

ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 07 RESTful Services

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 8, 2025

Outline

Key-Value Store

RESTful Services

RESTful API Design

Reading Assignment

- ▶ This lecture: 5
- ▶ Next lecture: 5

Outline

Key-Value Store

RESTful Services

RESTful API Design

Key-Value Store

- ▶ A simple example demonstrates ideas of cloud storage.
- ▶ Organize data as key-value pairs.
 - ▶ Instead of complex relations as in SQL database.
 - ▶ Each key is unique, with arbitrary value.
 - ▶ What is the difference between this and map and dictionary data structures in programming languages?
- ▶ Core requirement.
 - ▶ Store arbitrary key-value pairs.
 - ▶ Provide service endpoints (API) for put, get, delete.
 - ▶ Persistently store data.
 - ▶ Ensure idempotence.
- ▶ Build a minimum viable product (MVP) first.
 - ▶ Start with absolute minimal functionality.
 - ▶ Then add support for persistent store, security, operation.
 - ▶ Consider scalability, fault resilience, etc.
 - ▶ Serve as start point to learn industrial key-value stores.

Idempotence

- ▶ An operation is idempotent if calling it multiple times has the same effect as calling it once.
 - ▶ A property with origin in algebra.
- ▶ Idempotence is critical for cloud native systems since faults and failures need to be handled properly.
- ▶ For example, consider operations to control a light remotely by sending “toggle” command.
 - ▶ The light will return the current state of on or off.
 - ▶ The light is on and we want to turn it off. However, after sending “toggle”, we fail to receive the returned state.
 - ▶ Is the light on or off? What to do next? Keep sending “toggle” is not wise since that may turn the light on/off multiple times.
- ▶ In other words, this “toggle” operation is not idempotent.

Idempotence (Cont.)

- ▶ Instead of “toggle”, we may redesign our light remote control to use “turn on” and “turn off” commands.
 - ▶ To turn off the light, we keep sending “turn off” until the returned state is received successfully.
 - ▶ The light will be turned off exactly once no matter there are failures or not (assuming someone will repair any failed parts).
 - ▶ “turn on” and “turn off” operations are idempotent!
- ▶ Idempotent operations focus only on end states.
 - ▶ Safer when handling failures and faults.
 - ▶ Often simpler to implement.
 - ▶ More declarative to make communication more effective between developers – tell me “what needs to be done” instead of “how to do it”.
- ▶ So our key-value store only supports put, get, delete.
 - ▶ All of them are idempotent.
 - ▶ Instead of more complicated operations like “update if value equals”, which are not necessarily idempotent.

Generation 0: The Core Functionality

```
var store = make(map[string]string) // global variable
var ErrorNoSuchKey = errors.New("no such key") // sentinel errors

func Put(key string, value string) error {
    store[key] = value
    return nil
}

func Get(key string) (string, error) {
    value, ok := store[key]
    if !ok {
        return "", ErrorNoSuchKey
    }
    return value, nil
}

func Delete(key string) error {
    delete(store, key)
    return nil
}
```

- Need to provide service endpoints so the store can be accessed from different processes and servers and languages.

Outline

Key-Value Store

RESTful Services

RESTful API Design

Generation 1: The Monolith

- ▶ Provide RESTful service endpoints over HTTP protocol.
 - ▶ REpresentational State Transfer (REST) is a software architecture for stateless and layered web services.
 - ▶ Map HTTP methods and paths into functionality.
 - ▶ Accessed from network and supported by most languages.
 - ▶ Simpler than most alternatives like gRPC.
- ▶ Build RESTful web services with Go.
 - ▶ Standard net/http package.
 - ▶ Third party packages like gorilla/mux for enhanced features.

A Minimal RESTful Service

```
package main

import (
    "log"
    "net/http"
    "github.com/gorilla/mux"
)

func helloMuxHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello gorilla/mux!\n"))
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", helloMuxHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

- ▶ Go has the ability to use packages directly from GitHub.
 - ▶ Need to initialize the Go module to download packages and their dependencies (more in Project 2).
 - ▶ Also record the versions for the packages so future package updates won't break existing projects.

Some Background on Networking and HTTP Protocol

- ▶ Services on a host (server) are accessed via a combination of,
 - ▶ IP address of the host, e.g. 50.19.226.237
 - ▶ Type of transport layer protocol, e.g. TCP or UDP.
 - ▶ Port for the specific service, e.g. 8080.
 - ▶ A protocol defining the meaning of the transferred bytes.
- ▶ DNS services translates domain names into IP addresses.
 - ▶ E.g. “www.iit.edu” to 50.19.226.237
 - ▶ May help to achieve scalability by rotating server IP addresses for a single domain name.
- ▶ HTTP is a widely used protocol running on TCP transport.
 - ▶ Supported directly by browsers.
 - ▶ Access specific resource on a server with URL
`protocol://user@host:port/path?query`
 - ▶ port can be omitted for the default: 80 for protocol being http and 443 for protocol being https
 - ▶ path may look like a filesystem path, it may or may not map to an actual file.
 - ▶ Empty user and ?query may be omitted.

Where is our web service?

```
r.HandleFunc("/", helloMuxHandler)
log.Fatal(http.ListenAndServe(":8080", r))
```

- ▶ Support a path of / only.
- ▶ ":8080" indicates to use port 8080.
- ▶ Nothing before : in ":8080" means the service can be accessed from any IP address of the server.
 - ▶ A server can have multiple IP addresses.
 - ▶ Include "localhost", which is 127.0.0.1, widely used for development and testing.
 - ▶ Be careful with firewalls that may block the traffic.

Outline

Key-Value Store

RESTful Services

RESTful API Design

RESTful API Design

- ▶ Use URL path to specify which resource to access.
- ▶ Use HTTP methods to specify operations on the resource.
 - ▶ Usually one of GET, PUT, DELETE, POST.
 - ▶ May use ?query as well.
- ▶ For our key-value store,
 - ▶ Each key/value pair is specified by a path /v1/{key}, e.g. /v1/a refers to the pair with key being “a”.
 - ▶ Note that some part of the book incorrectly states the path to be /v1/key/{key}.
 - ▶ HTTP GET method maps to **Get**, where the value should be returned in the HTTP response body.
 - ▶ HTTP PUT method maps to **Put**, where the value is available from the HTTP request body.
 - ▶ HTTP DELETE method maps to **Delete**.

Implementing GET

```
func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r) // Retrieve "key" from the request
    key := vars["key"]

    value, err := Get(key) // Get value for key
    if errors.Is(err, ErrorNoSuchKey) {
        http.Error(w, err.Error(), http.StatusNotFound)
        return
    }
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Write([]byte(value)) // Write the value to the response
}

func main() {
    ...
    r.HandleFunc("/v1/{key}", keyValueGetHandler).Methods("GET")
}
```

- ▶ Pay attention to how to retrieve {key} from the path.
- ▶ A lot of error handling around `Get` that we have implemented in Generation 0.

Implementing PUT

```
func keyValuePutHandler(w http.ResponseWriter, r *http.Request) {  
    vars := mux.Vars(r) // Retrieve "key" from the request  
    key := vars["key"]  
  
    value, err := io.ReadAll(r.Body) // The request body has our value  
    defer r.Body.Close()  
    if err != nil { // If we have an error, report it  
        http.Error(w, err.Error(), http.StatusInternalServerError)  
        return  
    }  
  
    err = Put(key, string(value)) // Store the value as a string  
    if err != nil { // If we have an error, report it  
        http.Error(w, err.Error(), http.StatusInternalServerError)  
        return  
    }  
  
    w.WriteHeader(http.StatusCreated) // All good! Return StatusCreated  
}  
  
func main() {  
    ...  
    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")  
    ...  
}
```

Concurrency

- ▶ It is possible for RESTful requests to arrive at the same time and most web service frameworks handle them concurrently.
 - ▶ Functions like `keyValuePutHandler` and `keyValueGetHandler` may be called concurrently from multiple threads, which will
 - ▶ Call `Get`, `Put`, `Delete` from multiple threads, which is
 - ▶ Not safe since all of them access `store` that is not thread-safe.
- ▶ Use a mutex (lock) for simplicity.
 - ▶ Use `sync.RWMutex` to enable concurrent read to `store`.
 - ▶ Refer to [ece573-prj02/kvs/core.go](https://github.com/ece573-prj02/kvs/core.go) for details.
 - ▶ Still, only one thread can update `store` at a time.
 - ▶ And there will be a lot of lock contentions if a lot of PUT requests arrive at the same time.

Summary

- ▶ Take steps to design and implement RESTful services.