Datastore

datastore

- nosql database
  - primarily means: no joins across tables
    - generally means it’s less flexible
      - not as powerful in what it can do with queries
      - but if you don’t need that power, it’s a good solution
  - scales well
- our primary database in app engine

datastore

SCHEMALESS

You do not define ahead of time what your “table” looks like;
you don’t have to think about what “fields” will be in each “table;”
you don’t have to, beforehand, think of your entire data structure;
your database can evolve and change on the fly.

- nosql database
  - primarily means: no joins across tables
    - generally means it’s less flexible
      - not as powerful in what it can do with queries
      - but if you don’t need that power, it’s a good solution
  - scales well
- our primary database in app engine

SQL

- Tables
- Records
- Fields

NOSQL

- Kinds
- Entities
- Properties

SQL	Example	NOSQL
Table	Employee	Kind
Record	e1	Entity
Fields	Name Role HireDate Account	Properties

datastore

- table / **kind**
  - like a type
- record / **entity**
  - like a variable of a type
- fields / **properties**
  - like values

Kind: Employee

```
type Employee struct {
    Name     string
    Role     string
    HireDate time.Time
    Account  string
}


func handle(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)

    e1 := Employee{
        Name:     "Joe Citizen",
        Role:     "Manager",
        HireDate: time.Now(),
        Account:  user.Current(c).String(),
    }
```

Entity: e1

Properties:
- Name
- Role
- HireDate
- Account

In Go, Datastore entities are created from `struct` values. The structure's fields become the properties of the entity. To create a new entity, you set up the value you wish to store, create a key, and pass them both to `datastore.Put()`. Updating an existing entity is a matter of performing another `Put()` using the same key. To retrieve an entity from the Datastore, you first set up a value into which the entity will be unmarshalled, then pass a key and a pointer to that value to `datastore.Get()`.

CREATE

1. **set up the value**
   a. **create a value of type struct**
2. **create a key**
3. **use datastore.Put()**
   a. **pass the value & key to datastore.Put()**

RETRIEVE

1. **set up a value**
   a. **create a value of type struct**
2. **use datastore.Get()**
   a. **pass key & value pointer to datastore.Get()**

datastore

- kind
  - like a type
- entity
  - like a variable of a type
- properties
  - like values

Kind: Employee

```
type Employee struct {
    Name     string
    Role     string
    HireDate time.Time
    Account  string
}


func handle(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)

    e1 := Employee{
        Name:     "Joe Citizen",
        Role:     "Manager",
        HireDate: time.Now(),
        Account:  user.Current(c).String(),
    }
```

Entity: e1

You can query based on properties:

eg, give me all of the employees who have the role “manager”

Properties:

Name
Role
HireDate
Account

Objects in the Datastore are known as *entities*. An entity has one or more named *properties*, each of which can have one or more values. Property values can belong to a variety of data types, including integers, floating-point numbers, strings, dates, and binary data, among others. A query on a property with multiple values tests whether any of the values meets the query criteria. This makes such properties useful for membership testing.

> **Note:** Datastore entities are *schemaless*: unlike traditional relational databases, the Datastore does not require that all entities of a given kind have the same properties or that all of an entity's values for a given property be of the same data type. If a formal schema is needed, the application itself is responsible for ensuring that entities conform to it.

keys

Kinds, keys, and identifiers

Each Datastore entity is of a particular *kind*, which categorizes the entity for the purpose of queries; for instance, a human resources application might represent each employee at a company with an entity of kind `Employee`. In addition, each entity has its own *key*, which uniquely identifies it. The key consists of the following components:

- The entity's kind
- An *identifier*, which can be either
  - a *key name* string
  - an integer *ID*
- An optional *ancestor path* locating the entity within the Datastore hierarchy

The identifier is assigned when the entity is created. Because it is part of the entity's key, it is associated permanently with the entity and cannot be changed. It can be assigned in either of two ways:

- Your application can specify its own key name string for the entity.
- You can have the Datastore automatically assign the entity an integer numeric ID.

type Key

```
type Key struct {
    // contains filtered or unexported fields
}
```

Key represents the datastore key for a stored entity, and is immutable.

You're letting app engine provide the unique key

func NewIncompleteKey

```
func NewIncompleteKey(c appengine.Context, kind string, parent *Key) *Key
```

NewIncompleteKey creates a new incomplete key. kind cannot be empty.

func NewKey

```
func NewKey(c appengine.Context, kind, stringID string, intID int64, parent *Key) *Key
```

NewKey creates a new key. kind cannot be empty. Either one or both of stringID and intID must be zero. If both are zero, the key returned is incomplete. parent must either be a complete key or nil.

You're providing the unique key

main.go

```
7
8       "google.golang.org/appengine"
9       "google.golang.org/appengine/datastore"
10      "google.golang.org/appengine/user"
11  )
12
13  type Employee struct {
14      Name     string
15      Role     string
16      HireDate time.Time
17      Account  string
18  }
19
20
21  func handle(w http.ResponseWriter, r *http.Request) {
22      c := appengine.NewContext(r)
23
24      e1 := Employee{
25          Name:     "Joe Citizen",
26          Role:     "Manager",
27          HireDate: time.Now(),
28          Account:  user.Current(c).String(),
29      }
30
31      key, err := datastore.Put(c, datastore.NewIncompleteKey(c, "employee", nil), &e1)
32      if err != nil {
33          http.Error(w, err.Error(), http.StatusInternalServerError)
34          return
35      }
36
37      var e2 Employee
38      if err = datastore.Get(c, key, &e2); err != nil {
39          http.Error(w, err.Error(), http.StatusInternalServerError)
40          return
41      }
42
43      fmt.Fprintf(w, "Stored and retrieved the Employee named %q", e2.Name)
44  }
```

ancestors

Ancestors

- There’s often a relationship between one entity and another
  - for example:
    - a user can have
      - post
      - post
      - post
      - … posts …
    - the **ancestor** of a post is the user
      - users have posts
        - users have descendents
        - one of their descendents might be posts
    - could photos be a descendent of users
      - Yes - users could upload photos

Ancestor Queries

- We can query based upon ancestor
  - eg, give me all of the photos that have this ancestor
    - what I’m saying is, “give me all of the photos that belong to this user”

Ancestor paths

Entities in the Datastore form a hierarchically structured space similar to the directory structure of a file system. When you create an entity, you can optionally designate another entity as its *parent;* the new entity is a *child* of the parent entity (note that unlike in a file system, the parent entity need not actually exist). An entity without a parent is a *root entity.* The association between an entity and its parent is permanent, and cannot be changed once the entity is created. The Datastore will never assign the same numeric ID to two entities with the same parent, or to two root entities (those without a parent).

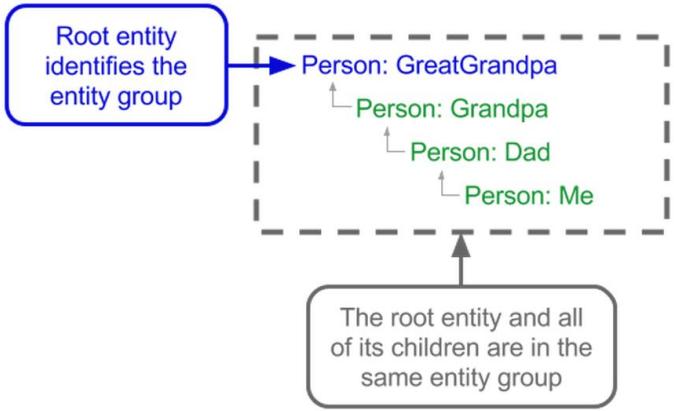
An entity's parent, parent's parent, and so on recursively, are its *ancestors;* its children, children's children, and so on, are its *descendants.* A root entity and all of its descendants belong to the same *entity group.* The sequence of entities beginning with a root entity and proceeding from parent to child, leading to a given entity, constitute that entity's *ancestor path.* The complete key identifying the entity consists of a sequence of kind-identifier pairs specifying its ancestor path and terminating with those of the entity itself:

```
[Person:GreatGrandpa, Person:Grandpa, Person:Dad, Person:Me]
```

For a root entity, the ancestor path is empty and the key consists solely of the entity's own kind and identifier:

```
[Person:GreatGrandpa]
```

This concept is illustrated by the following diagram:



getting record(s)

Getting Records

- Key
  - pass key & value pointer to datastore.Get()
- Field query
  - give me all of the employees who have the role “manager”
- Ancestor query
  - give me all of the photos that have this particular ancestor

Getting Records

- Key
  - pass key & value pointer to datastore.Get()
- Field query
  - give me all of the employees who have the role “manager”
- Ancestor query
  - give me all of the photos that have this particular ancestor

The query operates on entities of a given kind

Getting Records

The fields (properties) are indexed

- Key
  - pass key & value pointer to datastore.Get()
- Field query
  - give me all of the employees who have the role “manager”
- Ancestor query
  - give me all of the photos that have this particular ancestor

The query operates on entities of a given kind

Queries and indexes

In addition to retrieving entities from the Datastore directly by their keys, an application can perform a *query* to retrieve them by the values of their properties. The query operates on entities of a given *kind*; it can specify *filters* on the entities' property values, keys, and ancestors, and can return zero or more entities as *results*. A query can also specify *sort orders* to sequence the results by their property values. The results include all entities that have at least one value for every property named in the filters and sort orders, and whose property values meet all the specified filter criteria. The query can return entire entities, projected entities, or just entity keys.

Queries and indexes

In addition to retrieving entities from the Datastore directly by their keys, an application can perform a *query* to retrieve them by the values of their properties. The query operates on entities of a given *kind*; it can specify *filters* on the entities' property values, keys, and ancestors, and can return zero or more entities as *results*. A query can also specify *sort orders* to sequence the results by their property values. The results include all entities that have at least one value for every property named in the filters and sort orders, and whose property values meet all the specified filter criteria. The query can return entire entities, projected entities, or just entity keys.

A typical query includes the following:

- An *entity kind* to which the query applies
- Zero or more *filters* based on the entities' property values, keys, and ancestors
- Zero or more *sort orders* to sequence the results

A typical query includes the following:

- An *entity kind* to which the query applies
- Zero or more *filters* based on the entities' property values, keys, and ancestors
- Zero or more *sort orders* to sequence the results

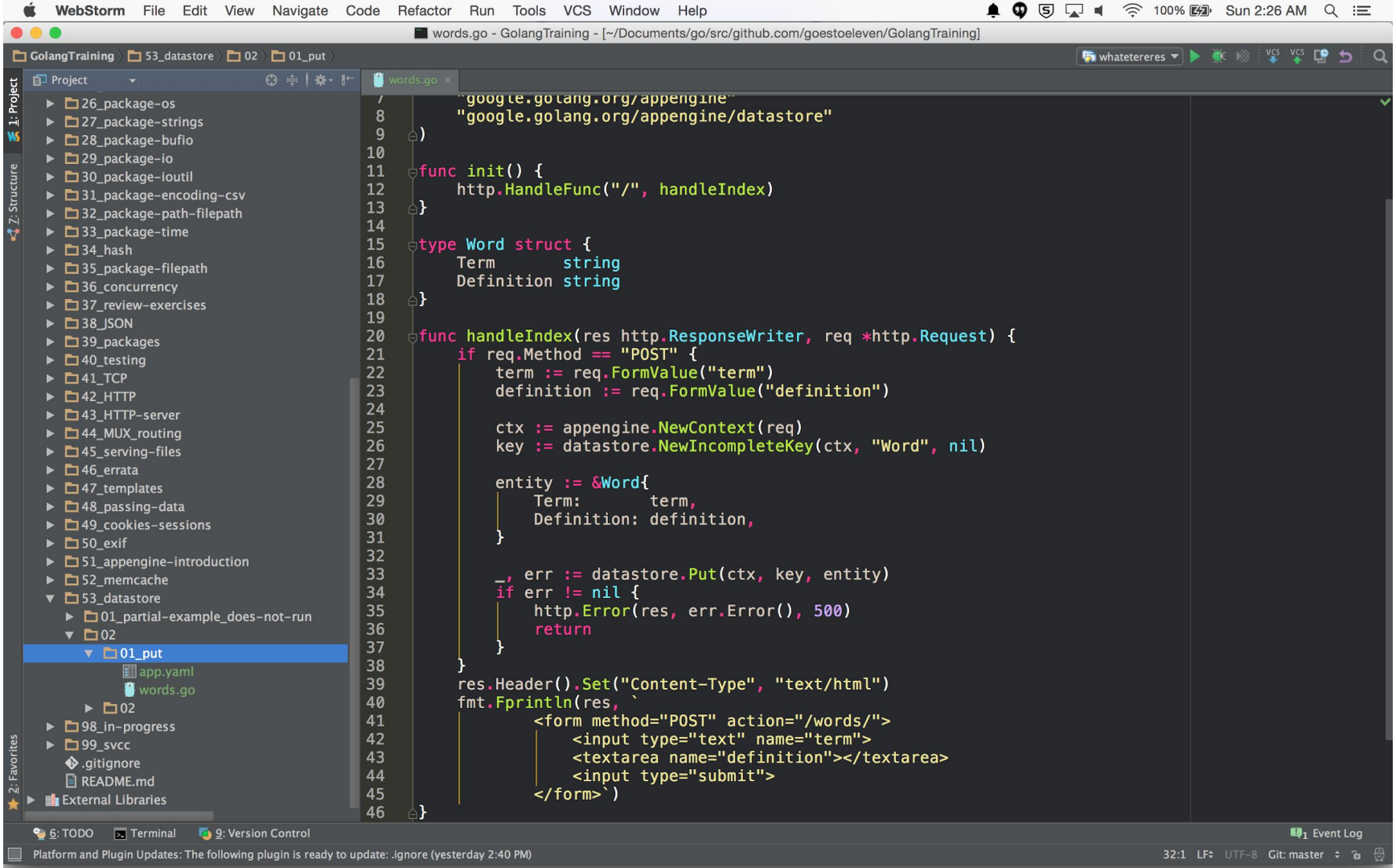
When executed, the query retrieves all entities of the given kind that satisfy all of the given filters, sorted in the specified order.

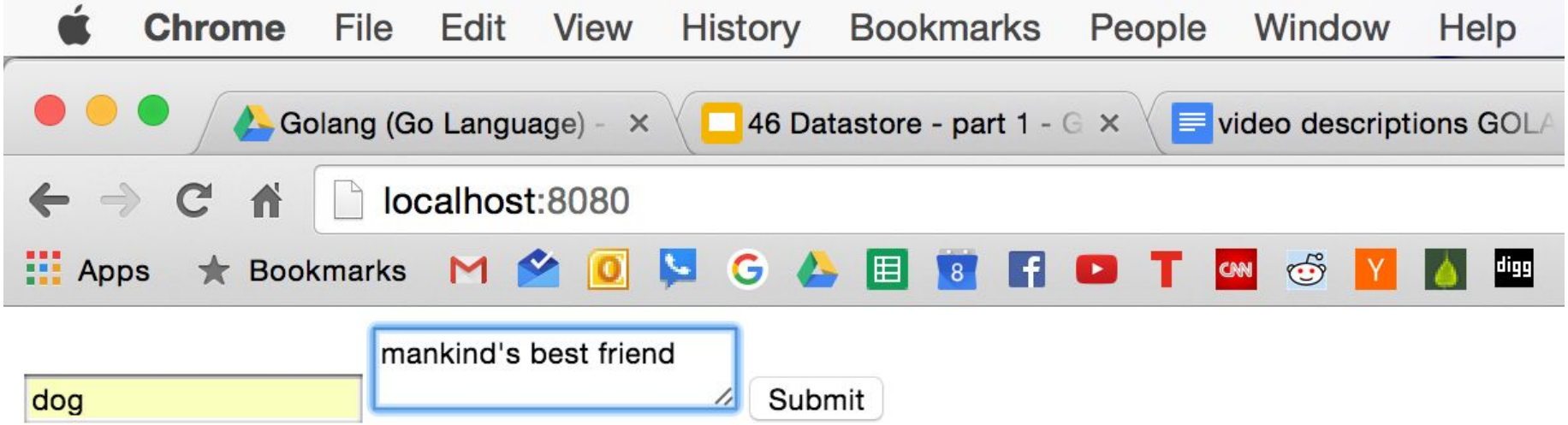
> **Note:** To conserve memory and improve performance, a query should, whenever possible, specify a limit on the number of results returned.

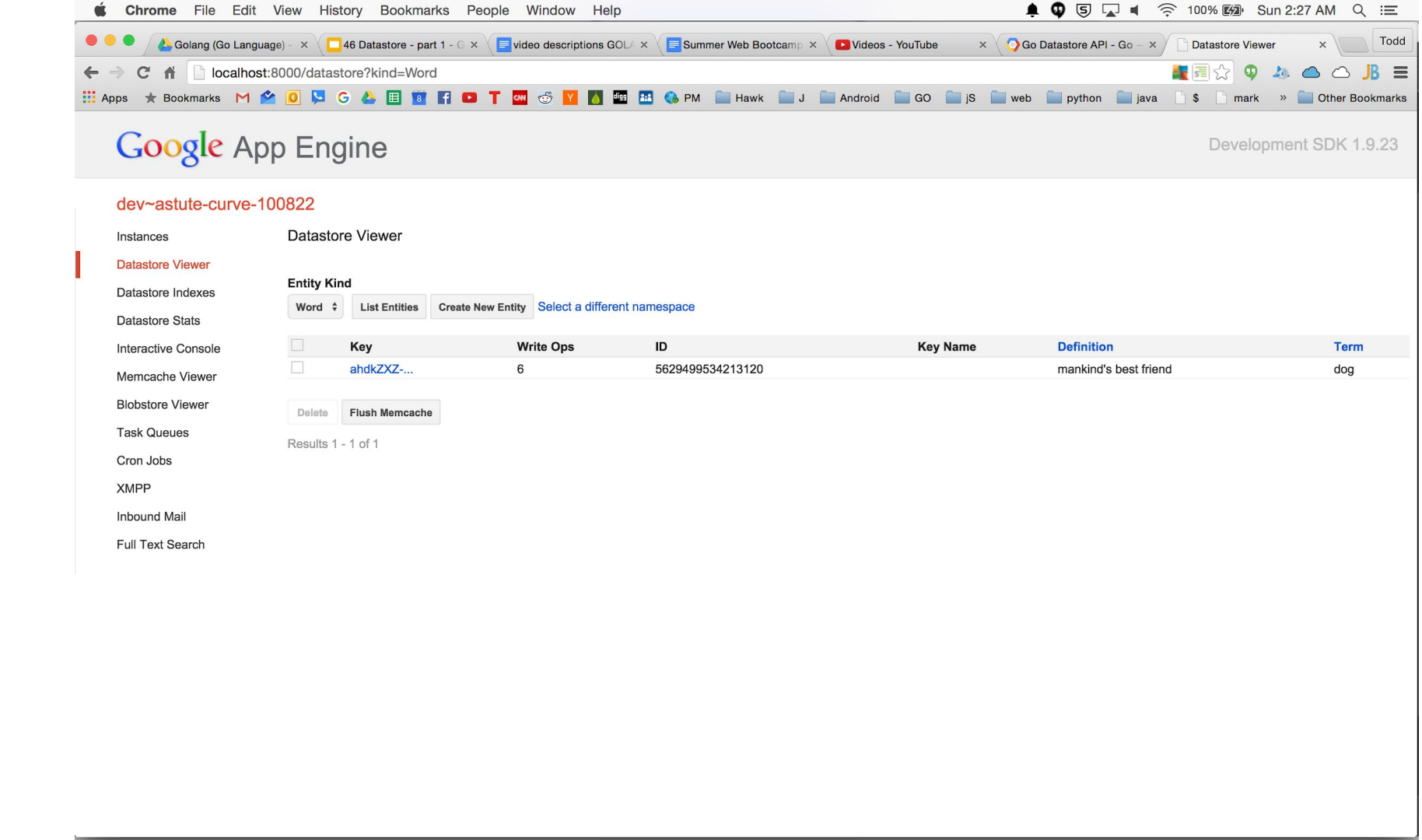
A query can also include an *ancestor filter* limiting the results to just the entity group descended from a specified ancestor. Such a query is known as an *ancestor query*. By default, ancestor queries return *strongly consistent* results, which are guaranteed to be up to date with the latest changes to the data. Non-ancestor queries, by contrast, can span the entire Datastore rather than just a single entity group, but are only *eventually consistent* and may return stale results. If strong consistency is important to your application, you may need to take this into account when structuring your data, placing related entities in the same entity group so they can be retrieved with an ancestor rather than a non-ancestor query for more information.

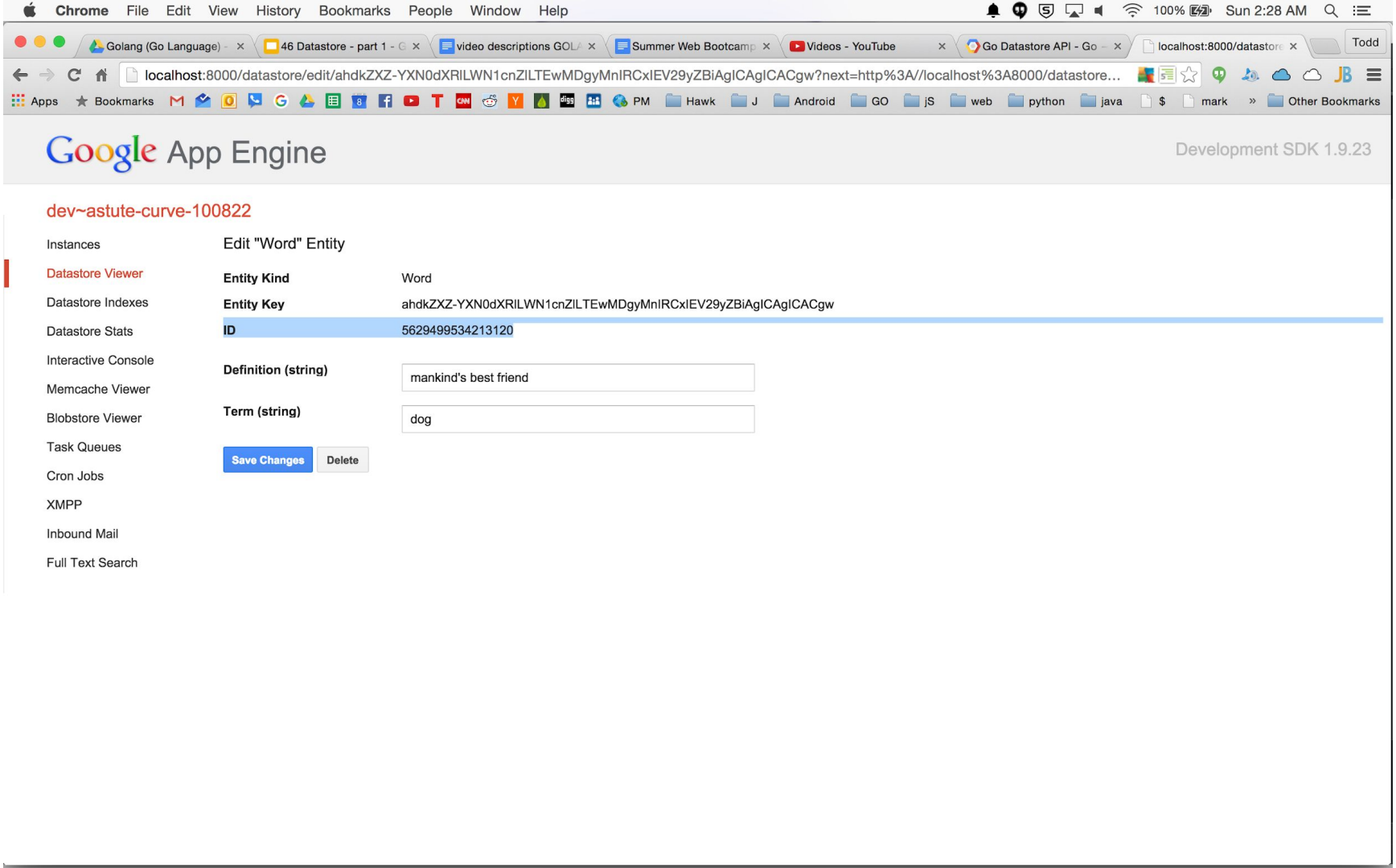
believe the bird, not the book

put

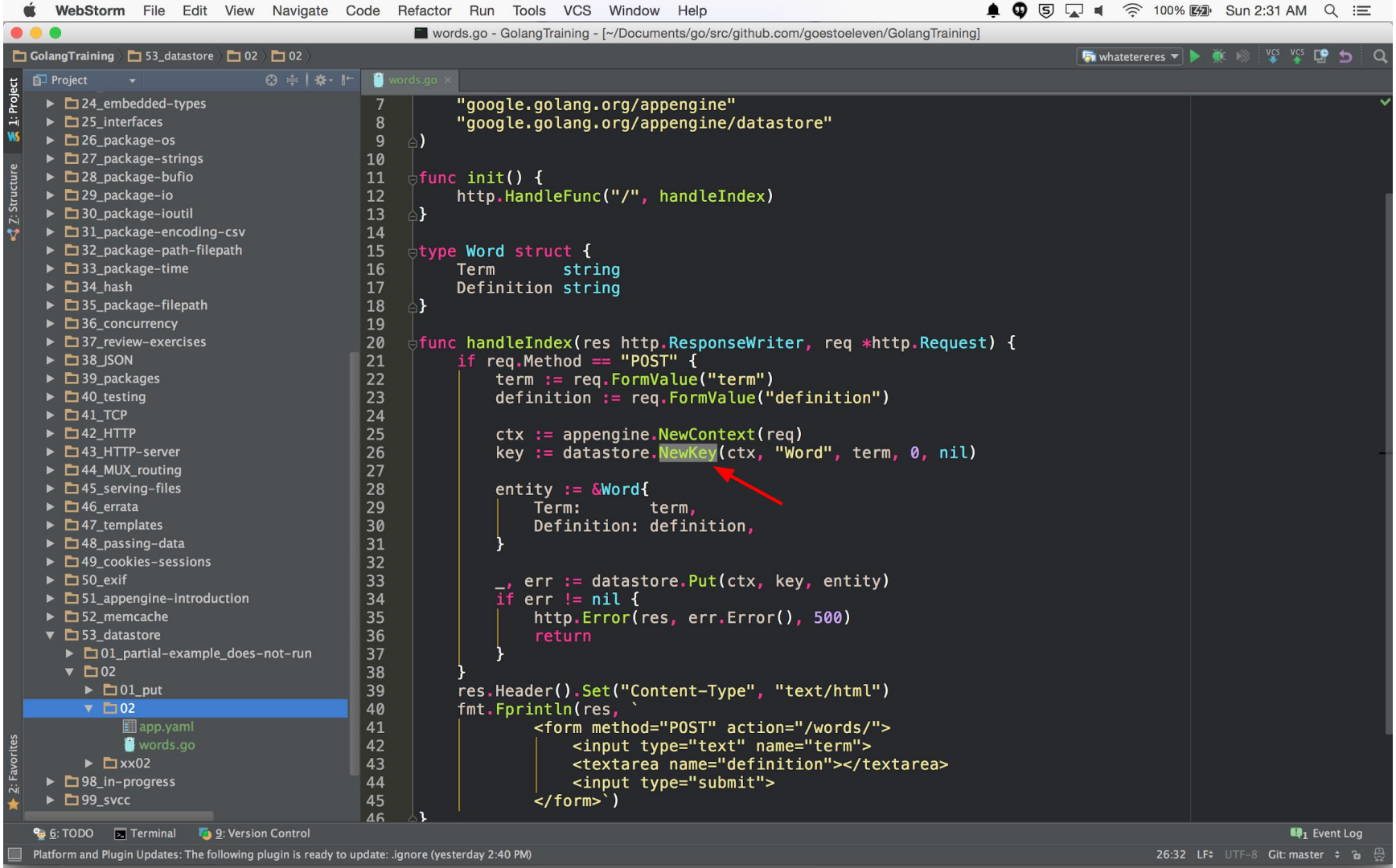

WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
100% Sun 2:26 AM
words.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 53_datastore 02 01_put
whatetereres
Project
words.go
26_package-os
27_package-strings
28_package-bufio
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
42_HTTP
43_HTTP-server
44_MUX_routing
45_serving-files
46_errata
47_templates
48_passing-data
49_cookies-sessions
50_exif
51_appengine-introduction
52_memcache
53_datastore
01_partial-example_does-not-run
02
01_put
app.yaml
words.go
02
98_in-progress
99_svcc
.gitignore
README.md
External Libraries
"google.golang.org/appengine"
"google.golang.org/appengine/datastore"
)
func init() {
http.HandleFunc("/", handleIndex)
}
type Word struct {
Term string
Definition string
}
func handleIndex(res http.ResponseWriter, req *http.Request) {
if req.Method == "POST" {
term := req.FormValue("term")
definition := req.FormValue("definition")
ctx := appengine.NewContext(req)
key := datastore.NewIncompleteKey(ctx, "Word", nil)
entity := &Word{
Term: term,
Definition: definition,
}
_, err := datastore.Put(ctx, key, entity)
if err != nil {
http.Error(res, err.Error(), 500)
return
}
}
res.Header().Set("Content-Type", "text/html")
fmt.Fprintln(res, `
<form method="POST" action="/words/">
<input type="text" name="term">
<textarea name="definition"></textarea>
<input type="submit">
</form>`)
}
6: TODO
Terminal
9: Version Control
Event Log
Platform and Plugin Updates: The following plugin is ready to update: .ignore (yesterday 2:40 PM)
32:1 LF UTF-8 Git: master

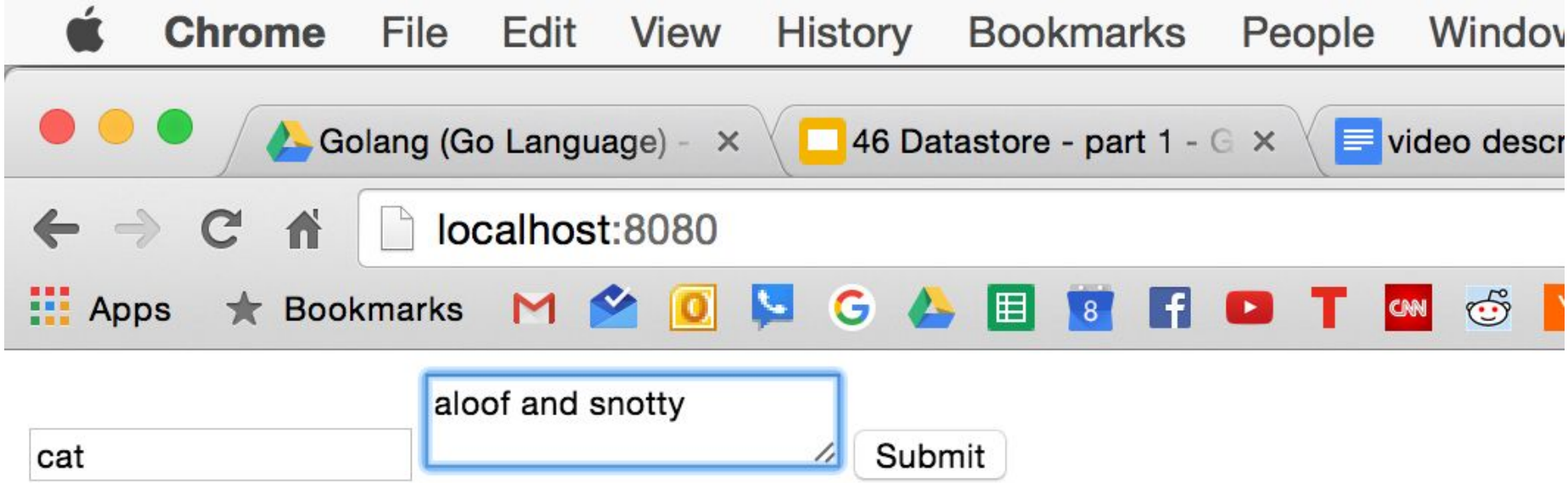

Chrome File Edit View History Bookmarks People Window Help
Golang (Go Language) -
46 Datastore - part 1 - G
video descriptions GOLA
localhost:8080
Apps
Bookmarks
mankind's best friend
dog
Submit

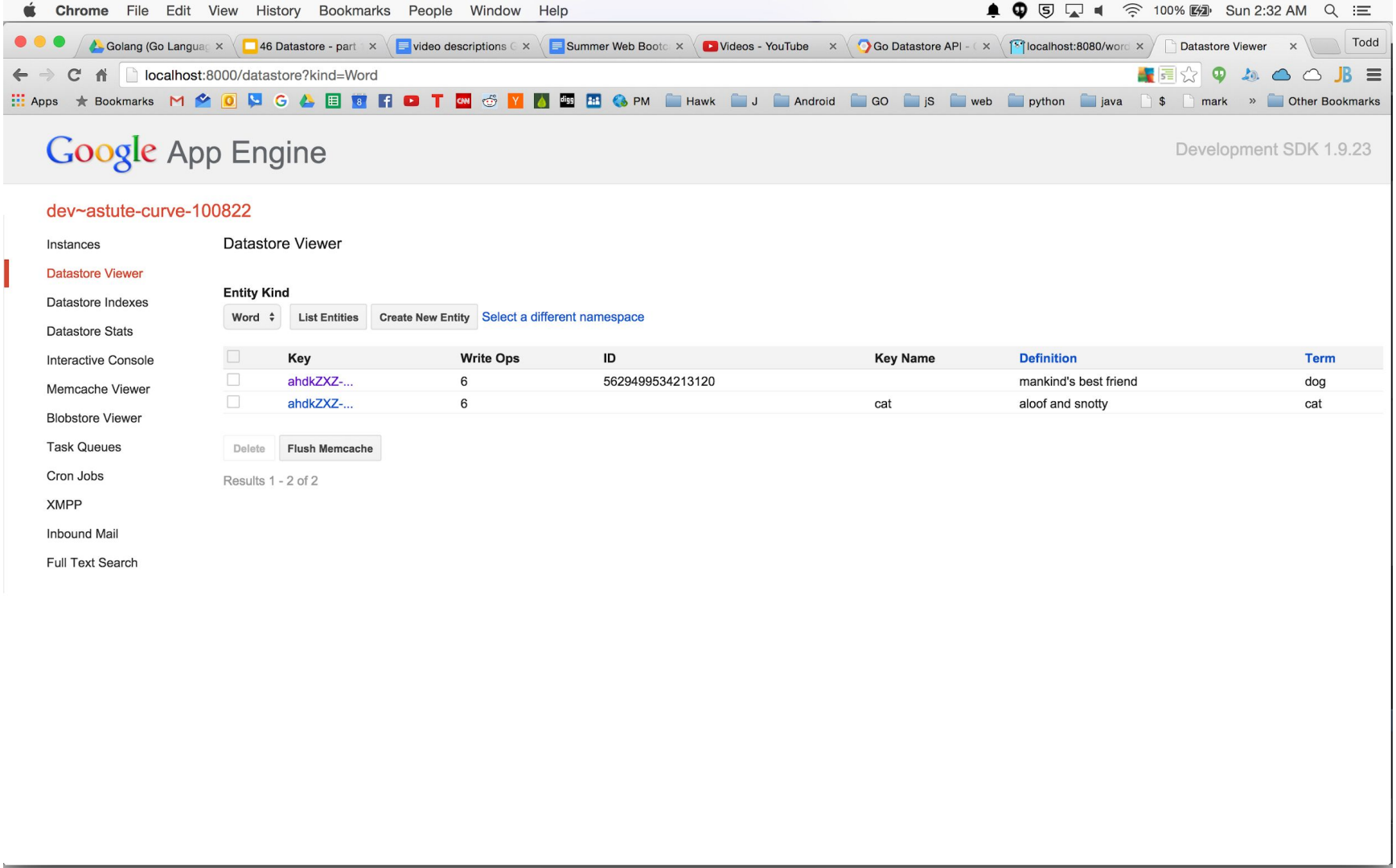

Chrome File Edit View History Bookmarks People Window Help
100% Sun 2:27 AM
Golang (Go Language) -
46 Datastore - part 1 - G
video descriptions GOL
Summer Web Bootcamp
Videos - YouTube
Go Datastore API - Go -
Datastore Viewer
Todd
localhost:8000/datastore?kind=Word
Apps Bookmarks PM Hawk J Android GO jS web python java $ mark Other Bookmarks
Google App Engine
Development SDK 1.9.23
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Datastore Viewer
Entity Kind
Word
List Entities
Create New Entity
Select a different namespace
Key
Write Ops
ID
Key Name
Definition
Term
ahdkZXZ-...
6
5629499534213120
mankind's best friend
dog
Delete
Flush Memcache
Results 1 - 1 of 1

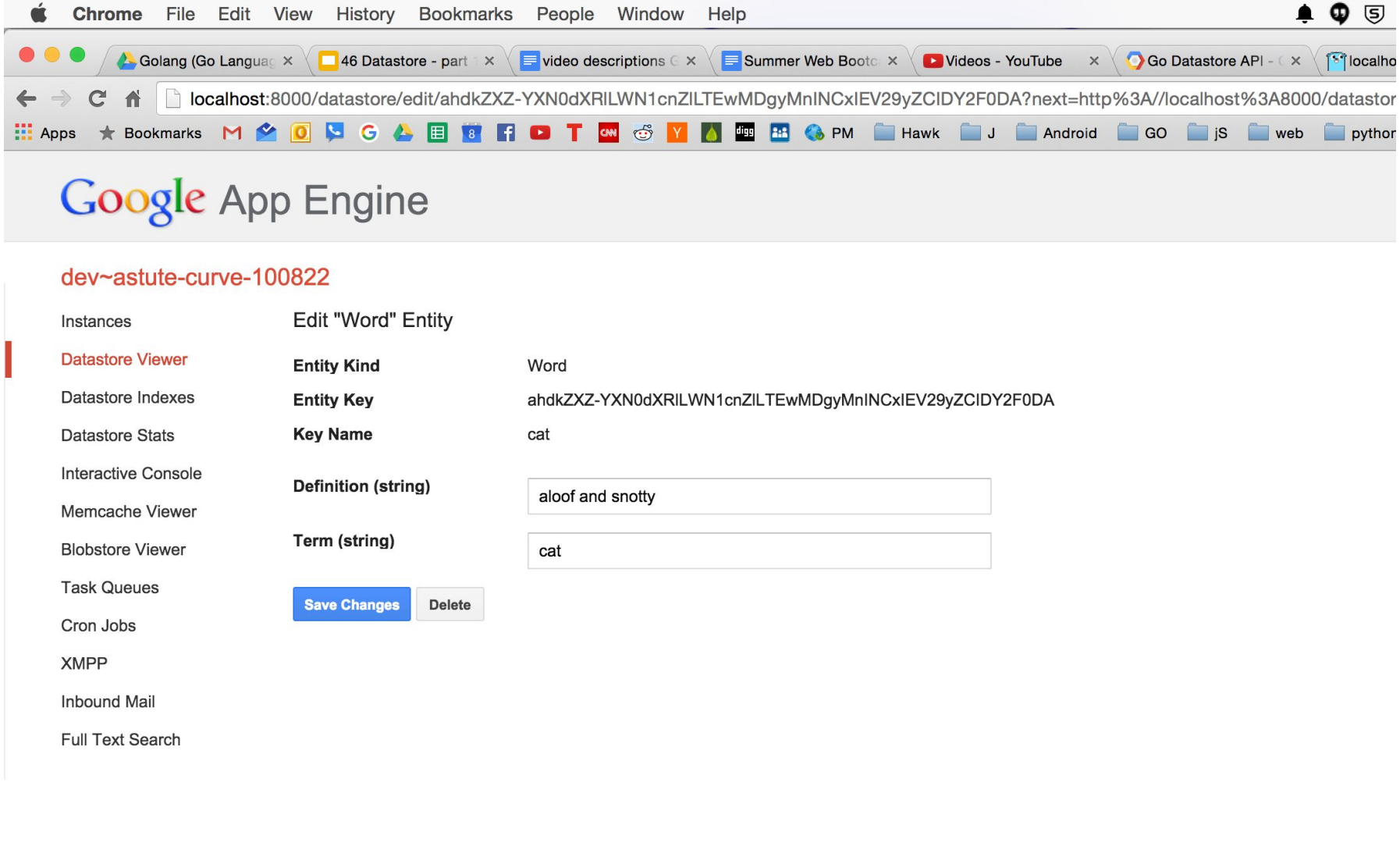

Chrome File Edit View History Bookmarks People Window Help
100% Sun 2:28 AM
Golang (Go Language) -
46 Datastore - part 1 - G
video descriptions GOL
Summer Web Bootcamp
Videos - YouTube
Go Datastore API - Go -
localhost:8000/datastore
Todd
localhost:8000/datastore/edit/ahdkZXZ-YXN0dXRlLWN1cnZlLTEwMDgyMnIRCxIEV29yZBiAgICAgICACgw?next=http%3A//localhost%3A8000/datastore...
Apps
Bookmarks
PM
Hawk
J
Android
GO
jS
web
python
java
$
mark
Other Bookmarks
Google App Engine
Development SDK 1.9.23
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Edit "Word" Entity
Entity Kind
Word
Entity Key
ahdkZXZ-YXN0dXRlLWN1cnZlLTEwMDgyMnIRCxIEV29yZBiAgICAgICACgw
ID
5629499534213120
Definition (string)
mankind's best friend
Term (string)
dog
Save Changes
Delete

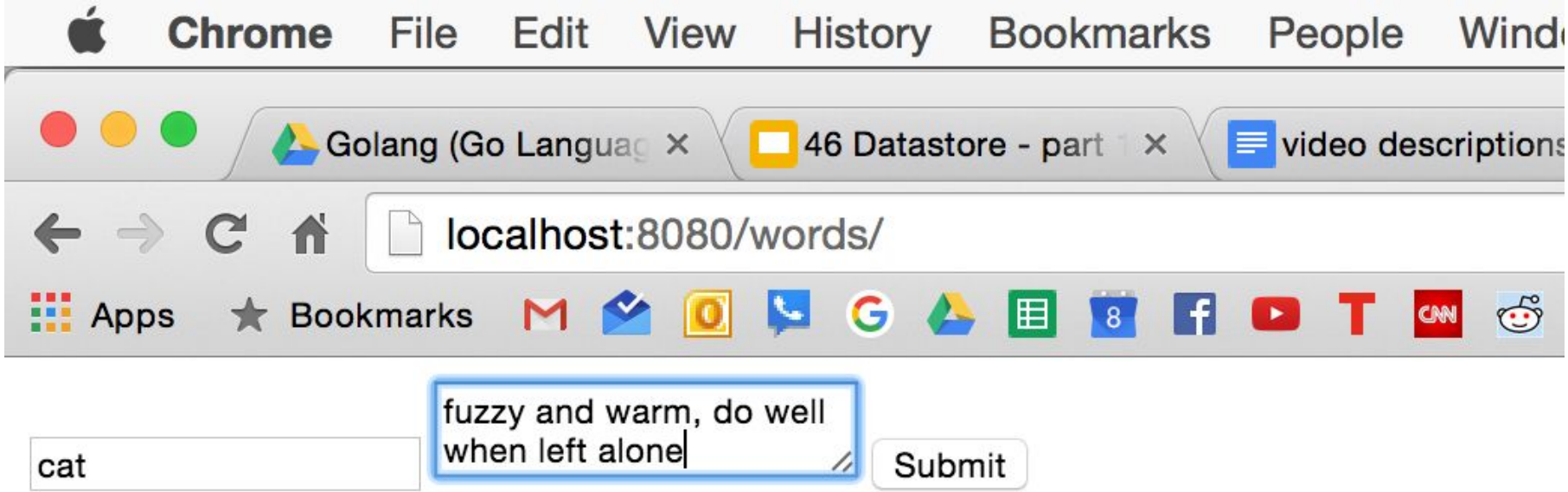
put

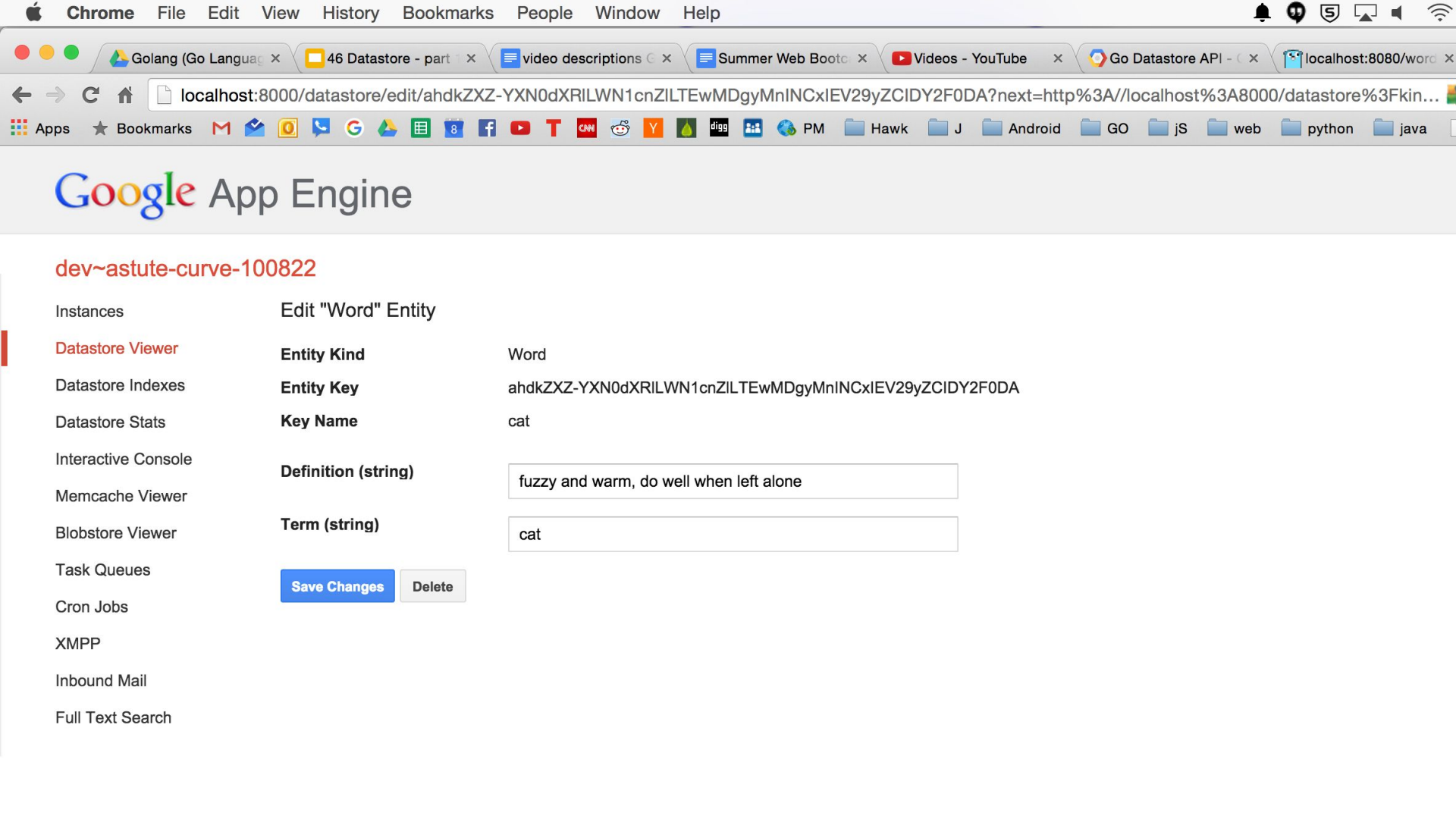

WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
Sun 2:31 AM
words.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 53_datastore 02 02
whatetereres
Project
words.go
24_embedded-types
25_interfaces
26_package-os
27_package-strings
28_package-bufio
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
42_HTTP
43_HTTP-server
44_MUX_routing
45_serving-files
46_errata
47_templates
48_passing-data
49_cookies-sessions
50_exif
51_appengine-introduction
52_memcache
53_datastore
01_partial-example_does-not-run
02
01_put
02
app.yaml
words.go
xx02
98_in-progress
99_svcc
"google.golang.org/appengine"
"google.golang.org/appengine/datastore"
)
func init() {
http.HandleFunc("/", handleIndex)
}
type Word struct {
Term string
Definition string
}
func handleIndex(res http.ResponseWriter, req *http.Request) {
if req.Method == "POST" {
term := req.FormValue("term")
definition := req.FormValue("definition")
ctx := appengine.NewContext(req)
key := datastore.NewKey(ctx, "Word", term, 0, nil)
entity := &Word{
Term: term,
Definition: definition,
}
_, err := datastore.Put(ctx, key, entity)
if err != nil {
http.Error(res, err.Error(), 500)
return
}
}
res.Header().Set("Content-Type", "text/html")
fmt.Fprintln(res, `
<form method="POST" action="/words/">
<input type="text" name="term">
<textarea name="definition"></textarea>
<input type="submit">
</form>`)
}
6: TODO
Terminal
9: Version Control
Event Log
Platform and Plugin Updates: The following plugin is ready to update: .ignore (yesterday 2:40 PM)
26:32 LF UTF-8 Git: master
1: Project
2: Favorites
Z: Structure

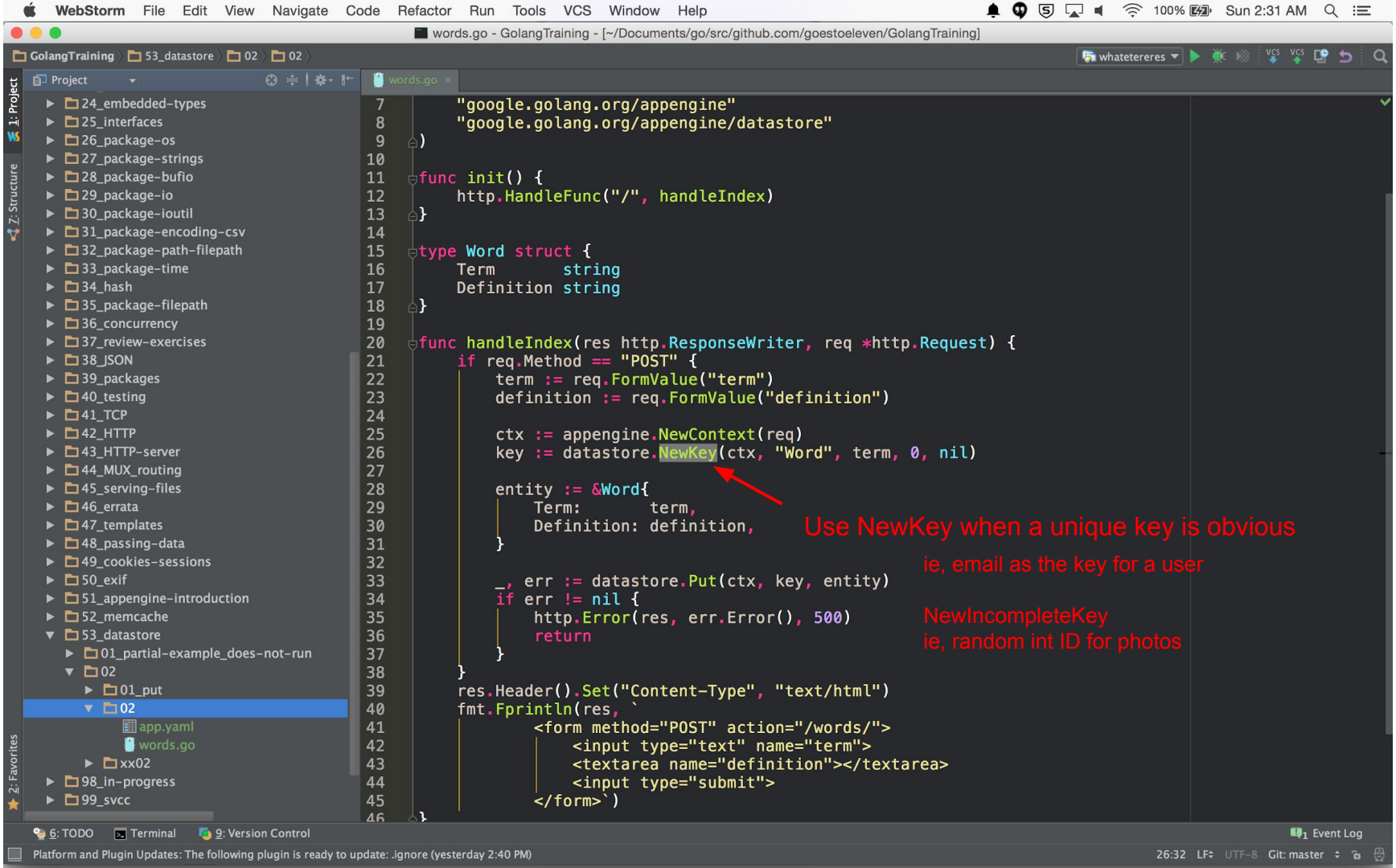

Chrome File Edit View History Bookmarks People Windov
Golang (Go Language) -
46 Datastore - part 1 - G
video descr
localhost:8080
Apps
Bookmarks
aloof and snotty
cat
Submit


Chrome File Edit View History Bookmarks People Window Help
100% Sun 2:32 AM
Golang (Go Langua
46 Datastore - part
video descriptions
Summer Web Bootc
Videos - YouTube
Go Datastore API -
localhost:8080/word
Datastore Viewer
Todd
localhost:8000/datastore?kind=Word
Apps Bookmarks PM Hawk J Android GO jS web python java $ mark Other Bookmarks
Google App Engine
Development SDK 1.9.23
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Datastore Viewer
Entity Kind
Word
List Entities
Create New Entity
Select a different namespace
Key
Write Ops
ID
Key Name
Definition
Term
ahdkZXZ-...
6
5629499534213120
mankind's best friend
dog
ahdkZXZ-...
6
cat
aloof and snotty
cat
Delete
Flush Memcache
Results 1 - 2 of 2


Chrome File Edit View History Bookmarks People Window Help
Golang (Go Languag
46 Datastore - part
video descriptions G
Summer Web Bootc
Videos - YouTube
Go Datastore API - G
localho
localhost:8000/datastore/edit/ahdkZXZ-YXN0dXRlLWN1cnZlLTEwMDgyMnINCxIEV29yZCIDY2F0DA?next=http%3A//localhost%3A8000/datastor
Apps
Bookmarks
PM
Hawk
J
Android
GO
jS
web
pythor
Google App Engine
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Edit "Word" Entity
Entity Kind
Word
Entity Key
ahdkZXZ-YXN0dXRlLWN1cnZlLTEwMDgyMnINCxIEV29yZCIDY2F0DA
Key Name
cat
Definition (string)
aloof and snotty
Term (string)
cat
Save Changes
Delete


Chrome File Edit View History Bookmarks People
Golang (Go Languag
46 Datastore - part
video descriptions
localhost:8080/words/
Apps
Bookmarks
cat
fuzzy and warm, do well when left alone
Submit


Chrome File Edit View History Bookmarks People Window Help
Golang (Go Languag
46 Datastore - part
video descriptions
Summer Web Bootc
Videos - YouTube
Go Datastore API -
localhost:8080/word
localhost:8000/datastore/edit/ahdkZXZ-YXN0dXRlLWN1cnZlLTEwMDgyMnINCxIEV29yZCIDY2F0DA?next=http%3A//localhost%3A8000/datastore%3Fkin...
Apps
Bookmarks
PM
Hawk
J
Android
GO
jS
web
python
java
Google App Engine
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Edit "Word" Entity
Entity Kind
Word
Entity Key
ahdkZXZ-YXN0dXRlLWN1cnZlLTEwMDgyMnINCxIEV29yZCIDY2F0DA
Key Name
cat
Definition (string)
fuzzy and warm, do well when left alone
Term (string)
cat
Save Changes
Delete


WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
Sun 2:31 AM
words.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 53_datastore 02 02
whatetereres
Project
words.go
24_embedded-types
25_interfaces
26_package-os
27_package-strings
28_package-bufio
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
42_HTTP
43_HTTP-server
44_MUX_routing
45_serving-files
46_errata
47_templates
48_passing-data
49_cookies-sessions
50_exif
51_appengine-introduction
52_memcache
53_datastore
01_partial-example_does-not-run
02
01_put
02
app.yaml
words.go
xx02
98_in-progress
99_svcc
"google.golang.org/appengine"
"google.golang.org/appengine/datastore"
)
func init() {
http.HandleFunc("/", handleIndex)
}
type Word struct {
Term string
Definition string
}
func handleIndex(res http.ResponseWriter, req *http.Request) {
if req.Method == "POST" {
term := req.FormValue("term")
definition := req.FormValue("definition")
ctx := appengine.NewContext(req)
key := datastore.NewKey(ctx, "Word", term, 0, nil)
entity := &Word{
Term: term,
Definition: definition,
}
_, err := datastore.Put(ctx, key, entity)
if err != nil {
http.Error(res, err.Error(), 500)
return
}
}
res.Header().Set("Content-Type", "text/html")
fmt.Fprintln(res, `
<form method="POST" action="/words/">
<input type="text" name="term">
<textarea name="definition"></textarea>
<input type="submit">
</form>`)
}
Use NewKey when a unique key is obvious
ie, email as the key for a user
NewIncompleteKey
ie, random int ID for photos
6: TODO
Terminal
9: Version Control
Event Log
Platform and Plugin Updates: The following plugin is ready to update: .ignore (yesterday 2:40 PM)
26:32 LF UTF-8 Git: master

getting records

Getting Records

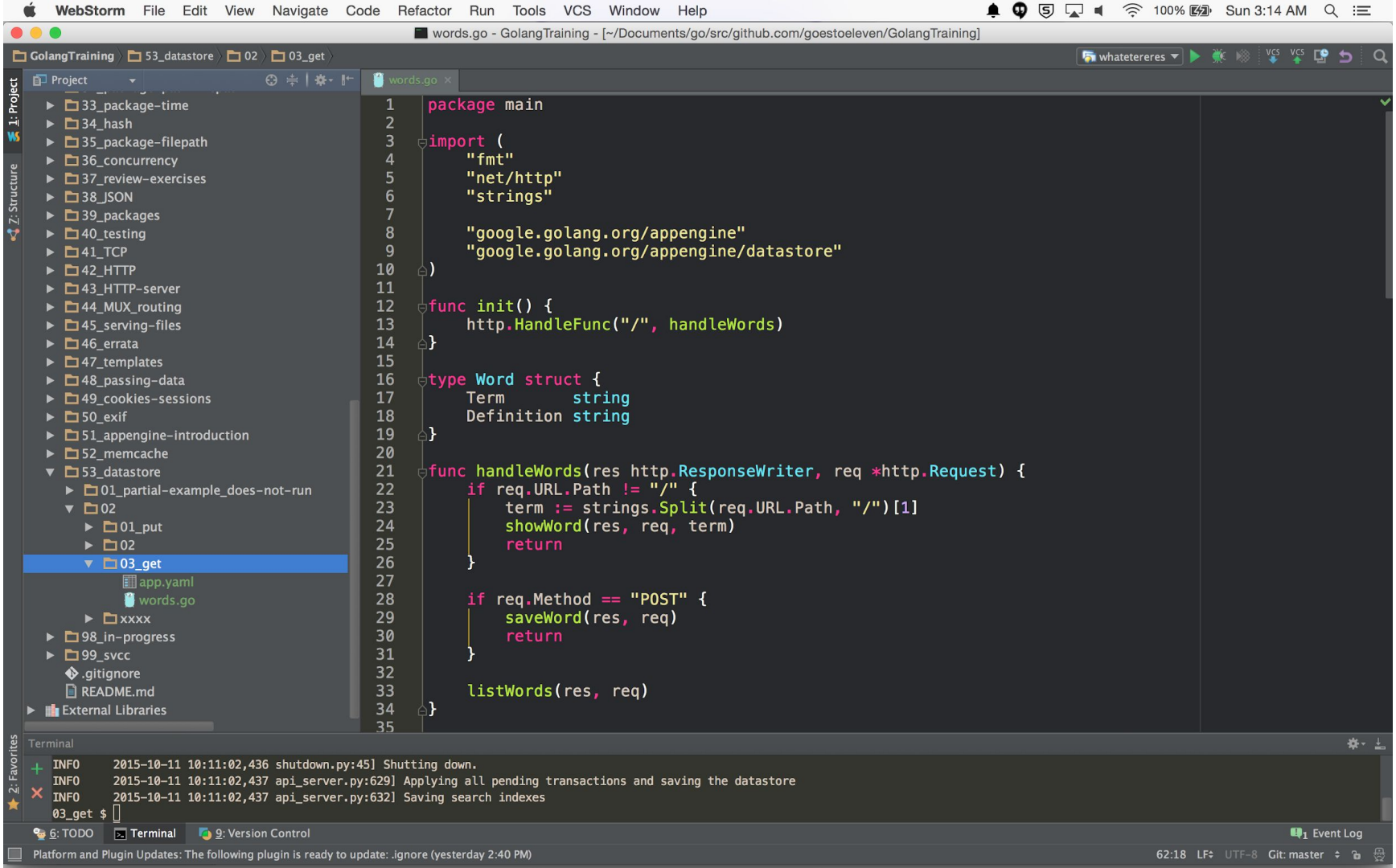
The fields (properties) are indexed

- Key
  - pass key & value pointer to datastore.Get()
- Field query
  - give me all of the employees who have the role “manager”
- Ancestor query
  - give me all of the photos that have this particular ancestor

The query operates on entities of a given kind

get

key


WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
100% Sun 3:14 AM
words.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 53_datastore 02 03_get
whatetereres
Project
words.go
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
42_HTTP
43_HTTP-server
44_MUX_routing
45_serving-files
46_errata
47_templates
48_passing-data
49_cookies-sessions
50_exif
51_appengine-introduction
52_memcache
53_datastore
01_partial-example_does-not-run
02
01_put
02
03_get
app.yaml
words.go
xxxx
98_in-progress
99_svcc
.gitignore
README.md
External Libraries
1 package main
2
3 import (
4 "fmt"
5 "net/http"
6 "strings"
7
8 "google.golang.org/appengine"
9 "google.golang.org/appengine/datastore"
10)
11
12 func init() {
13 http.HandleFunc("/", handleWords)
14 }
15
16 type Word struct {
17 Term string
18 Definition string
19 }
20
21 func handleWords(res http.ResponseWriter, req *http.Request) {
22 if req.URL.Path != "/" {
23 term := strings.Split(req.URL.Path, "/")[1]
24 showWord(res, req, term)
25 return
26 }
27
28 if req.Method == "POST" {
29 saveWord(res, req)
30 return
31 }
32
33 listWords(res, req)
34 }
35
Terminal
INFO 2015-10-11 10:11:02,436 shutdown.py:45] Shutting down.
INFO 2015-10-11 10:11:02,437 api_server.py:629] Applying all pending transactions and saving the datastore
INFO 2015-10-11 10:11:02,437 api_server.py:632] Saving search indexes
03_get $
6: TODO
Terminal
9: Version Control
Event Log
Platform and Plugin Updates: The following plugin is ready to update: .ignore (yesterday 2:40 PM)
62:18 LF UTF-8 Git: master
1: Project
2: Favorites
Z: Structure

```
func showWord(res http.ResponseWriter, req *http.Request, term string) {
    ctx := appengine.NewContext(req)
    key := datastore.NewKey(ctx, "Word", term, 0, nil)
    var entity Word
    err := datastore.Get(ctx, key, &entity)
    if err == datastore.ErrNoSuchEntity {
        http.NotFound(res, req)
        return
    } else if err != nil {
        http.Error(res, err.Error(), 500)
        return
    }
    res.Header().Set("Content-Type", "text/html")
    fmt.Fprintln(res, `
        <dl>
            <dt>`+entity.Term+`</dt>
            <dd>`+entity.Definition+`</dd>
        </dl>
    `)
}
```

NewQuery

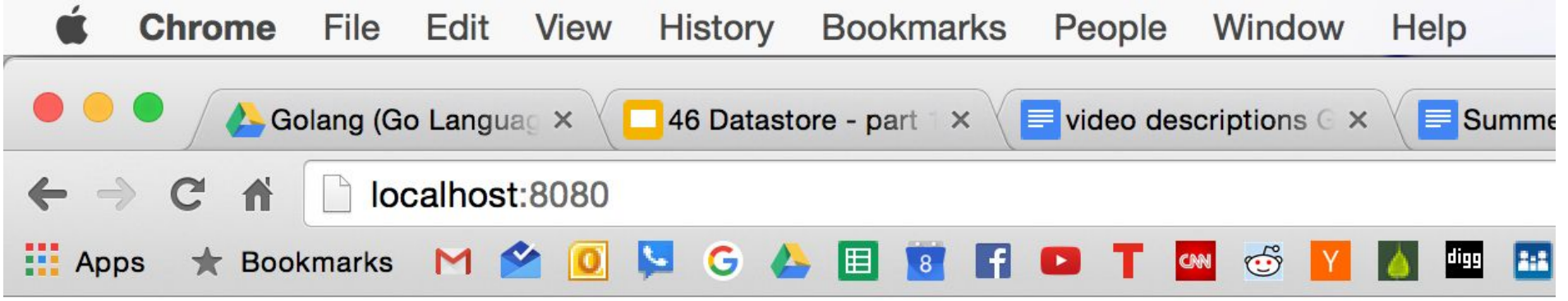
kind query

```
func listWords(res http.ResponseWriter, req *http.Request) {
    ctx := appengine.NewContext(req)
    q := datastore.NewQuery("Word").Order("Term")

    html := ""

    iterator := q.Run(ctx)
    for {
        var entity Word
        _, err := iterator.Next(&entity)
        if err == datastore.Done {
            break
        } else if err != nil {
            http.Error(res, err.Error(), 500)
            return
        }
        html += `
            <dt>` + entity.Term + `</dt>
            <dd>` + entity.Definition + `</dd>
        `
    }

    res.Header().Set("Content-Type", "text/html")
    fmt.Fprintln(res, `
            <dl>
                `+html+`
            </dl>
            <form method="POST">
                <input type="text" name="term">
                <textarea name="definition"></textarea>
                <input type="submit">
            </form>
            `)
}
```

put

(review)

```go
func saveWord(res http.ResponseWriter, req *http.Request) {
    term := req.FormValue("term")
    definition := req.FormValue("definition")
    ctx := appengine.NewContext(req)
    key := datastore.NewKey(ctx, "Word", term, 0, nil)
    entity := &Word{
        Term:       term,
        Definition: definition,
    }

    _, err := datastore.Put(ctx, key, entity)
    if err != nil {
        http.Error(res, err.Error(), 500)
        return
    }

    http.Redirect(res, req, "/", 302)
}
```



cat

fuzzy and warm, do well when left alone

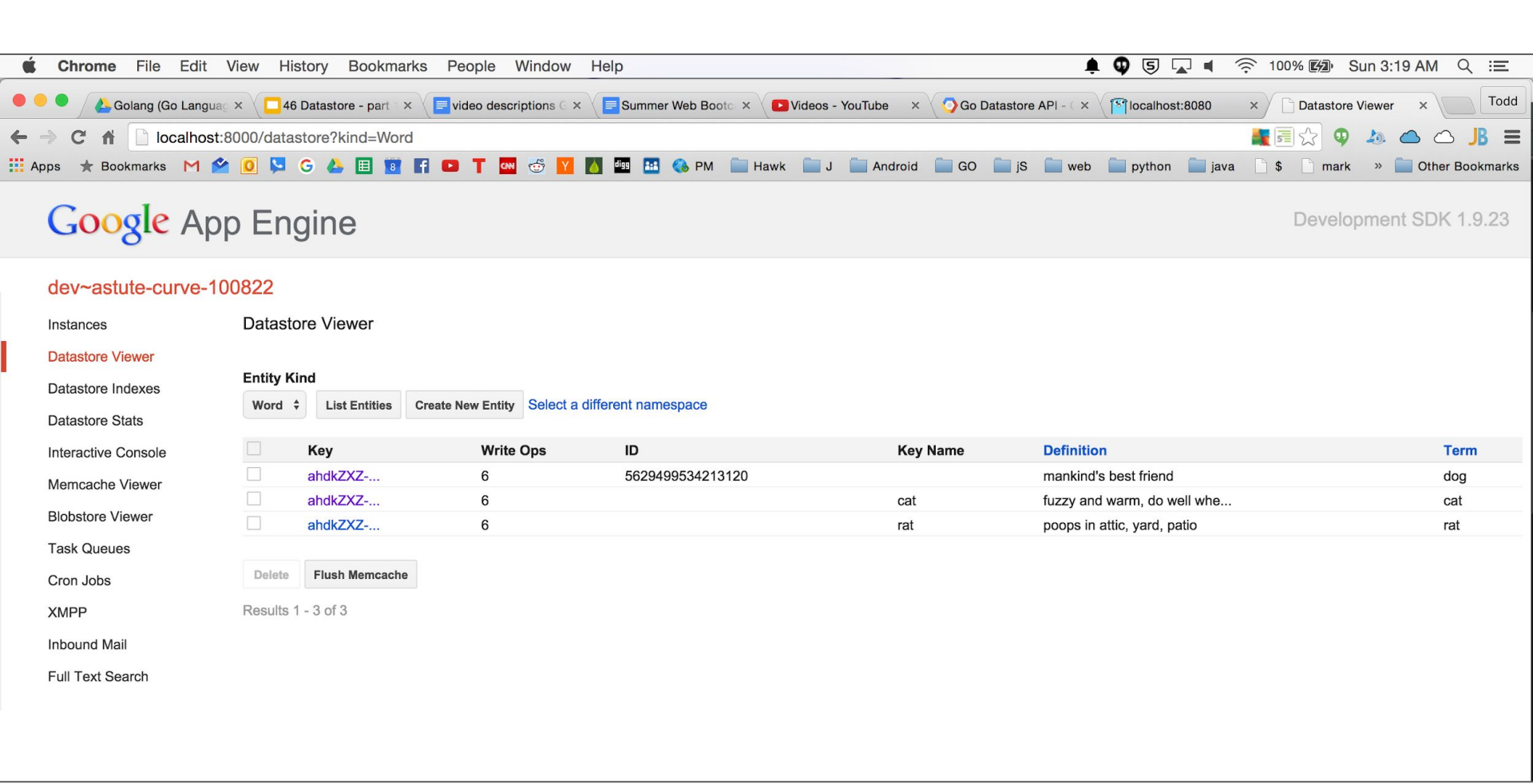
dog

mankind's best friend

rat

poops in attic, yard, patio

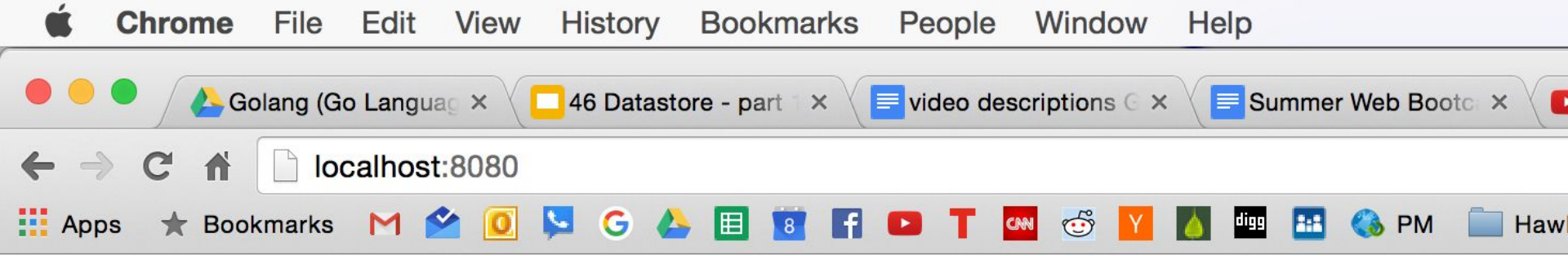
Submit


Chrome File Edit View History Bookmarks People Window Help
100% Sun 3:19 AM
Golang (Go Langua
46 Datastore - part
video descriptions C
Summer Web Boote
Videos - YouTube
Go Datastore API - (
localhost:8080
Datastore Viewer
Todd
localhost:8000/datastore?kind=Word
Apps Bookmarks PM Hawk J Android GO jS web python java $ mark Other Bookmarks
Google App Engine
Development SDK 1.9.23
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Datastore Viewer
Entity Kind
Word
List Entities
Create New Entity
Select a different namespace
Key
Write Ops
ID
Key Name
Definition
Term
ahdkZXZ-...
6
5629499534213120
mankind's best friend
dog
ahdkZXZ-...
6
cat
fuzzy and warm, do well whe...
cat
ahdkZXZ-...
6
rat
poops in attic, yard, patio
rat
Delete
Flush Memcache
Results 1 - 3 of 3

NewQuery

field filter

```
35
36  func listWords(res http.ResponseWriter, req *http.Request) {
37      ctx := appengine.NewContext(req)
38      q := datastore.NewQuery("Word").
39          Filter("Term >=", "c").
40          Filter("Term <", "e").
41          Order("Term")
42
43      html := ""
```



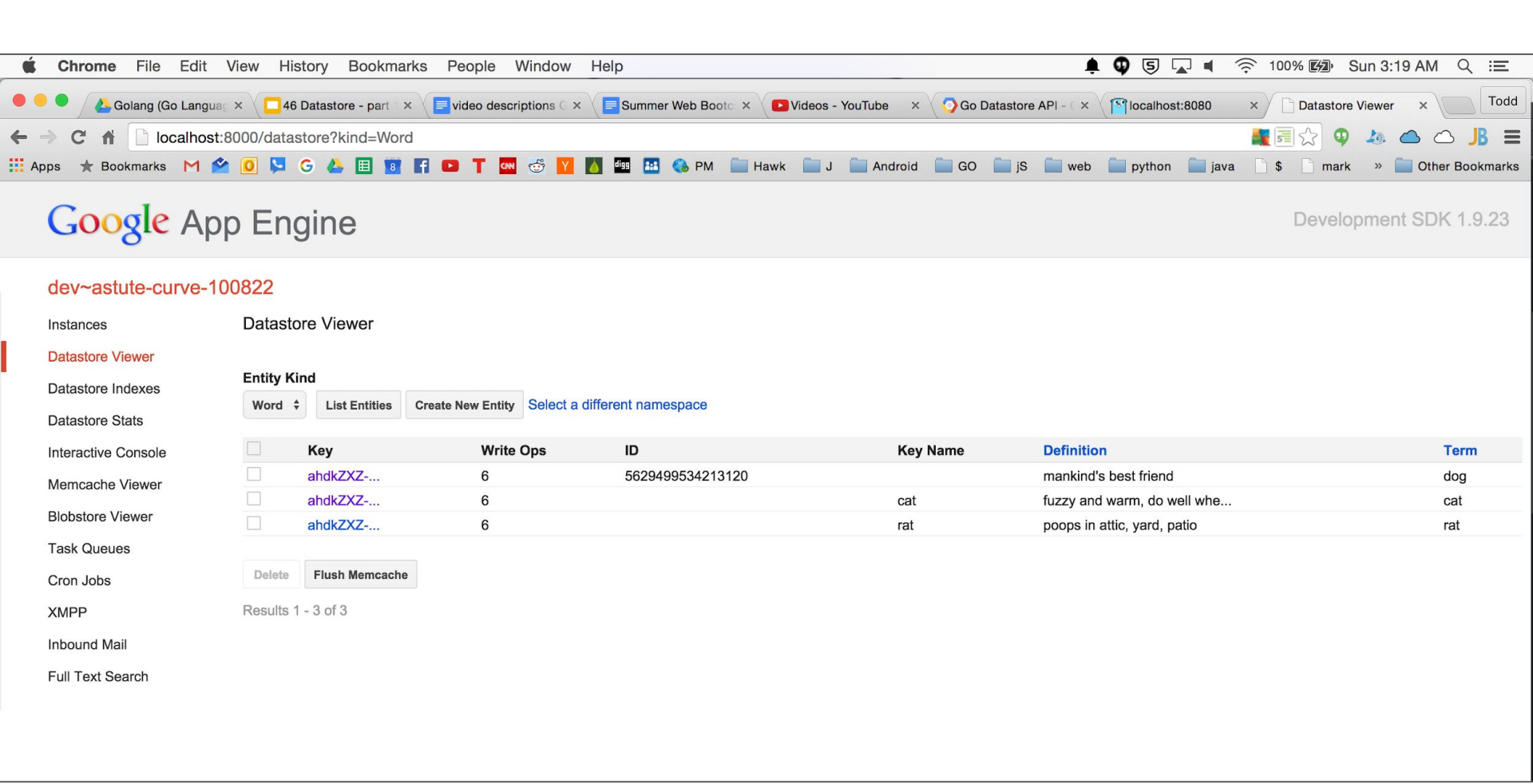
cat

fuzzy and warm, do well when left alone

dog

mankind's best friend

Submit


Chrome File Edit View History Bookmarks People Window Help
100% Sun 3:19 AM
Golang (Go Langua
46 Datastore - part
video descriptions C
Summer Web Boote
Videos - YouTube
Go Datastore API - (
localhost:8080
Datastore Viewer
Todd
localhost:8000/datastore?kind=Word
Apps Bookmarks PM Hawk J Android GO jS web python java $ mark Other Bookmarks
Google App Engine
Development SDK 1.9.23
dev~astute-curve-100822
Instances
Datastore Viewer
Datastore Indexes
Datastore Stats
Interactive Console
Memcache Viewer
Blobstore Viewer
Task Queues
Cron Jobs
XMPP
Inbound Mail
Full Text Search
Datastore Viewer
Entity Kind
Word
List Entities
Create New Entity
Select a different namespace
Key
Write Ops
ID
Key Name
Definition
Term
ahdkZXZ-...
6
5629499534213120
mankind's best friend
dog
ahdkZXZ-...
6
cat
fuzzy and warm, do well whe...
cat
ahdkZXZ-...
6
rat
poops in attic, yard, patio
rat
Delete
Flush Memcache
Results 1 - 3 of 3

docs

The datastore package

```
import "appengine/datastore"
```

Introduction

Package datastore provides a client for App Engine's datastore service.

Basic Operations

Entities are the unit of storage and are associated with a key. A key consists of an optional parent key, a string application ID, a string kind (also known as an entity type), and either a StringID or an IntID. A StringID is also known as an entity name or key name.

It is valid to create a key with a zero StringID and a zero IntID; this is called an incomplete key, and does not refer to any saved entity. Putting an entity into the datastore under an incomplete key will cause a unique key to be generated for that entity, with a non-zero IntID.

An entity's contents are a mapping from case-sensitive field names to values. Valid value types are:

```
- signed integers (int, int8, int16, int32 and int64),
- bool,
- string,
- float32 and float64,
- []byte (up to 1 megabyte in length),
- any type whose underlying type is one of the above predeclared types,
- ByteString,
- *Key,
- time.Time (stored with microsecond precision),
- appengine.BlobKey,
- appengine.GeoPoint,
```

func Put

```
func Put(c appengine.Context, key *Key, src interface{}) (*Key, error)
```

Put saves the entity src into the datastore with key k. src must be a struct pointer or implement PropertyLoadSaver; if a struct pointer then any unexported fields of that struct will be skipped. If k is an incomplete key, the returned key will be a unique key generated by the datastore.

func Get

```
func Get(c appengine.Context, key *Key, dst interface{}) error
```

Get loads the entity stored for k into dst, which must be a struct pointer or implement PropertyLoadSaver. If there is no such entity for the key, Get returns ErrNoSuchEntity.

The values of dst's unmatched struct fields are not modified, and matching slice-typed fields are not reset before appending to them. In particular, it is recommended to pass a pointer to a zero valued struct on each Get call.

ErrFieldMismatch is returned when a field is to be loaded into a different type than the one it was stored from, or when a field is missing or unexported in the destination struct. ErrFieldMismatch is only returned if dst is a struct pointer.

```
type Query
    func NewQuery(kind string) *Query
    func (q *Query) Ancestor(ancestor *Key) *Query
    func (q *Query) Count(c appengine.Context) (int, error)
    func (q *Query) Distinct() *Query
    func (q *Query) End(c Cursor) *Query
    func (q *Query) EventualConsistency() *Query
    func (q *Query) Filter(filterStr string, value interface{}) *Query
    func (q *Query) GetAll(c appengine.Context, dst interface{}) ([]*Key, error)
    func (q *Query) KeysOnly() *Query
    func (q *Query) Limit(limit int) *Query
    func (q *Query) Offset(offset int) *Query
    func (q *Query) Order(fieldName string) *Query
    func (q *Query) Project(fieldNames ...string) *Query
    func (q *Query) Run(c appengine.Context) *Iterator
    func (q *Query) Start(c Cursor) *Query
```

type Query

```
type Query struct {
    // contains filtered or unexported fields
}
```

Query represents a datastore query.

func NewQuery

```
func NewQuery(kind string) *Query
```

NewQuery creates a new Query for a specific entity kind.

An empty kind means to return all entities, including entities created and managed by other App Engine features, and is called a kindless query. Kindless queries cannot include filters or sort orders on property values.

```
35
36  func listWords(res http.ResponseWriter, req *http.Request) {
37      ctx := appengine.NewContext(req)
38      q := datastore.NewQuery("Word").
39          Filter("Term >=", "c").
40          Filter("Term <", "e").
41          Order("Term")
```

func (*Query) Filter

```
func (q *Query) Filter(filterStr string, value interface{}) *Query
```

Filter returns a derivative query with a field-based filter. The filterStr argument must be a field name followed by optional space, followed by an operator, one of ">", "<", ">=", "<=", or "=". Fields are compared against the provided value using the operator. Multiple filters are AND'ed together.

```
35
36  func listWords(res http.ResponseWriter, req *http.Request) {
37      ctx := appengine.NewContext(req)
38      q := datastore.NewQuery("Word").
39          Filter("Term >=", "c").
40          Filter("Term <", "e").
41          Order("Term")
```

func (*Query) Order

```
func (q *Query) Order(fieldName string) *Query
```

Order returns a derivative query with a field-based sort order. Orders are applied in the order they are added. The default order is ascending; to sort in descending order prefix the fieldName with a minus sign (-).

```
35
36  func listWords(res http.ResponseWriter, req *http.Request) {
37      ctx := appengine.NewContext(req)
38      q := datastore.NewQuery("Word").
39          Filter("Term >=", "c").
40          Filter("Term <", "e").
41          Order("Term")
```

Variables

```
var (
    // ErrInvalidEntityType is returned when functions like Get or Next are
    // passed a dst or src argument of invalid type.
    ErrInvalidEntityType = errors.New("datastore: invalid entity type")
    // ErrInvalidKey is returned when an invalid key is presented.
    ErrInvalidKey = errors.New("datastore: invalid key")
    // ErrNoSuchEntity is returned when no entity was found for a given key.
    ErrNoSuchEntity = errors.New("datastore: no such entity")
)
```

```
var Done = errors.New("datastore: query has no more results")
```

Done is returned when a query iteration has completed.

```
var ErrConcurrentTransaction = errors.New("datastore: concurrent transaction")
```

ErrConcurrentTransaction is returned when a transaction is rolled back due to a conflict with a concurrent transaction.

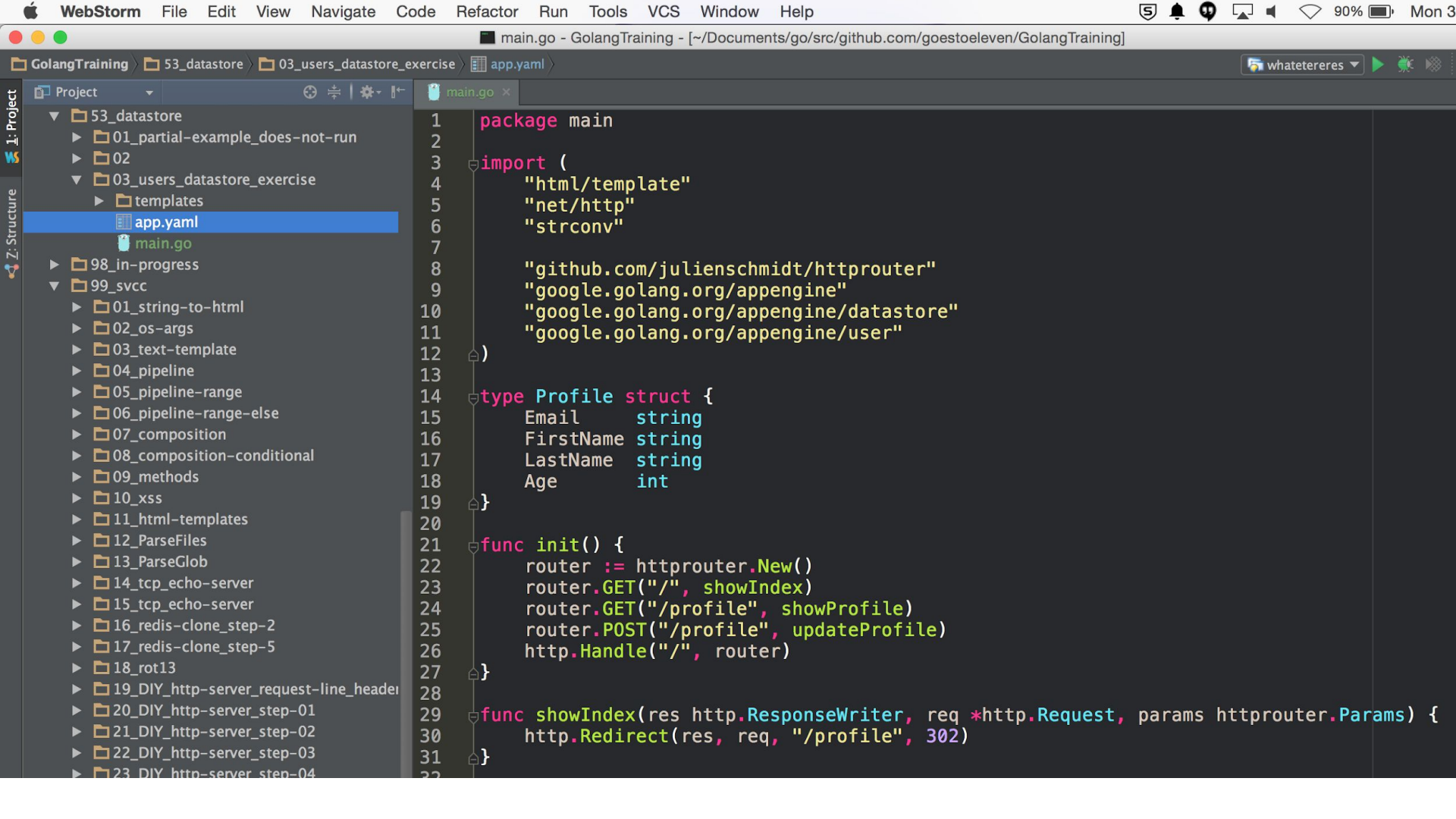
```
35
36  func listWords(res http.ResponseWriter, req *http.Request) {
37      ctx := appengine.NewContext(req)
38      q := datastore.NewQuery("Word").
39          Filter("Term >=", "c").
40          Filter("Term <", "e").
41          Order("Term")
42
43      html := ""
44
45      iterator := q.Run(ctx)
46      for {
47          var entity Word
48          _, err := iterator.Next(&entity)
49          if err == datastore.Done {
50              break
51          } else if err != nil {
52              http.Error(res, err.Error(), 500)
53              return
54          }
55          html += `
```

exercise

Datastore

Create a user profile page that requires login (via Google) and save profile information in the data store.

exercise solution


WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
90% Mon 3
main.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 53_datastore 03_users_datastore_exercise app.yaml
whatetereres
Project
main.go
53_datastore
01_partial-example_does-not-run
02
03_users_datastore_exercise
templates
app.yaml
main.go
98_in-progress
99_svcc
01_string-to-html
02_os-args
03_text-template
04_pipeline
05_pipeline-range
06_pipeline-range-else
07_composition
08_composition-conditional
09_methods
10_xss
11_html-templates
12_ParseFiles
13_ParseGlob
14_tcp_echo-server
15_tcp_echo-server
16_redis-clone_step-2
17_redis-clone_step-5
18_rot13
19_DIY_http-server_request-line_header
20_DIY_http-server_step-01
21_DIY_http-server_step-02
22_DIY_http-server_step-03
23_DIY_http-server_step-04
1 package main
2
3 import (
4 "html/template"
5 "net/http"
6 "strconv"
7
8 "github.com/julienschmidt/httprouter"
9 "google.golang.org/appengine"
10 "google.golang.org/appengine/datastore"
11 "google.golang.org/appengine/user"
12)
13
14 type Profile struct {
15 Email string
16 FirstName string
17 LastName string
18 Age int
19 }
20
21 func init() {
22 router := httprouter.New()
23 router.GET("/", showIndex)
24 router.GET("/profile", showProfile)
25 router.POST("/profile", updateProfile)
26 http.Handle("/", router)
27 }
28
29 func showIndex(res http.ResponseWriter, req *http.Request, params httprouter.Params) {
30 http.Redirect(res, req, "/profile", 302)
31 }

```
32
33  func showProfile(res http.ResponseWriter, req *http.Request, params httprouter.Params) {
34      tpl, err := template.ParseFiles("templates/templates.gohtml")
35      if err != nil {
36          panic(err)
37      }
38
39      ctx := appengine.NewContext(req)
40      u := user.Current(ctx)
41      key := datastore.NewKey(ctx, "Profile", u.Email, 0, nil)
42      var profile Profile
43      err = datastore.Get(ctx, key, &profile)
44      if err != nil {
45          profile.Email = u.Email
46      }
47
48      err = tpl.ExecuteTemplate(res, "edit-form", profile)
49      if err != nil {
50          http.Error(res, err.Error(), 500)
51      }
52  }
```

```
{{define "edit-form"}}<!DOCTYPE html>
<html>
  <head>
    <title>User Profile</title>
    <style>
      html, body {
        padding: 0;
        margin: 0;
        font-size: 18px;
      }
      body {
        background-color: #EAEAEA;
      }
      * {
        box-sizing: border-box;
      }
      form {
        border-radius: 8px;
        padding: 16px;
        border: 2px solid #AAA;
        background: white;
        width: 400px;
        margin: 32px auto;

      }
    </style>
  </head>
  <body>
    <form method="POST">
      <table>
        <tr>
          <td><label for="email">Email</label></td>
          <td><input type="email"
                     readonly="readonly"
                     id="email"
                     name="email"
                     value="{{.Email}}"></td>
        </tr>
        <tr>
          <td><label for="firstname">First Name</label></td>
          <td><input type="text"
                     name="firstname"
                     id="firstname"
                     value="{{.FirstName}}"></td>
        </tr>
        <tr>
          <td><label for="lastname">Last Name</label></td>
          <td><input type="text"
                     name="lastname"
                     id="lastname"
                     value="{{.LastName}}"></td>
        </tr>
        <tr>
          <td><label for="age">Age</label></td>
          <td><input type="text"
                     name="age"
                     id="age"
                     value="{{.Age}}"></td>
        </tr>
        <tr>
          <td colspan="2">
            <input type="submit" value="Update">
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
{{end}}
```

templates.gohtml × main.go ×

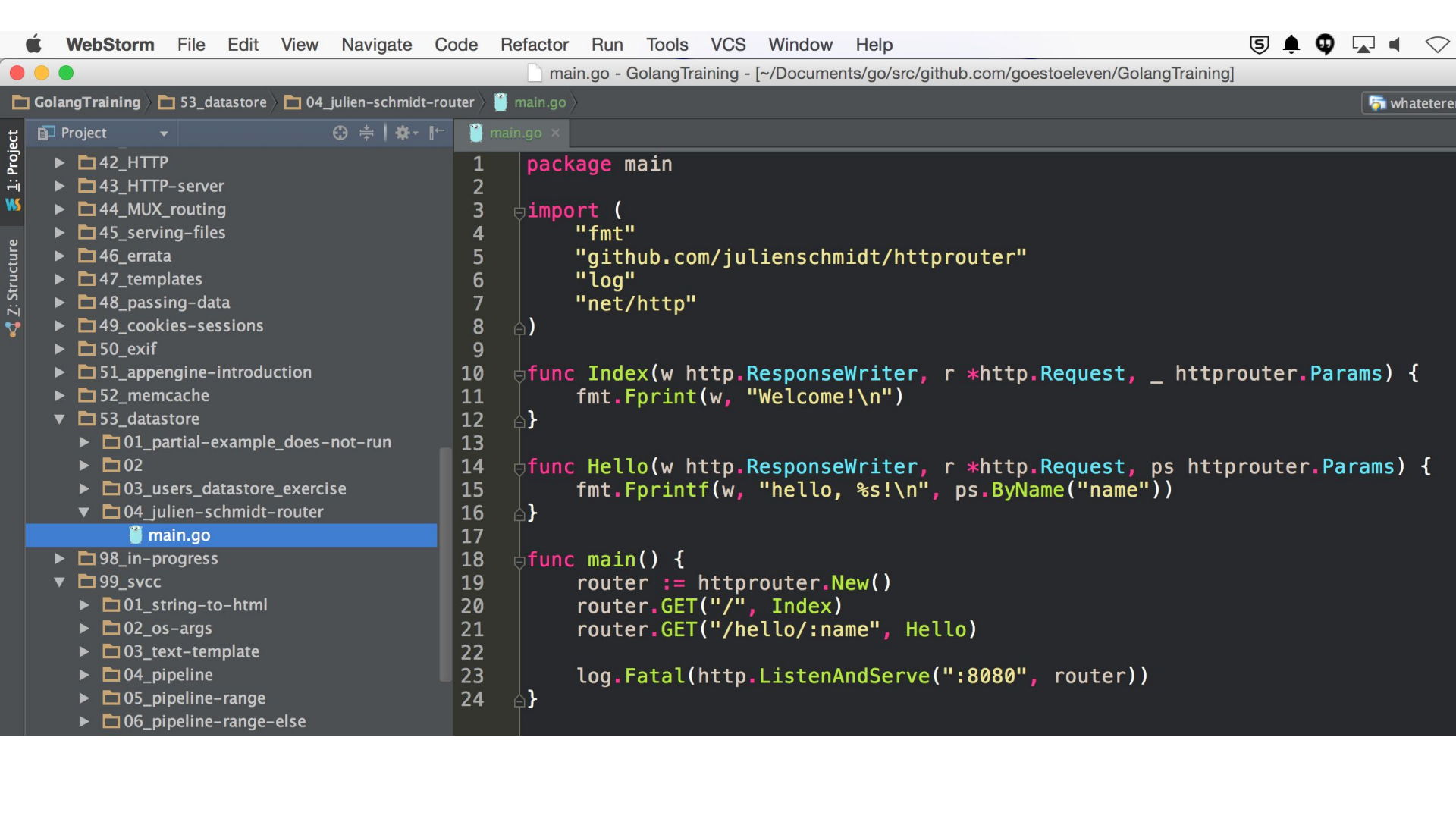
```
<!DOCTYPE html>
<html>
  <head>
    <title>User Profile</title>
    <style>
      html, body {
        padding: 0;
        margin: 0;
```

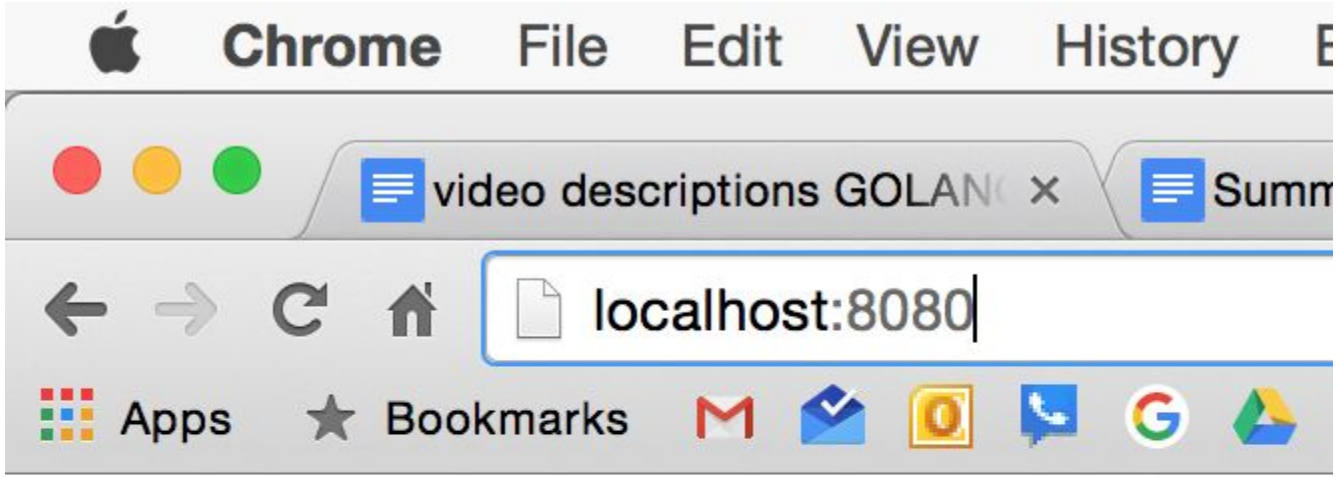
You could also do it like this
no **{{define "edit-form"}}**

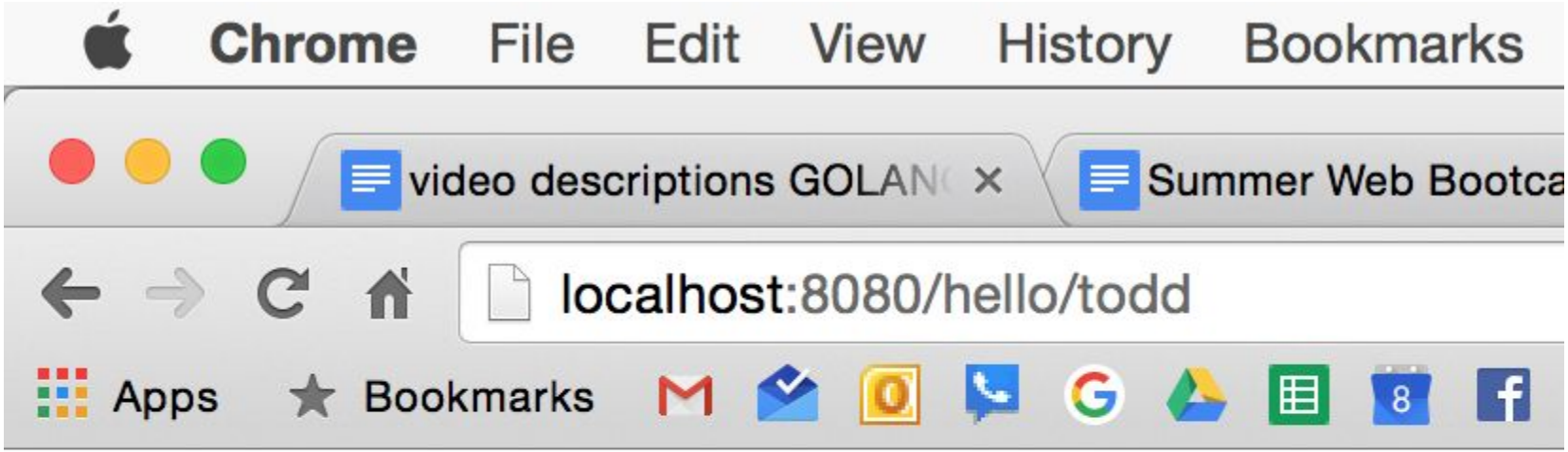
```
func showProfile(res http.ResponseWriter, req *http.Request, params httprouter.Params) {
    tpl, err := template.ParseFiles("templates/templates.gohtml")
    if err != nil {
        panic(err)
    }

    ctx := appengine.NewContext(req)
    u := user.Current(ctx)
    key := datastore.NewKey(ctx, "Profile", u.Email, 0, nil)
    var profile Profile
    err = datastore.Get(ctx, key, &profile)
    if err != nil {
        profile.Email = u.Email
    }

    err = tpl.ExecuteTemplate(res, "templates.gohtml", profile)
    if err != nil {
        http.Error(res, err.Error(), 500)
    }
}
```

```go
func updateProfile(res http.ResponseWriter, req *http.Request, params httprouter.Params) {
    ctx := appengine.NewContext(req)
    u := user.Current(ctx)
    key := datastore.NewKey(ctx, "Profile", u.Email, 0, nil)
    age, _ := strconv.Atoi(req.FormValue("age"))
    profile := Profile{
        Email:     u.Email,
        FirstName: req.FormValue("firstname"),
        LastName:  req.FormValue("lastname"),
        Age:       age,
    }
    _, err := datastore.Put(ctx, key, &profile)
    if err != nil {
        http.Error(res, err.Error(), 500)
    }
    http.Redirect(res, req, "/profile", 302)
}
```

julien schmidt router


WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
main.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 53_datastore 04_julien-schmidt-router main.go
Project
main.go
42_HTTP
43_HTTP-server
44_MUX_routing
45_serving-files
46_errata
47_templates
48_passing-data
49_cookies-sessions
50_exif
51_appengine-introduction
52_memcache
53_datastore
01_partial-example_does-not-run
02
03_users_datastore_exercise
04_julien-schmidt-router
main.go
98_in-progress
99_svcc
01_string-to-html
02_os-args
03_text-template
04_pipeline
05_pipeline-range
06_pipeline-range-else
1: Project
Z: Structure
1 package main
2
3 import (
4 "fmt"
5 "github.com/julienschmidt/httprouter"
6 "log"
7 "net/http"
8)
9
10 func Index(w http.ResponseWriter, r *http.Request, _ httprouter.Params) {
11 fmt.Fprint(w, "Welcome!\n")
12 }
13
14 func Hello(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
15 fmt.Fprintf(w, "hello, %s!\n", ps.ByName("name"))
16 }
17
18 func main() {
19 router := httprouter.New()
20 router.GET("/", Index)
21 router.GET("/hello/:name", Hello)
22
23 log.Fatal(http.ListenAndServe(":8080", router))
24 }


Chrome File Edit View History
video descriptions GOLAN
Summ
localhost:8080
Apps
Bookmarks
Welcome!


Chrome File Edit View History Bookmarks
video descriptions GOLAN
Summer Web Bootca
localhost:8080/hello/todd
Apps
Bookmarks
hello, todd!

extra reading

(not for in-class lecture)

nosql

NoSQL

From Wikipedia, the free encyclopedia

> *"Structured storage" redirects here. For the Microsoft technology also known as structured storage, see COM Structured Storage.*

A **NoSQL** (originally referring to "non SQL" or "non relational" [1]) database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[2] triggered by the needs of Web 2.0 companies such as Facebook, Google and Amazon.com.[3][4][5]

Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines, which is a problem for relational databases,[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, graph, or document) differ slightly from those used by default in relational databases, making some operations faster in NoSQL and others faster in relational databases. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by noSQL databases are also viewed as "more flexible" than relational database tables.[6]

NoSQL databases are increasingly used in big data and real-time web applications.[7] NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.[8][9]

Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc JOIN's across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[10] Most NoSQL stores lack true ACID transactions, although a few recent systems, such as FairCom c-treeACE, Google Spanner (though technically a NewSQL database), FoundationDB, Symas LMDB and OrientDB have made them central to their designs. (See ACID and JOIN Support.) Instead they offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately.

Not all NoSQL systems live up to the promised "eventual consistency" and partition tolerance, but in experiments with network partitioning often exhibited lost writes and other forms of data loss.[11] Fortunately, some NoSQL systems provide concepts such as Write-ahead logging to avoid data loss.[12] Current relational databases also "do not allow referential integrity constraints to span databases" as well.[13]

Types and examples of NoSQL databases [edit]

There have been various approaches to classify NoSQL databases, each with different categories and subcategories, some of which overlap. A basic classification based on data model, with examples:

- **Column**: Accumulo, Cassandra, Druid, HBase, Vertica
- **Document**: Clusterpoint, Apache CouchDB, Couchbase, DocumentDB, HyperDex, Lotus Notes, MarkLogic, MongoDB, OrientDB, Qizx
- **Key-value**: CouchDB, Oracle NoSQL Database, Dynamo, FoundationDB, HyperDex, MemcacheDB, Redis, Riak, FairCom c-treeACE, Aerospike, OrientDB, MUMPS
- **Graph**: Allegro, Neo4J, InfiniteGraph, OrientDB, Virtuoso, Stardog
- **Multi-model**: OrientDB, FoundationDB, ArangoDB, Alchemy Database, CortexDB

A more detailed classification is the following, based on one from Stephen Yen:[19]

Key-value stores [edit]

Main article: Key-value database

Key-value (KV) stores use the associative array (also known as a map or dictionary) as their fundamental data model. In this model, data is represented as a collection of key-value pairs, such that each possible key appears at most once in the collection.[20][21]

The key-value model is one of the simplest non-trivial data models, and richer data models are often implemented on top of it. The key-value model can be extended to an *ordered* model that maintains keys in lexicographic order. This extension is powerful, in that it can efficiently process key *ranges*.[22]

Key-value stores can use consistency models ranging from eventual consistency to serializability. Some support ordering of keys. Some maintain data in memory (RAM), while others employ solid-state drives or rotating disks.

Examples include Oracle NoSQL Database, redis, and dbm.

Document store [edit]

Main articles: Document-oriented database and XML database

The central concept of a document store is the notion of a "document". While each document-oriented database implementation differs on the details of this definition, in general, they all assume that documents encapsulate and encode data (or information) in some standard formats or encodings. Encodings in use include XML, YAML, and JSON as well as binary forms like BSON. Documents are addressed in the database via a unique *key* that represents that document. One of the other defining characteristics of a document-oriented database is that in addition to the key lookup performed by a key-value store, the database offers an API or query language that retrieves documents based on their contents

Different implementations offer different ways of organizing and/or grouping documents:

- Collections
- Tags
- Non-visible metadata
- Directory hierarchies

Compared to relational databases, for example, collections could be considered analogous to tables and documents analogous to records. But they are different: every record in a table has the same sequence of fields, while documents in a collection may have fields that are completely different.