Разработка REST-серверов на Go. Часть 1: стандартная библиотека

Это — первый материал из серии статей, посвящённой разработке REST-серверов на Go. В этих статьях я планирую описать реализацию простого REST-сервера с использованием нескольких различных подходов. В результате эти подходы можно будет сравнить друг с другом, можно будет понять их относительные преимущества друг перед другом.  
  
Первый вопрос разработчиков, которые только начинают применять Go, часто выглядит так: «Какой фреймворк стоит использовать для решения задачи X». И хотя это — совершенно нормальный вопрос, если задавать его, имея в виду веб-приложения и серверы, написанные на многих других языках, в случае с Go при ответе на этот вопрос нужно принять во внимание множество тонкостей. Существуют серьёзные аргументы как «за», так и «против» использования фреймворков в Go-проектах. Я, работая над статьями из этой серии, вижу своей целью объективное разностороннее исследование этого вопроса.  
  
 Предыдущие части:  
  
 Вы тут — Разработка REST-серверов на Go. Часть 1: стандартная библиотека  
Разработка REST-серверов на Go. Часть 2: применение маршрутизатора gorilla/mux  
Разработка REST-серверов на Go. Часть 3: использование веб-фреймворка Gin  
Разработка REST-серверов на Go. Часть 4: применение OpenAPI и Swagger  
Разработка REST-серверов на Go. Часть 5: Middleware  
Разработка REST-серверов на Go. Часть 6: аутентификация  
  
 Задача  
Для начала хочу сказать, что тут я исхожу из предположения о том, что читателю знакомо понятие «REST-сервер». Если вам нужно освежить знания — взгляните на этот хороший материал (но есть и много других подобных статей). Дальше я буду считать, что вы поймёте, что я имею в виду, когда я буду использовать понятия «путь», «HTTP-заголовок», «код ответа» и прочие подобные.  
  
В нашем случае сервер представляет собой простую бэкенд-систему для приложения, реализующего функционал управления задачами (вроде Google Keep, Todoist и прочих подобных). Сервер предоставляет клиентам следующий REST API:  
  
POST   /task/              :  создаёт задачу и возвращает её ID  
GET    /task/<taskid>      :  возвращает одну задачу по её ID  
GET    /task/              :  возвращает все задачи  
DELETE /task/<taskid>      :  удаляет задачу по ID  
GET    /tag/<tagname>      :  возвращает список задач с заданным тегом  
GET    /due/<yy>/<mm>/<dd> :  возвращает список задач, запланированных на указанную дату  
  
Обратите внимание на то, что этот API создан специально для нашего примера. В следующих материалах этой серии статей мы поговорим о более структурированном и стандартизированном подходе к проектированию API.  
  
Наш сервер поддерживает GET-, POST- и DELETE-запросы, некоторые из них — с возможностью использования нескольких путей. То, что в описании API приведено в угловых скобках (<...>), обозначает параметры, которые клиент предоставляет серверу в виде части запроса. Например, запрос GET /task/42 направлен на получение с сервера задачи с ID 42. ID — это уникальные идентификаторы задач.  
  
Для кодирования данных используется формат JSON. При выполнении запроса POST /task/ клиент отправляет серверу JSON-представление задачи, которую нужно создать. И, аналогично, в ответах на те запросы, в описании которых сказано, что они что-то «возвращают», содержатся JSON-данные. В частности, они размещаются в теле HTTP-ответов.  
  
Код  
Далее мы будем заниматься поэтапным написанием кода сервера на Go. Его полный вариант можно найти здесь. Это — самодостаточный Go-модуль, в котором не используются зависимости. После клонирования или копирования директории проекта на компьютер сервер можно тут же, ничего дополнительно не устанавливая, запустить:  
  
$ SERVERPORT=4112 go run .  
  
Обратите внимание на то, что в качестве SERVERPORT можно использовать любой порт, который будет прослушивать локальный сервер в ожидании подключений. После того, как сервер будет запущен, с ним, используя отдельное окно терминала, можно поработать, воспользовавшись, например, утилитой curl. Взаимодействовать с ним можно и с помощью каких-нибудь других подобных программ. Примеры команд, использующихся для отправки запросов к серверу, можно найти в этом скрипте. В директории, содержащей этот скрипт, есть средства для автоматизированного тестирования сервера.  
  
Модель  
Начнём с обсуждения модели (или «уровня данных») для нашего сервера. Найти её можно в пакете taskstore (internal/taskstore в директории проекта). Это — простая абстракция, представляющая базу данных, в которой хранятся задачи. Вот её API:  
  
func New() \*TaskStore  
  
// CreateTask создаёт новую задачу в хранилище.  
func (ts \*TaskStore) CreateTask(text string, tags []string, due time.Time) int  
  
// GetTask получает задачу из хранилища по ID. Если ID не существует -  
// будет возвращена ошибка.  
func (ts \*TaskStore) GetTask(id int) (Task, error)  
  
// DeleteTask удаляет задачу с заданным ID. Если ID не существует -  
// будет возвращена ошибка.  
func (ts \*TaskStore) DeleteTask(id int) error  
  
// DeleteAllTasks удаляет из хранилища все задачи.  
func (ts \*TaskStore) DeleteAllTasks() error  
  
// GetAllTasks возвращает из хранилища все задачи в произвольном порядке.  
func (ts \*TaskStore) GetAllTasks() []Task  
  
// GetTasksByTag возвращает, в произвольном порядке, все задачи  
// с заданным тегом.  
func (ts \*TaskStore) GetTasksByTag(tag string) []Task  
  
// GetTasksByDueDate возвращает, в произвольном порядке, все задачи, которые  
// запланированы на указанную дату.  
func (ts \*TaskStore) GetTasksByDueDate(year int, month time.Month, day int) []Task  
  
Вот — объявление типа Task:  
  
type Task struct {  
  Id   int       `json:"id"`  
  Text string    `json:"text"`  
  Tags []string  `json:"tags"`  
  Due  time.Time `json:"due"`  
}  
  
В пакете taskstore этот API реализован с использованием простого словаря map[int]Task, данные при этом хранятся в памяти. Но несложно представить себе реализацию этого API, основанную на базе данных. В реальном приложении TaskStore, вероятнее всего, будет интерфейсом, реализовать который могут разные бэкенды. Но для нашего простого примера достаточно и такого API. Если вы хотите поупражняться — реализуйте TaskStore с использованием чего-то вроде MongoDB.  
  
Подготовка сервера к работе  
Функция main нашего сервера устроена довольно просто:  
  
func main() {  
  mux := http.NewServeMux()  
  server := NewTaskServer()  
  mux.HandleFunc("/task/", server.taskHandler)  
  mux.HandleFunc("/tag/", server.tagHandler)  
  mux.HandleFunc("/due/", server.dueHandler)  
  
 log.Fatal(http.ListenAndServe("localhost:"+os.Getenv("SERVERPORT"), mux))  
}  
  
Уделим немного времени команде NewTaskServer, а потом поговорим о маршрутизаторе и об обработчиках путей.  
  
NewTaskServer — это конструктор для нашего сервера, имеющего тип taskServer. Сервер включает в себя TaskStore, что безопасно с точки зрения конкурентного доступа к данным.  
  
type taskServer struct {  
  store \*taskstore.TaskStore  
}  
  
func NewTaskServer() \*taskServer {  
  store := taskstore.New()  
  return &taskServer{store: store}  
}  
  
Маршрутизация и обработчики путей  
Теперь вернёмся к маршрутизации. Тут используются стандартный HTTP-мультиплексор, входящий в состав пакета net/http:  
  
mux.HandleFunc("/task/", server.taskHandler)  
mux.HandleFunc("/tag/", server.tagHandler)  
mux.HandleFunc("/due/", server.dueHandler)  
  
Стандартный мультиплексор обладает достаточно скромными возможностями. В этом — и его сила, и его слабость. Его сильная сторона заключается в том, что с ним очень легко разобраться, так как в его работе нет ничего сложного. А слабость стандартного мультиплексора выражается в том, что иногда его применение делает решение задачи сопоставления запросов с имеющимися в системе путями достаточно утомительным. То, что, по логике вещей, хорошо было бы расположить в каком-то одном месте, приходится размещать в разных местах. Скоро мы поговорим об этом подробнее.  
  
Так как стандартный мультиплексор поддерживает только точное сопоставление запросов с префиксами путей, мы практически вынуждены ориентироваться лишь на корневые пути, находящиеся на верхнем уровне, и передавать задачу поиска точного пути обработчикам путей.  
  
Изучим обработчик путей taskHandler:  
  
func (ts \*taskServer) taskHandler(w http.ResponseWriter, req \*http.Request) {  
  if req.URL.Path == "/task/" {  
  
 // Запрос направлен к "/task/", без идущего в конце ID.  
  
 if req.Method == http.MethodPost {  
  
 ts.createTaskHandler(w, req)  
  
 } else if req.Method == http.MethodGet {  
  
 ts.getAllTasksHandler(w, req)  
  
 } else if req.Method == http.MethodDelete {  
  
 ts.deleteAllTasksHandler(w, req)  
  
 } else {  
  
 http.Error(w, fmt.Sprintf("expect method GET, DELETE or POST at /task/, got %v", req.Method), http.StatusMethodNotAllowed)  
  
 return  
  
 }  
  
Мы начинаем работу с проверки на точное совпадение пути с /task/ (это означает, что в конце нет <taskid>). Тут нам нужно понять то, какой HTTP-метод используется, и вызвать соответствующий метод сервера. Большинство обработчиков путей — это достаточно простые обёртки для API TaskStore. Посмотрим на один из таких обработчиков:  
  
func (ts \*taskServer) getAllTasksHandler(w http.ResponseWriter, req \*http.Request) {  
  log.Printf("handling get all tasks at %s\n", req.URL.Path)  
  
 allTasks := ts.store.GetAllTasks()  
  js, err := json.Marshal(allTasks)  
  if err != nil {  
  
 http.Error(w, err.Error(), http.StatusInternalServerError)  
  
 return  
  }  
  w.Header().Set("Content-Type", "application/json")  
  w.Write(js)  
}  
  
Он решает две основные задачи:  
  
  
Получает данные из модели (TaskStore).  
Формирует HTTP-ответ для клиента.  
  
Обе эти задачи достаточно просты и понятны, но если исследовать код других обработчиков путей, можно обратить внимание на то, что вторая задача имеет свойство повторяться — она заключается в маршалинге JSON-данных, в подготовке корректного HTTP-заголовка ответа и в выполнении других подобных действий. Мы ещё раз поднимем этот вопрос позже.  
  
Вернёмся теперь к taskHandler. Пока мы видели только то, как он обрабатывает запросы, в которых имеется точное совпадение с путём /task/. А как насчёт пути /task/<taskid>? Именно тут в дело вступает вторая часть функции:  
  
} else {  
  // В запросе есть ID, выглядит он как "/task/<id>".  
  path := strings.Trim(req.URL.Path, "/")  
  pathParts := strings.Split(path, "/")  
  if len(pathParts) < 2 {  
  
 http.Error(w, "expect /task/<id> in task handler", http.StatusBadRequest)  
  
 return  
  }  
  id, err := strconv.Atoi(pathParts[1])  
  if err != nil {  
  
 http.Error(w, err.Error(), http.StatusBadRequest)  
  
 return  
  }  
  
 if req.Method == http.MethodDelete {  
  
 ts.deleteTaskHandler(w, req, int(id))  
  } else if req.Method == http.MethodGet {  
  
 ts.getTaskHandler(w, req, int(id))  
  } else {  
  
 http.Error(w, fmt.Sprintf("expect method GET or DELETE at /task/<id>, got %v", req.Method), http.StatusMethodNotAllowed)  
  
 return  
  }  
}  
  
Когда запрос не в точности соответствует пути /task/, мы ожидаем, что за косой чертой будет идти числовой ID задачи. Вышеприведённый код анализирует этот ID и вызывает соответствующий обработчик (основываясь на методе HTTP-запроса).  
  
Остальной код более или менее похож на тот, что мы уже рассмотрели, понять его должно быть несложно.  
  
Улучшение сервера  
Теперь, когда у нас есть базовая работающая версия сервера, пришло время подумать о возможных проблемах, которые могут с ним возникнуть, и о том, как его улучшить.  
  
Одной из используемых нами программных конструкций, которая, очевидно, нуждается в улучшении, и о которой мы уже говорили, является повторяющийся код подготовки JSON-данных при формировании HTTP-ответов. Я создал отдельную версию сервера, stdlib-factorjson, в которой эта проблема решена. Я выделил эту реализацию сервера в отдельную папку для того чтобы её было легче сравнить с первоначальным кодом сервера и проанализировать изменения. Главное новшество этого кода представлено следующей функцией:  
  
// renderJSON преобразует 'v' в формат JSON и записывает результат, в виде ответа, в w.  
func renderJSON(w http.ResponseWriter, v interface{}) {  
  js, err := json.Marshal(v)  
  if err != nil {  
  
 http.Error(w, err.Error(), http.StatusInternalServerError)  
  
 return  
  }  
  w.Header().Set("Content-Type", "application/json")  
  w.Write(js)  
}  
  
Используя эту функцию, мы можем переписать код всех обработчиков путей, укоротив его. Вот, например, как теперь выглядит код getAllTasksHandler:  
  
func (ts \*taskServer) getAllTasksHandler(w http.ResponseWriter, req \*http.Request) {  
  log.Printf("handling get all tasks at %s\n", req.URL.Path)  
  
 allTasks := ts.store.GetAllTasks()  
  renderJSON(w, allTasks)  
}  
  
Более фундаментальное улучшение заключалось бы в том, чтобы сделать код сопоставления запросов и путей чище, и в том, чтобы, по возможности, собрать этот код в одном месте. Хотя текущий подход по сопоставлению запросов и путей упрощает отладку, соответствующий код трудно понять с первого взгляда, так как он разбросан по нескольким функциям. Например, предположим, что мы пытаемся разобраться с тем, как обрабатывается запрос DELETE, который направлен к /task/<taskid>. Для этого выполняются следующие действия:  
  
  
Во-первых — мы находим в main мультиплексор и узнаём, что корневой путь /task/ обрабатывается в taskHandler.  
Далее, в taskHandler, нам надо найти выражение else, которое отвечает за обработку путей, не точно совпадающих с /task/. Там нам надо прочитать код преобразования <taskid> в целое число.  
И наконец — мы смотрим на выражение if, в котором перечислены различные методы, применяемые при обработке запросов, соответствующих этому пути, и выясняем, что метод DELETE обрабатывается в deleteTaskHandler.  
  
Можно поместить весь этот код в одно место. Так работать с ним будет гораздо проще и удобнее. Именно на решение этой задачи направлены HTTP-маршрутизаторы сторонних разработчиков. О них мы поговорим во второй части этой серии статей.  
  
❒ Это — первая часть из серии статей, посвящённой разработке серверов на Go. Посмотреть список статей можно в начале оригинала этого материала.  
  
 Переводы этой серии:  
  
 Вы тут — Разработка REST-серверов на Go. Часть 1: стандартная библиотека  
Разработка REST-серверов на Go. Часть 2: применение маршрутизатора gorilla/mux  
Разработка REST-серверов на Go. Часть 3: использование веб-фреймворка Gin  
Разработка REST-серверов на Go. Часть 4: применение OpenAPI и Swagger  
Разработка REST-серверов на Go. Часть 5: Middleware  
Разработка REST-серверов на Go. Часть 6: аутентификация  
  
 Только зарегистрированные пользователи могут участвовать в опросе. Войдите, пожалуйста. Скажите, стоит ли переводить остальные статьи?   
  
 88.52%  
  
 Да   
  
 162  
  
 11.48%  
  
 Нет   
  
 21  
  
 Проголосовали 183 пользователя.   
  
 Воздержались 24 пользователя.