Гайдлайны

Внутренние гайдлайны разработки

- Общая концепция разработки сервиса
- Обработка ошибок
- Опциональные значения в моделях

Общая концепция разработки сервиса

Общая архитектура

Сервис разрабатывается в слоеной модели. Код содержит три основных слоя в виде пакетов:

- **controller** - слой определяет взаимодействие с сервером fiber:
  - Разбирают запросы пользователей
  - Вызывает функции валидации объектов предметной области
  - Содержит документацию к предоставляемому API
  - Делегирует обработку данных слою **usecase**
- **usecase** - слой бизнес-логики.
  - Содержит типовые кейсы взаимодействия с системой
  - Определяет логику сервера приложений при взаимодействии с системой
  - Делегирует ввод/вывод слою **storage**
- **storage** - слой хранения данных
  - Содержит логику ввода/вывода в конкретные хранилища данных

Отдельно стоит пакет **domain,** который:

- Описывает структуры данных предметной области
- Логику их взаимодействия со всеми слоями системы
- Логику конвертации объектов предметной области в другие объекты предметной области
- Валидацию объектов предметной области

Этот пакет используется всеми компонентами системы.

Обработка ошибок

Общие требования

Для работы с ошибками требуется использовать вместо стандартного пакета **errors**, пакет **github.com/pkg/errors**, так как он позволяет учитывать стэк вызова, при логировании ошибки.

В любых операциях на уровне выше слоя **storage**, нужно использовать внутренний пакет **apierrors**, для формирования ошибок выдаваемых пользователю. Т.е. на слоях **controller** и **usecase**, а так же в пакете domain нужно всегда возвращать соответствующую ошибку из пакета **apierrors**. Если из слоя ниже или другой библиотеки была получена ошибка другого типа, её следует обернуть в ошибку из пакета **apierrors**.

Если соответствующей ошибки нет, ее можно добавить, согласовав назначение нового класса ошибок. Не следует делать специализированные классы ошибок, класс должен отражать широкое множество ошибок схожих по области в которой они возникли. Например, **ValidationError** - класс, отражающий любые ошибки валидации, не следует создавать **MyStructFieldValidationError**, дабы не плодить сущности.

Централизованная обработка ошибок

Условно ошибки можно разделить на два вида:

- Обработанная - ошибка была завернута в **apierror**, значит разработчик предусмотрел её обработку и возврат пользователю.
- Внештатная - любая не обработанная разработчиком ошибка.

Сервис обрабатывает ошибки в едином обработчике, следующим образом:

- Если в обработчик пришла ошибка типа **apierror**, она выводится в лог, если обернутая в нее ошибка поддерживает получение стэка-вызовов - стэк также запишется в лог. Если не поддерживает, в логе будет только сообщение об ошибке. В логе ошибка помечается признаком **severity=handled**.
- Если пришла ошибка веб-сервера fiber - она записывается в лог с **severity=fiber**.

- В остальных случаях, обработчик считает, что ошибка была не обработана и она записывается в лог с **severity=unhandled**.

При любом варианте записи в лог вызывающему отдается ответ с классом ошибки и причиной её возникновения. Ответ выглядит следующим образом:

```
{
  "success": false,
  "message": "Внутренняя ошибка сервиса",
  "error": {
    "class": "internal",
    "type": "runtime.errorString",
    "reason": "runtime error: integer divide by zero"
  }
}
```

Класс ошибки определяет важность для клиентского приложения, сообщение описывает проблему, объект error показывает причину возникновения ошибки.

Соответствие внутренней и внешней классификации ошибок:

	Класс ошибки apierror
handled	Определяется разработчиком
unhandled	internal
fiber	internal

Опциональные значения в моделях

Цель

В некоторых случаях передача всех полей модели от клиента не имеет смысла. Например, клиент хочет обновить только наименование какой-то сущности. Необходимо предоставить простой программный интерфейс для объявления таких полей, получения значений только заполненных полей перед выполнением запроса к базе данных.

Пакет

> gitlab.corp.rarus.cloud/rarus-lms/backend/pkg/optional

Пакет предоставляет обобщенный тип `optional.Optional[T any]`, который оборачивает другой тип и хранит указатель на значение этого типа. Таким образом можно легко различить наличие и отсутствие значения в опциональном поле.

Например, имеем модель клиента, у которого есть необязательное для передачи поле `age`:

```
type Client struct {
	Id   uint64                   `json:"id"`
	Name string                   `json:"name"`
	Age  optional.Optional[uint8] `json:"age"`
}
```

При входном JSON отсутствующим полем age, поле не имеет значения:

```
jsonData := []byte(`{
        "id": 1,
        "name": "Василий",
    }`)
c := Client{}
json.Unmarshal(jsonData, &c) // Обертка использует парсер внутреннего типа
```

```
c.Age.IsSet() // если значение не установлено - false

v, err := c.Age.GetValue() //если значение == nil, возвращает error

c.Age.Value // указатель на внутренний тип, *uint8 в данном случае

v = c.Age.OrDefault(30) // если значение не установлено, вернет значение по-умолчанию
// v == 30, но c.Age.IsSet() == false

c.Age.SetValue(30) // Теперь значение изменилось и изменения будут учтены при получении
map[string]any
```

Для записи в базу не фиксированного набора значений, обычно нужно получить ключи и значения из структуры. Для этого можно использовать функции:

```
jsonData := []byte(`{
"id": 1,
"name": "Василий",
}`)
c := Client{}
json.Unmarshal(jsonData, &c)

m := optional.StructToMap(c) // здесь будет map[string]any, но содержащая только обязательные поля и Optional поля с заполненными значениями
// m = {"id":1, "name":"Василимй"}

k,v := optional.GetKeysAndValues(m) // далее можно получить ключи и значения, для передачи в аргументы драйвера к базе
// k = ["id", "name"]
// v = [1, "Василий"]

squirrel.Update("some_table").Columns(k).Values(v).Where(...)
```

Функция optional.StructToMap формирует ключи хэш-таблицы по следующим правилам:

- Приватные поля всегда игнорируются
- Если у поля не указаны тэги - оно игнорируется
- Если у поля указан тэг `` `db:"some_name"` ``, ключ берется из него
- Если у поля указан тэг `` `json:"some_name"` ``, но нет тэга `db`, ключ берется из него