fmt.Printf("%s\n", buf)
   user := User{}
   if err := json.Unmarshal(body, &user); err != nil {
        SendError(w, http.StatusBadRequest,  err, "JSON unmarshaling failed")
        return
   }

2) body, err := ioutil.ReadAll(r.Body)
   if err != nil {
        SendError(w, http.StatusBadRequest,  err, "Error reading request body",)
        return
   }
   user := User{}
   if err := json.Unmarshal(body, &user); err != nil {
        SendError(w, http.StatusBadRequest,  err, "JSON unmarshaling failed")
        return
   }
```

How to Unmarshal/Marshal HTTP Request JSON Body

```
3) defer r.Body.Close()
   user := User{}
   if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
        SendError(w, http.StatusBadRequest,  err, "JSON unmarshaling failed")
        return
   }
   fmt.Printf("AFTER UNMARSHAL:%#v\n", user)
   ...
```

Example: Simple Users REST/JSON API Endpoint (1)

```
import (
        "encoding/json"
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "strconv"
        "sync/atomic"
)
type User struct {
        Id       int
        Name     string
        Email    string
        Password string
        Active   bool
}
var database = make(map[int]User)
var sequence uint64
```

Example: Simple Users REST/JSON API Endpoint (2)

```
func users(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodPost:
                defer r.Body.Close()
                user := User{}
                if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
                        SendError(w, http.StatusBadRequest,  err, "JSON unmarshaling failed")
                        return
                }
                fmt.Printf("AFTER UNMARSHAL:%#v\n", user)
                newID := int(atomic.AddUint64(&sequence, 1))
                user.Id = newID
                database[newID] = user
                w.Header().Add("Content Type", "application/json")
                w.Header().Add("Location", r.URL.String()+"/"+strconv.Itoa(newID))
                w.WriteHeader(http.StatusCreated)
                data, err := json.MarshalIndent(user, "", "    ")
                if err != nil {
                        SendError(w, http.StatusBadRequest,  err, "JSON marshaling failed")
                        return
                }
                w.Write(data)
```

Example: Simple Users REST/JSON API Endpoint (3)

```
func users(w http.ResponseWriter, r *http.Request) {

                ...

        case http.MethodGet:
                w.Header().Add("Content Type", "application/json")
                users := make([]User, len(database))
                i := 0
                for _, u := range database {
                        users[i] = u
                        i++
                }
                data, err := json.MarshalIndent(users, "", "    ")
                if err != nil {
                        log.Printf("JSON marshaling failed: %s", err)
                }
                w.Write(data)
        }
}
```

Example: Simple Users REST/JSON API Endpoint (4)

```
func SendError(w http.ResponseWriter, status int, err error, message string) {
	var text string
	if err != nil {
		text = fmt.Sprintf("%s: %s", message, err)
	} else {
		text = fmt.Sprintf("%s", message)
	}
	log.Println(text)
	w.WriteHeader(status)
	fmt.Fprintf(w,`{"error": "%s"}`, text)
}


func main() {
	http.HandleFunc("/users", users)
	log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Example: Simple REST Client (1)

```
const APIUrl = "http://localhost:8080/users"
func main() {
	var resp *http.Response
	var err error
	// Post new User
	resp, err = http.Post(APIUrl, "application/json",
		bytes.NewBuffer([]byte(`{"name":"admin", "email":"admin@gmail.com"}`)))
	defer resp.Body.Close()
	if err != nil {
		panic(err)
	}
	PrintResponse(resp)
	// Get all Users
	resp, err = http.Get(APIUrl)
	defer resp.Body.Close()
	if err != nil {
		panic(err)
	}
	PrintResponse(resp)
}
```

Example: Simple REST Client (2)

```
func PrintResponse(resp *http.Response) {
	// Print the HTTP response status.
	fmt.Println("\nResponse status:", resp.Status)
	fmt.Println("Response headers:", resp.Header)

	// Print the the response body.
	scanner := bufio.NewScanner(resp.Body)
	for scanner.Scan() {
		fmt.Println(scanner.Text())
	}

	if err := scanner.Err(); err != nil {
		panic(err)
	}
}
```

Automatically Reloading Web App on Change

- **Fresh** – a command line tool that builds and (re)starts your web application everytime you save a Go or template file:

```
go get github.com/pilu/fresh
fresh
```

- **Gin** – a simple command line utility for live-reloading Go web applications:

```
go get github.com/codegangsta/gin
gin run main.go
```

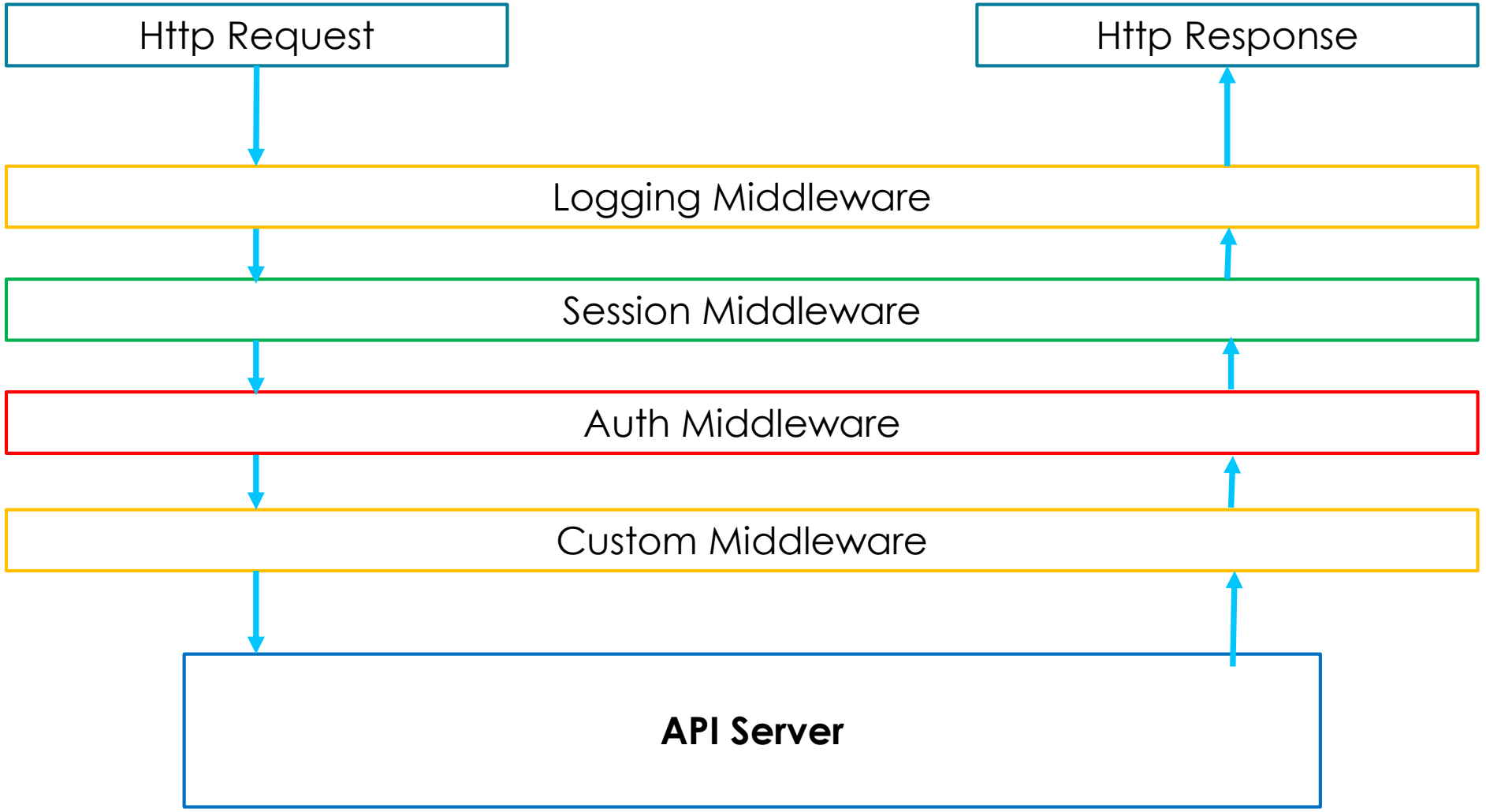
- **Compile Daemon** for Go – Watches your .go files in a directory and invokes go build if a file changed. Nothing more.

```
go get github.com/githubnemo/CompileDaemon
CompileDaemon -command="./MyProgram -my-options"
```

- **nodemon** – simply create a nodemon.json file like:

```
{  "watch": ["*"],
   "ext": "go graphql",
   "ignore": ["*gen*.go"],
   "exec": "go run scripts/gqlgen.go && (killall -9 server || true ) && go run ./server/server.go"
}
```

Pipes and Filters Pattern: Http Middleware



Example: Simple Http Middleware

```
func myMiddleware(handler http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                log.Println("Executing myMmiddleware before request...")
                handler.ServeHTTP(w, r) // pass the call to the handler
                log.Println("Executing myMmiddleware after request...")
        })
}
func myHandlerFunc(w http.ResponseWriter, r *http.Request) {
        log.Println("Executing myHandler...") // Main logic implemented here
        w.Write([]byte("Response: OK"))
}
func main() {
        http.HandleFunc("/users", users)
        // HandlerFunc returns an HTTP Handler using supplied function
        myHandler := http.HandlerFunc(myHandlerFunc)
        http.Handle("/my", myMiddleware(myHandler))
        log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Result:

```
2020/02/02 10:43:30 Executing myMmiddleware before request...
2020/02/02 10:43:30 Executing myHandler...
2020/02/02 10:43:30 Executing myMmiddleware after request...
```

Example: Error Handling Middleware

```
type webError struct {
        Error   error
        Message string
        Code    int
}

type appHandler func(http.ResponseWriter, *http.Request) *webError

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        if err := fn(w, r); err != nil { // err is *appError, not os.Error.
                // or http.Error(w, err.Message, err.Code)
                SendError(w, err.Code, err.Error, "Application error: ")
        }
}
func main() {
        http.Handle("/users", filterPOSTByContentType(setServerTimeHeader(appHandler(users))))
         ...
```

Result:

```
Response status: 400 Bad Request
Response headers: map[Content-Length:[72] Content-Type:[application/json] Date:[Sun, 02 Feb 2020
10:55:15 GMT] Server-Time(UTC):[1580640915]]
{"error": "Application error: : invalid character 'a' after object key"}
```

Http Routers in Go (there are more)

Path	Synopsis
github.com/gorilla/mux 12938 IMPORTS · 11003 STARS	Package mux implements a request router and dispatcher.
github.com/julienschmidt/httprouter 3763 IMPORTS · 10654 STARS	Package httprouter is a trie based high performance HTTP request router.
github.com/docker/docker/api/server/router 3349 IMPORTS · 56209 STARS	
github.com/go-chi/chi 921 IMPORTS · 6903 STARS	Package chi is a small, idiomatic and composable router for building HTTP services.
github.com/tedsuo/rata 416 IMPORTS · 11 STARS	Package rata provides three things: Routes, a Router, and a RequestGenerator.
github.com/gliderlabs/logspout/router 193 IMPORTS · 3431 STARS	generated by go-extpoints -- DO NOT EDIT
github.com/pressly/chi 151 IMPORTS · 6927 STARS	Package chi is a small, idiomatic and composable router for building HTTP services.
github.com/gocraft/web 191 IMPORTS · 1359 STARS	Go Router + Middleware.

References

- R. Fielding, Architectural Styles and the Design of Networkbased Software Architectures, PhD Thesis, University of California, Irvine, 2000
- Fielding's blog discussing REST – http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- Representational state transfer (REST) in Wikipedia – http://en.wikipedia.org/wiki/Representational_state_transfer
- Hypermedia as the Engine of Application State (HATEOAS) in Wikipedia – http://en.wikipedia.org/wiki/HATEOAS
- JavaScript Object Notation (JSON) – http://www.json.org/

Recommended Literature

- The Go Documentation - https://golang.org/doc/
- The Go Bible: Effective Go - https://golang.org/doc/effective_go.html
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD