Пример потока данных в Clean Architecture на Go

Предисловие

Clean Architecture (Чистая Архитектура), предложенная Робертом Мартином (Uncle Bob), — это подход к проектированию ПО, который подчеркивает разделение на слои с четкими зависимостями.

Основная идея: сделать систему независимой от фреймворков, UI, баз данных и внешних сервисов. Зависимости направлены “внутрь” — внешние слои зависят от внутренних, но не наоборот.

Ключевые слои Clean Architecture:

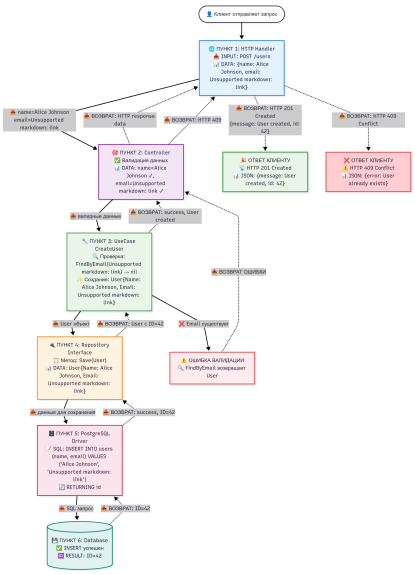
- **Доменный слой (Domain/Entities)**: Содержит бизнес-сущности и бизнес-правила. Это “ядро” системы, не зависящее от внешнего мира.
- **Прикладной слой (Application/Use Cases)**: Описывает сценарии использования — бизнес-логику, которая оркеструет сущности. Зависит от домена, но не от инфраструктуры.
- **Слой адаптеров (Adapters/Controllers, Presenters)**: Обеспечивает взаимодействие с внешним миром (например, HTTP, CLI). Преобразует внешние запросы в вызовы Use Cases.
- **Инфраструктурный слой (Infrastructure/Frameworks & Drivers)**: Реализует детали, такие как доступ к БД, API-клиенты. Зависит от внутренних слоев через интерфейсы (инверсия зависимостей).

Практический пример: Создание пользователя

Сценарий: Пользователь регистрируется, отправляя HTTP POST-запрос на `/users` с JSON-телом:

```
{
  "name": "Alice Johnson",
  "email": "alice@example.com"
}
```

Приложение использует PostgreSQL как БД. Рассмотрим пошаговый поток данных:



Входящий поток: От запроса к сохранению в БД

1. HTTP запрос → Адаптер (HTTP Handler)

Что происходит: Адаптер получает сырой HTTP запрос и парсит его.

- **Входные данные**:
  - Метод: POST
  - Путь: `/users`
  - Тело: `{"name": "Alice Johnson", "email": "alice@example.com"}`
- **Логирование**:

```
[Адаптер] Получен POST-запрос на /users
[Адаптер] Парсинг JSON: name="Alice Johnson", email="alice@example.com"
[Адаптер] Вызываю контроллер с name="Alice Johnson", email="alice@example.com"
```

- **Роль в архитектуре**: Внешний слой адаптеров — точка входа, зависящая от HTTP.

2. Адаптер → Контроллер

Что происходит: Контроллер валидирует данные.

- **Входные данные**: `name = "Alice Johnson"`, `email = "alice@example.com"` (валидные)
- **Логирование**:

```
[Контроллер] Получены данные: name="Alice Johnson", email="alice@example.com"
[Контроллер] Валидация пройдена
[Контроллер] Вызываю UseCase для создания пользователя
```

- **Роль в архитектуре**: Часть прикладного слоя — координирует, но без глубокой логики.

3. Контроллер → UseCase

Что происходит: UseCase применяет бизнес-правила и проверяет уникальность.

- **Входные данные**:
  - Проверка: `FindByEmail("alice@example.com")` → `nil` (не существует)
  - Создание: `User{Name: "Alice Johnson", Email: "alice@example.com"}`
- **Логирование**:

```
[UseCase] Запуск бизнес-логики для создания пользователя: name="Alice Johnson"
[UseCase] Проверка существования: вызываю repo.FindByEmail("alice@example.com")
[UseCase] Пользователь не найден - можно создавать
[UseCase] Создаю модель User: {Name: "Alice Johnson", Email: "alice@example.com"}
[UseCase] Вызываю repo.Save(User)
```

- **Роль в архитектуре**: Доменный/прикладной слой — чистая бизнес-логика, независимая от БД.

4. UseCase → Repository Interface

Что происходит: Абстрактный вызов сохранения.

- **Входные данные**: `User{Name: "Alice Johnson", Email: "alice@example.com"}`
- **Логирование**:

```
[Repository-интерфейс] Вызов Save для User: {Name: "Alice Johnson", Email:
"alice@example.com"}
[Repository-интерфейс] Перенаправляю в реализацию (PostgresUserRepository)
```

- **Роль в архитектуре**: Доменный слой — интерфейс для инверсии зависимостей.

5. Repository Interface → Driver (PostgreSQL)

Что происходит: Формирование и выполнение SQL-запроса.

- **Реальные значения**:

```
INSERT INTO users (name, email)
VALUES ('Alice Johnson', 'alice@example.com')
RETURNING id
```

  Результат: `ID=42`
- **Логирование**:

```
[Драйвер] Получен вызов Save: User {Name: "Alice Johnson", Email: "alice@example.com"}
[Драйвер] Формирую SQL: INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id
[Драйвер] Выполняю запрос с параметрами: $1="Alice Johnson", $2="alice@example.com"
[Драйвер] Результат: Новый ID=42, ошибок нет
```

- **Роль в архитектуре**: Инфраструктурный слой — детали реализации.

6. Database → Успешный ответ

Что происходит: Сохранение в PostgreSQL.

- **Реальные значения**: Вставка записи с `id=42`
- **Логирование**:

```
[БД] Выполнена INSERT: Добавлена запись с id=42, name="Alice Johnson",
email="alice@example.com"
```

- **Роль в архитектуре**: Внешняя зависимость в инфраструктуре.

Визуализация полного цикла на диаграмме

□□ Прямые стрелки (сплошные): Входящий поток

Движение данных ОТ клиента К базе данных

- `→` Обычные стрелки показывают передачу запроса и данных вниз по архитектуре
- На каждой стрелке подписано, какие именно данные передаются

□□ Обратные стрелки (пунктирные): Исходящий поток

Движение результата ОТ базы данных К клиенту

- `-.->` Пунктирные стрелки показывают возврат результата вверх по архитектуре
- На каждой обратной стрелке подписано, какой результат возвращается

???????? ???????????? ???????????

1. **Изоляция слоев**: Смена БД не затрагивает бизнес-логику
2. **Тестируемость**: UseCase можно тестировать с mock-репозиториями
3. **Понятность**: Каждый слой имеет четкую ответственность
4. **Гибкость**: Легко добавлять новые адаптеры (GraphQL, gRPC)

Заключение

Этот пример показывает, как Clean Architecture делает поток данных предсказуемым и устойчивым. Каждый слой выполняет свою роль, а зависимости направлены правильно.

Revision #2
Created 8 September 2025 08:56:42 by Admin
Updated 8 September 2025 09:06:25 by Admin