АНАЛИЗ ЧИСТОЙ АРХИТЕКТУРЫ GOLANG REST API С ВНЕДРЕНИЕМ ЗАВИСИМОСТЕЙ, СЛЕДУЯ ПРИНЦИПАМ SOLID

Коптева А.В.[1], Князев И.В.[2]

[1]Коптева Анна Витальевна – старший разработчик программного обеспечения , Яндекс, г. Москва;
[2]Князев Илья Вадимович – старший разработчик программного обеспечения, June Homes, г. Белгород

Аннотация: *в данной статье рассматривается пример чистой архитектуры Golang, реализующей внедрение зависимостей и mock-объектов для выполнения модульного тестирования с целью получения надежного и безопасного исходного кода. Идея самого шаблона состоит в том, чтобы создать разделенные системы, в которых реализация домена нижнего уровня не зависит от реализации верхнего и может быть заменена, не затрагивая бизнес-логику распределенной системы и не нарушая целостность системы.*
Ключевые слова: *чистая архитектура, golang, mock-объект, внедрение зависимостей, модульное тестирование.*

ANALYZE THE CLEAN ARCHITECTURE OF GOLANG REST API WITH DEPENDENCY INJECTION FOLLOWING SOLID PRINCIPLES

Kopteva A.V.[1], Kniazev I.V.[2]

[1]Kopteva Anna Vitalievna - Senior Software Developer, YANDEX,MOSCOW;
[2]Kniazev Ilya Vadimovich - Senior Software Developer, JUNE HOMES,BELGOROD

Abstract: *this article discusses an example of a clean Golang architecture that implements dependency injection and mock injection to perform unit testing for reliable and secure source code. The idea behind the pattern itself is to create partitioned systems in which the implementation of the lower-level domain is independent of the implementation of the upper-level and can be replaced without affecting the business logic of the distributed system or compromising the integrity of the system.*
Keywords: *clean architecture, golang, mock-object, dependency injection, module testing.*

Вступление

Целью архитектуры является создание системы, которая:

— Независима от фреймворка. Система должна иметь возможность стать независимой системой, не связанной с какой-либо реализацией фреймворка, которая приводит к раздуванию системы. Фреймворка должен быть использован как инструмент для поддержки реализации системы, а не ограничения возможностей системы.

— Хорошо поддаётся проверке. Исходный код может содержать ошибки, и тесты - единственный способ не допустить их. Это означает, что покрытие тестами должно охватывать как можно больше слоев,что повышает надежность кода.

— Независима от базы данных. Бизнес-логика не должна быть привязана к базе данных, система должна иметь возможность в случае необходимости менять базы данных, например, MySQL, Maria DB, PosgreSQL, Mongo DB, Dynamo DB без нарушения логики.

— Независима от сторонних библиотек. Никакая сторонняя библиотека не должна быть реализована непосредственно в системной логике, необходимо абстрагироваться от того, что система может заменить библиотеку в зависимости от требований.

Каждая реализация должна осуществляться только с использованием интерфейса, у разработчика не должно быть прямого доступа к реализации, это предоставляет возможность внедрить зависимость и заменить ее mock-объектом во время модульных тестов. Например: UserService -> реализует IUserRepository вместо прямого UserRepository.

```
type UserService struct {
  interfaces.IUserRepository
}

func (service *UserService) GetPoints(user1Name string, user2Name string) (string, error) {
  basePoint := [4]string{"Love", "Fifteen", "Thirty", "Forty"}
```

```
 var result string
 user1, err := service.GetUserByName(user1Name)
 if err != nil {
  //Handle error
 }
 user2, err := service.GetUserByName(user2Name)
 if err != nil {
  //Handle error
 }
 if user1.Point < 4 && user2.Point < 4 && !(user1.Point+user2.Point == 6) {
  s := basePoint[user1.Point]
  if user1.Point == user2.Point {
   result = s + "-All"
  } else {
    result = s + "-" + basePoint[user2.Point]
  }
 }
 if user1.Point == user2.Point {
  result = "Deuce"
 }
 return result, nil
}
```

Пример реализации:

```
user1, err := service.GetUserByName(user1Name)
user2, err := service.GetUserByName(user2Name)
```

На самом деле оба примера являются абстрактной реализацией интерфейса, а не самой реальной реализацией. Таким образом, имеется возможность переключить реализацию IUserService и IUserRepository во время инъекции на любую реализацию без изменения логики реализации.

Структура проекта

Структура папок должна быть создана с учетом принципа разделения ответственности, при котором каждая структура должна нести ответственность за достижение несвязности системы. Ниже представлена рекомендуемая структура проекта.

```
/
|- controllers
|- infrastructures
|- interfaces
|- models
|- repositories
|- services
|- viewmodels
main.go
router.go
servicecontainer.go
```

Каждая папка представляет собой собственное пространство имен, и каждый файл или структура в одной папке должны использовать только то же пространство имен, что и их корневая папка.

В *controllers* размещаются все структуры в пространстве имен контроллеров, контроллеры являются обработчиком всех входящих запросов к маршрутизатору, он делает именно это, бизнес-логика и уровень доступа к данным должны выполняться отдельно.

infrasctructures содержит все структуры в пространстве имен infrasctructures, В infrasctructures осуществляется настройка системы для подключения к внешнему источнику данных, она используется для размещения таких вещей, как конфигурации подключения к базе данных, MySQL, MariaDB, MongoDB, DynamoDB.

interfaces содержит все структуры в пространстве имен *interfaces*. Интерфейсы являются мостом между разными доменами, поэтому они могут взаимодействовать друг с другом, в конкретном случае это должен быть для них единственный способ взаимодействия.

Интерфейс в Golang немного отличается от того, который можно найти в других языках программирования, таких как Java или C #, в то время как последние реализуют интерфейс явно, Golang реализует интерфейс неявно. Необходимо реализовать все методы, которые есть в интерфейсе.

В данной системе UserController реализует IUserService, чтобы иметь возможность взаимодействовать с введенной реализацией, то есть IUserService будет внедрен с UserService.

То же самое относится к UserService, который реализует IUserRepository, что дает возможность взаимодействовать с внедренной реализацией. В нашем случае IUserRepository будет внедрен с UserRepository во время компиляции. UserRepository, с другой стороны, будет внедрен с конфигурацией инфраструктуры, которая была настроена ранее, это предоставляет возможность изменить реализацию UserRepository, без изменения реализатора, который в данном случае UserService. То же самое касается отношений UserService и UserController, можно реорганизовать UserService, не касаясь реализатора, которым является UserController.

models содержит все структуры в пространстве имен models, модель - это структура, отражающая объект данных базы данных. Модели должны определять только структуры данных, другие функции не должны быть включены в данную структуру.

repositories содержит все структуры в пространстве имен repositories. В репозиториях реализована реализация уровня доступа к данным. Все запросы и операции с данными базы данных должны происходить здесь, и реализатор должен быть независимым от того, какое ядро базы данных используется, как выполняются запросы. Этот уровень отвечает за извлечение данных в соответствии с интерфейсом, который они реализуют.

services размещены все структуры в пространстве имен services. В службах находится бизнес-логика, она обрабатывает запросы контроллера, извлекает данные с нужного уровня данных и выполняет их обработку, чтобы вернуть результат контроллеру. Контроллер может реализовать множество интерфейсов сервисов для необходимых запросов, и контроллер должен быть независимым от того, как сервисы реализуют свою логику, зона их ответственности, — это то, что они должны иметь возможность получать требуемый результат в соответствии с интерфейсом, который они реализуют.

viewmodels содержит все структуры в пространстве имен viewmodels, модели viewmodels — это модели, которые будут использоваться в качестве ответа, возвращаемого при вызове REST API.

main.go - это точка входа в систему. Здесь находятся привязки маршрутизатора, он запускает синглтон ChiRouter и вызывает InitRouter для привязки маршрутизатора.

router.go - это место привязки контроллеров к соответствующим маршрутам для обработки желаемого HTTP-запроса.

servicecontainer.go - это место, где внедрены все реализации интерфейсов. Подробно рассмотрим раздел внедрения зависимостей.

Инъекция зависимости (англ. Dependency Injection)

Внедрение зависимостей - это основной компонент разработки через тестирование (англ. test-driven development, TDD), без него невозможно правильно выполнить TDD, потому что без mock-объектов нельзя должным образом отделить разные части кода. Это одно из заблуждений - это проведение интеграционного тестирования, которое соединяет логику с базой данных вместо модульного. Модульное тестирование должно выполняться независимо, и база данных не должна использоваться при проведении модульного тестирования. Однако в Golang зависимость должна быть введена во время компиляции, а не во время выполнения, что несколько отличается от реализации Java / C #, но в любом случае это инъекция зависимостей.

По сути, модульный тест создан для проверки логики, а не целостности данных, и, использование базы данных во время модульного тестирования добавит сложность самим тестам.

Внедрение зависимостей является важной частью правильного TDD. Оно осуществляется с помощью mock-объектов. Итак, как вы видите в нашей структуре проекта, вместо того, чтобы все компоненты напрямую общались друг с другом, мы используем интерфейс, например, UserController.

```
type UserController struct {
  interfaces.IUserService
}
func (controller *UserController) GetUserPoint(res http.ResponseWriter, req *http.Request) {
  user1Name := chi.URLParam(req, "user1")
  user2Name := chi.URLParam(req, "user2")
  points, err := controller.GetPoints(user1Name, user2Name)
  if err != nil {
    //Handle error
```

```
  }
  json.NewEncoder(res).Encode(viewmodels.PointsVM{points})
}
```

UserController использует интерфейс IUserService, а поскольку IUserService имеет метод GetPoints, UserController может вызвать его и сразу получить результат.

```
type IUserService interface {
  GetPoints(user1Name string, user2Name string) (string, error)
}
```

Вместо прямого вызова UserService UserController использует интерфейс UserService, который является реализацией IUserService, может быть много реализаций IUserService, не ограничиваясь только UserService, это может быть BrotherService и другие.

```
func (k *kernel) InjectUserController() controllers.UserController {
  sqlConn, _ := sql.Open("sqlite3", "/var/tmp/tennis.db")
  sqliteHandler := &infrastructures.SQLiteHandler{}
  sqliteHandler.Conn = sqlConn
  userRepository := &repositories.UserRepository{sqliteHandler}
  userService := &services.UserService{&repositories.UserRepositoryWithCircuitBreaker{userRepository}}
  userController := controllers.UserController{userService}
  return userController
}
```

В servicecontainer.go создается userController. Таким образом, в userController IUserService будет внедрен userService вместе со всей реализацией, поэтому, например, GetUserByName теперь возвращает все, что реализовано userService.

```
userService := new(mocks.IUserService)
```

В UserController_test.go используется mock-объект для внедрения реализации сервиса.

Mocking

Mocking — это концепция, понимание которой необходимо для выполнения TDD. Ключевым моментом является имитация зависимостей, необходимых для запуска тестов. Рассмотрим mocking на примере библиотеки testfy. По сути, mock-объект заменяет инъекцию вместо реальной реализации на макет.

```
userService := new(mocks.IUserService)
```

Затем необходимо создать фиктивную функциональность GetPoints для его ответов.

```
userService.On("GetPoints", "Rafael", "Serena").Return("Forty-Fifteen", nil)
```

Далее mock-объект вводится в userService из UserController, поэтому внедрение зависимостей важно для этого процесса, поскольку это единственный способ внедрить интерфейс с mock-объектом вместо реальной реализации.

```
userController: = UserController {userService}
```

Мы генерируем mock-объекта с помощью команды:

```
mockery -name=IUserService
```

Результат будет внутри mocks / IUserService.go, и его можно использовать для тестирования.

Тестирование

Используя внедрение зависимостей и mock-объекты, можно реализовать юнит-тестирование.

```
func TestUserPoint(t *testing.T) {
  // create an instance of our test object
  userService := new(mocks.IUserService)
  // setup expectations
  userService.On("GetPoints", "Rafael", "Serena").Return("Forty-Fifteen", nil)
  userController := UserController{userService}
  // call the code we are testing
  req := httptest.NewRequest("GET", "http://localhost:8080/getPoint/Rafael/vs/Serena", nil)
  w := httptest.NewRecorder()
  r := chi.NewRouter()
  r.HandleFunc("/getPoint/{user1}/vs/{user2}", userController.GetUserPoint)
  r.ServeHTTP(w, req)
  expectedResult := viewmodels.PointsVM{}
  expectedResult.Point = "Forty-Fifteen"
  actualResult := viewmodels.PointsVM{}
  json.NewDecoder(w.Body).Decode(&actualResult)
  // assert that the expectations were met
  assert.Equal(t, expectedResult, actualResult)
}
```

После внедрения userService объекта userController с помощью mock-объекта, вызывается userController.GetUser и имитируется запрос на всем пути от маршрутизатора.

```
req := httptest.NewRequest("GET", "http://localhost:8080/getPoint/Rafael/vs/Serena", nil)
 w := httptest.NewRecorder()

 r := chi.NewRouter()
 r.HandleFunc("/getPoint/{user1}/vs/{user2}", userController.GetUserPoint)

 r.ServeHTTP(w, req)
```

Для проверки результата используется библиотека testify.

```
assert.Equal(t, expectedResult, actualResult)
```

Заключение

При разработке чистой архитектуры Golang приложений важно следовать принципам SOLID. Это поможет повысить надёжность распределённых систем, а также оптимизировать время на доработку системы. Покрытие тестами, используя mock-объекты, способствует исключить ошибки при добавлении бизнес-логики в кодовую базу распределённой системы. Применение чистой архитектуры важно уже на начальном этапе разработки и проектирования системы.

Список литературы / References

1. *Махров А.В.* Чистая архитектура мобильных приложений на платформе Андроид с использование KOTLIN, RXJAVA И DAGGER2 // Актуальные научные исследования в современном мире, 2018. №7-1 (39). С. 22-26.
2. *Альбекова З.М.* Принципы SOLID в ООП // межд. конф. (Пенза, 12 ноября 2019). Пенза: Наука и Просвещение, 2019. С. 51–53.
3. *Коптева А.В., Князев И.В.* Анализ проблемы преобразования данных формата JSON в строго типизированных языках программирования на примере Golang // Актуальные научные исследования в современном мире. Проблемы науки, 2021. № 7 (66). С. 5-10.
4. Документация Golang / [Электронный ресурс], 2021. Режим доступа: https://pkg.go.dev/encoding/json / (дата обращения: 04.09.2021).
5. *Мартин Р.* Чистая архитектура. Искусство разработки программного обеспечения. Санкт-Петербург: Издательский Дом ПИТЕР, 2018. 352 с.
6. *Цукалос М.* Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go. М.: Прогресс книга, 2021. 720 с.
7. *Доновaн Алан А. ., Кернигaн Брайан У.* Язык программирования Go. М.: Вильямс, 2018. 432 с.
8. *Батчер М., Фарина М.* Go на практике. М.: ДМК Пресс, 2017. 376 с.