

Лекция 4.

Архитектура приложений для работы с реляционными СУБД

**Чистый SQL в приложении,
нативные драйверы СУБД**

Чистый SQL, нативные расширения/драйверы

Для отправки SQL запроса используются функции языка программирования

- Код сильно зависит от конкретной СУБД.
- Нет единого интерфейса — каждый драйвер имеет свой синтаксис.
- Сложнее обеспечить безопасность (но всё ещё возможно через экранирование или prepared statements).
- Требуется больше ручной работы по обработке ошибок и соединений.

Тем не менее, такой подход иногда используется в легковесных скриптах или при необходимости максимальной производительности.

Чистый SQL, СУБД SQLite

```
<?php
try {
    // Подключение к базе (файл создаётся автоматически)
    $db = new SQLite3('database.sqlite');

    // Создание таблицы
    $db->exec("CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        email TEXT UNIQUE
    )");

    // Вставка данных с защитой от инъекций
    $stmt = $db->prepare('INSERT INTO users (name, email) VALUES (:name, :email)');
    $stmt->bindValue(':name', 'Анна', SQLITE3_TEXT);
    $stmt->bindValue(':email', 'anna@example.com', SQLITE3_TEXT);
    $stmt->execute();

    // Выборка данных
    $result = $db->query('SELECT * FROM users');
    while ($row = $result->fetchArray(SQLITE3_ASSOC)) {
        echo "ID: {$row['id']}, Имя: {$row['name']}, Email: {$row['email']}\n";
    }

    $db->close();
} catch (Exception $e) {
    echo "Ошибка: " . $e->getMessage();
}
?>
```



ЧИСТЫЙ SQL, СУБД MySQL

```
<?php
$host = 'localhost';
$user = 'root';
$pass = '';
$dbname = 'testdb';

// Подключение
$mysqli = new mysqli($host, $user, $pass, $dbname);

// Проверка соединения
if ($mysqli->connect_error) {
    die("Ошибка подключения: " . $mysqli->connect_error);
}

// Создание таблицы
$mysqli->query("CREATE TABLE IF NOT EXISTS products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10,2)
)");

// Подготовленный запрос (защита от инъекций)
$stmt = $mysqli->prepare("INSERT INTO products (name, price) VALUES (?, ?)");
$name = "Смартфон";
$price = 39990.00;
$stmt->bind_param("sd", $name, $price); // s = string, d = double
$stmt->execute();
```

Чистый SQL, СУБД MySQL

```
// Выборка данных
$result = $mysqli->query("SELECT * FROM products");
while ($row = $result->fetch_assoc()) {
    echo "ID: {$row['id']}, Название: {$row['name']}, Цена: {$row['price']}\n";
}

$mysqli->close();
?>
```

Чистый SQL, СУБД PostgreSQL

```
<?php
$host = 'localhost';
$port = '5432';
$dbname = 'mydb';
$user = 'myuser';
$password = 'mypass';

// Строка подключения
$conn_string = "host=$host port=$port dbname=$dbname user=$user password=$password";

// Подключение
$conn = pg_connect($conn_string);
if (!$conn) {
    die("Не удалось подключиться к PostgreSQL: " . pg_last_error());
}

// Создание таблицы
pg_query($conn, "CREATE TABLE IF NOT EXISTS orders (
    id SERIAL PRIMARY KEY,
    description TEXT NOT NULL,
    amount NUMERIC(10,2)
)");

// Вставка с защитой от инъекций через pg_query_params()
pg_query_params($conn,
    "INSERT INTO orders (description, amount) VALUES ($1, $2)",
    ['Покупка книг', 1250.75]
);
```

Чистый SQL, СУБД PostgreSQL

```
// Выборка данных
$result = pg_query($conn, "SELECT * FROM orders");
while ($row = pg_fetch_assoc($result)) {
    echo "ID: {$row['id']}, Описание: {$row['description']}, Сумма: {$row['amount']}\n";
}

pg_close($conn);
?>
```

Чистый SQL и нативные драйверы, проблемы

1. Разный API для каждой СУБД

Код для MySQL (`mysql`) не работает с PostgreSQL (`pg_*`) без полной переписки.

2. Сложность поддержки нескольких СУБД

Если вы захотите перейти с MySQL на PostgreSQL — почти весь код придётся переписывать.

3. Ограниченная переносимость SQL

Например:

- `AUTO_INCREMENT` (MySQL) ≠ `SERIAL` (PostgreSQL) ≠ `AUTOINCREMENT` (SQLite)
- Разные функции: `NOW()` (MySQL) vs `CURRENT_TIMESTAMP` (PostgreSQL)

Чистый SQL и нативные драйверы, проблемы

4. Риск SQL-инъекций при неправильном экранировании

Если вы используете `pg_escape_string()` или `mysqli_real_escape_string()` — легко ошибиться. Лучше всегда использовать подготовленные запросы (`prepare` / `pg_query_params`).

5. Отсутствие единой обработки ошибок

У `mysqli` — свой способ, у `pg_*` — другой, у `SQLite3` — исключения или `lastErrorMsg()` .

6. Сложность тестирования и мокирования

Нативные функции сложнее замокать в unit-тестах по сравнению с интерфейсами PDO.

7. Меньше документации и примеров

PDO — стандарт де-факто, поэтому большинство современных руководств используют его.

**SQL, абстрагированный от
конкретной СУБД**

PDO в PHP

PDO (PHP Data Objects) — это **встроенный в PHP интерфейс (расширение)** для доступа к базам данных. Он предоставляет **единый и унифицированный API** для взаимодействия с различными СУБД (SQLite, MySQL, PostgreSQL, Oracle, SQL Server и др.), позволяя писать более переносимый, безопасный и поддерживаемый код.

PDO — это **абстрактный слой доступа к данным**, а не полноценная ORM. Он:

- Не генерирует SQL за вас.
- Позволяет писать **чистый SQL**, но без привязки к конкретной СУБД (в разумных пределах).
- Обеспечивает **единый способ подключения, выполнения запросов и обработки ошибок** независимо от того, какая база данных используется.

Подключение к разным СУБД через PDO

```
// SQLite
$db = new PDO('sqlite:app.db');

// MySQL
$db = new PDO('mysql:host=localhost;dbname=test', $user, $pass);

// PostgreSQL
$db = new PDO('pgsql:host=localhost;dbname=test', $user, $pass);
```

Безопасный запрос через PDO

```
php
1 $pdo = new PDO('mysql:host=localhost;dbname=shop', $user, $pass);
2 $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
3
4 // Подготовленный запрос – безопасен от инъекций
5 $stmt = $pdo->prepare("SELECT * FROM products WHERE price < :max_price");
6 $stmt->execute(['max_price' => 1000]);
7
8 while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
9     echo $row['name'] . "\n";
10 }
```

Даже если `$max_price` пришёл из `$_GET`, он никогда не будет интерпретирован как SQL-код.

Решение проблем нативных драйверов

	SQL и нативные драйверы СУБД	PDO
Разный API для каждой СУБД, сложность переключения между СУБД	Полное переписывание кода	Единый интерфейс, меняется только строка подключения
Обработка ошибок	Разные механизмы (mysqli – через свойства, pg_ – через функции, SQLite3 – через исключения)	Единый механизм (три режима: молчаливый, предупреждения, исключения)
Риск SQL-инъекций	Экранирование запросов вручную	Параметризованные запросы с автоматическим экранированием

Решение проблем нативных драйверов

	SQL и нативные драйверы СУБД	PDO
Сложность тестирования	Нативные функции тяжело заменять моками в юнит-тестах	Можно передавать \$pdo как зависимость и подменять этот объект при тестировании
Поддержка транзакций	Отсутствие поддержки транзакций в едином стиле	Унифицированное управление транзакциями
Совместимость с современными практиками	Нет	Стандарт, используется внутри многих фреймворков

Плохой код на чистом SQL

```
// ПЛОХОЙ ПРИМЕР – ТОЛЬКО ДЛЯ СРАВНЕНИЯ!  
$email = $_GET['email']; // например: 'alice@example.  
  
// Подключение (в старом стиле)  
$db = new SQLite3('blog.db');  
  
// Опасный запрос с конкатенацией!  
$query = "SELECT p.title  
        FROM posts p  
        JOIN authors a ON p.author_id = a.id  
        WHERE a.email = '$email'";  
  
$result = $db->query($query);  
  
while ($row = $result->fetchArray(SQLITE3_ASSOC)) {  
    echo htmlspecialchars($row['title']) . "<br>";  
}
```

Безопасный и унифицированный код на PDO

```
try {
    $pdo = new PDO('sqlite:blog.db');
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    $email = $_GET['email'];

    $stmt = $pdo->prepare("
        SELECT p.title
        FROM posts p
        JOIN authors a ON p.author_id = a.id
        WHERE a.email = ?
    ");

    $stmt->execute([$email]);

    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        echo htmlspecialchars($row['title']) . "<br>";
    }

} catch (PDOException $e) {
    die("Ошибка БД: " . $e->getMessage());
}
```

PDO решает ключевую проблему: унификацию доступа к разным базам данных в PHP.

Он:

- Повышает безопасность (через prepared statements),
- Упрощает смену СУБД,
- Делает код чище и современнее.

Единый интерфейс для доступа к разным СУБД

ЯЗЫК	АНАЛОГ / ПОДХОД
Python	DB-API 2.0 (например, <code>sqlite3</code> , <code>psycopg2</code> , <code>PyMySQL</code>) + SQLAlchemy Core
Java	JDBC (Java Database Connectivity)
C# / .NET	ADO.NET + DbProviderFactory
Go	<code>database/sql</code> пакет + драйверы (например, <code>go-sql-driver/mysql</code>)
Ruby	<code>DBI</code> (устарел) или <code>ActiveRecord</code> / <code>Sequel</code>
Node.js	Нет единого стандарта, но есть <code>knex.js</code> или <code>typeorm</code> с мульти-БД поддержкой

Дублирование кода (boilerplate) при работе с PDO

Boilerplate при работе с PDO

Каждый раз писать:

php

```
1 $stmt = $pdo->prepare("SELECT name FROM users WHERE id = ?");
2 $stmt->execute([$id]);
3 $row = $stmt->fetch();
4 $name = $row['name'] ?? null;
```

— утомительно, громоздко и снижает читаемость.

Решение — обернуть типовые операции в вспомогательные функции или класс-обёртку. Такие функции, как `getCell()`, `getRow()`, `getAll()`, действительно существенно упрощают код и делают его лаконичным.

Вспомогательные функции

```
<?php

function getCell(PDO $pdo, string $sql, array $params = []): mixed
{
    $stmt = $pdo->prepare($sql);
    $stmt->execute($params);
    return $stmt->fetchColumn();
}

function getRow(PDO $pdo, string $sql, array $params = []): ?array
{
    $stmt = $pdo->prepare($sql);
    $stmt->execute($params);
    return $stmt->fetch(PDO::FETCH_ASSOC) ?: null;
}

function getAll(PDO $pdo, string $sql, array $params = []): array
{
    $stmt = $pdo->prepare($sql);
    $stmt->execute($params);
    return $stmt->fetchAll(PDO::FETCH_ASSOC);
}
```

Вспомогательные функции - использование

```
// Получить одно значение
$email = getCell($pdo, "SELECT email FROM users WHERE id = ?", [42]);

// Получить одну строку
$user = getRow($pdo, "SELECT * FROM users WHERE id = ?", [42]);
if ($user) {
    echo "Привет, {$user['name']}!";
}

// Получить все записи
$products = getAll($pdo, "SELECT * FROM products WHERE active = 1");
```

Эти функции полностью безопасны от SQL-инъекций, потому что:

- Используют подготовленные запросы (`prepare` + `execute`).
- Параметры передаются отдельно от SQL-строки.

⚠ Но: SQL-запрос (`$sql`) должен быть статическим или полностью контролироваться разработчиком. Никогда не подставляйте в `$sql` данные от пользователя!

Класс-обертка для PDO

```
class Db
{
    private PDO $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function cell(string $sql, array $params = []): mixed
    {
        $stmt = $this->pdo->prepare($sql);
        $stmt->execute($params);
        return $stmt->fetchColumn();
    }

    public function row(string $sql, array $params = []): ?array
    {
        $stmt = $this->pdo->prepare($sql);
        $stmt->execute($params);
        return $stmt->fetch(PDO::FETCH_ASSOC) ?: null;
    }
}
```

Класс-обертка для PDO

```
public function all(string $sql, array $params = []): array
{
    $stmt = $this->pdo->prepare($sql);
    $stmt->execute($params);
    return $stmt->fetchAll(PDO::FETCH_ASSOC);
}

public function exec(string $sql, array $params = []): int
{
    $stmt = $this->pdo->prepare($sql);
    $stmt->execute($params);
    return $stmt->rowCount();
}
}
```

Использование:

```
php
1 $db = new Db($pdo);
2
3 $name = $db->cell("SELECT name FROM users WHERE id = ?", [1]);
4 $user = $db->row("SELECT * FROM users WHERE email = ?", ['test@example.com']);
5 $activeUsers = $db->all("SELECT * FROM users WHERE active = 1");
6 $db->exec("UPDATE users SET last_login = NOW() WHERE id = ?", [1]);
```

Преимущества такого подхода

ПРЕИМУЩЕСТВО	ОПИСАНИЕ
✓ Меньше boilerplate	Нет повторяющегося кода <code>prepare → execute → fetch</code> .
✓ Читаемость	Логика запроса видна сразу, без шума.
✓ Безопасность	Все запросы параметризованы.
✓ Лёгкость тестирования	Можно мокать <code>Db</code> или передавать <code>\$pdo</code> через DI.
✓ Гибкость	Легко добавить логирование, кэширование, профилирование.

Data Access Layer (DAL)

DAL как архитектурный принцип

DAL (Data Access Layer) — это уровень (слой) приложения, отвечающий исключительно за взаимодействие с источниками данных (обычно базами данных). Его цель — инкапсулировать всю логику доступа к данным, чтобы остальная часть приложения (бизнес-логика, контроллеры и т.д.) не зависела напрямую от СУБД, SQL или способа подключения.

Реализация DAL через вспомогательные функции

DAL через вспомогательные функции

“Дополнительные (вспомогательные) функции — это самый простой процедурный способ реализации DAL (Data Access Layer).”

◆ Почему это именно DAL?

Потому что такие функции:

- **Инкапсулируют доступ к данным** (скрывают детали подключения, подготовки запросов, обработки результата).
- **Отделяют логику работы с БД от остального кода** (бизнес-логики, контроллеров и т.д.).
- **Предоставляют чистый интерфейс** вроде `getUserById(42)` вместо «сырого» SQL.

Это и есть суть DAL — **абстрагировать источник данных**, независимо от уровня сложности реализации.

DAL через вспомогательные функции

◆ Когда это уместно?

- Небольшие сайты, скрипты, прототипы.
 - Микросервисы с 2–5 таблицами.
 - Образовательные проекты.
 - Легаси-код, куда нельзя внедрить ORM.
-

◆ Ограничения

- Не масштабируется на сотни сущностей.
- Нет типизации (возвращаются массивы, а не объекты).
- Сложно поддерживать сложные связи (JOIN, вложенные сущности).
- Трудно применять единые политики (кеширование, логирование) без рефакторинга.

Но для своего класса задач — это идеальное решение.

Реализация DAL через паттерн Repository

DAL через репозиторий

“DAL через реализацию репозитория — это следующий, более зрелый шаг в абстрагировании данных по сравнению с процедурным подходом (вспомогательными функциями).”

Он сохраняет все преимущества простого DAL, но добавляет **структуру, масштабируемость и соответствие принципам объектно-ориентированного проектирования.**

От функций к репозиторию

Шаг 1: Процедурный DAL

php

```
1 $user = db_get_row($pdo, "SELECT * FROM users WHERE id = ?", [$id]);
```

Шаг 2: Репозиторий

php

```
1 interface UserRepositoryInterface
2 {
3     public function findById(int $id): ?User;
4 }
5
6 class PdoUserRepository implements UserRepositoryInterface
7 {
8     public function __construct(private PDO $pdo) {}
9
10    public function findById(int $id): ?User
11    {
12        $stmt = $this->pdo->prepare("SELECT * FROM users WHERE id = ?");
13        $stmt->execute([$id]);
14        $row = $stmt->fetch();
15        return $row ? new User($row['id'], $row['name']) : null;
16    }
17 }
```

DAL посредством репозитория

Использование:

```
php
1 // Бизнес-логика зависит от интерфейса, а не от PDO
2 function sendWelcomeEmail(UserRepositoryInterface $users, int $id): void
3 {
4     $user = $users->findById($id);
5     if ($user) {
6         mail($user->email, "Добро пожаловать!", "..."); // предположим, email есть
7     }
8 }
```

👉 Теперь бизнес-логика полностью не знает о существовании SQL, PDO или даже базы данных.

Преимущества репозитория

1. Чёткая граница ответственности

Репозиторий знает только как сохранять/загружать сущность. Больше ничего.

2. Поддержка разных источников данных

Можно реализовать:

- `PdoUserRepository` — из базы,
- `ApiUserRepository` — из REST API,
- `InMemoryUserRepository` — для тестов.

А бизнес-логика не изменится.

3. Лёгкое внедрение зависимостей (DI)

Особенно важно в фреймворках (Symfony, Laravel) и при тестировании.

Преимущества репозитория

4. Типобезопасность

Возвращаются не ассоциативные массивы, а **объекты с чёткими свойствами**.

5. Подготовка к сложным сценариям

Позже можно добавить:

- кэширование в репозиторий,
- пагинацию,
- спецификации (Specification pattern),
- события (pre-save, post-load).

Когда переходить к репозиторию?

- У вас больше 2–3 сущностей.
- Вы пишете тесты.
- Вы чувствуете, что функции `db_get_row()` размножаются и дублируются.
- Вы хотите отделить «как хранить» от «что делать».

“💡 Репозиторий — это «золотая середина»: достаточно просто для понимания, но достаточно мощно для роста проекта.”

Семантический разрыв между реляционными БД и ООП

Фундаментальная проблема

Семантический разрыв — это несоответствие между:

- моделью данных в ООП-коде (объекты, классы, наследование, поведение),
- и моделью данных в реляционной БД (таблицы, строки, столбцы, внешние ключи).

Другими словами:

| *“Объекты — это не таблицы, а таблицы — это не объекты.”*

Это различие в парадигмах порождает сложности при сохранении и загрузке данных.

Фундаментальная проблема

В ООП-коде (PHP):

```
php
1 class Author {
2     public string $name;
3     public array $books; // массив объектов Book
4 }
5
6 class Book {
7     public string $title;
8     public Author $author; // ссылка на объект
9 }
```

Здесь:

- `Author` и `Book` — полноценные объекты.
- Между ними есть связь «один ко многим» через свойства.
- Объекты могут иметь методы, инкапсуляцию, наследование.

Фундаментальная проблема

В реляционной БД (SQLite):

```
sql
1 v CREATE TABLE authors (
2     id INTEGER PRIMARY KEY,
3     name TEXT
4 );
5
6 v CREATE TABLE books (
7     id INTEGER PRIMARY KEY,
8     title TEXT,
9     author_id INTEGER,
10    FOREIGN KEY (author_id) REFERENCES authors(id)
11 );
```

Здесь:

- Нет объектов — только плоские таблицы.
- Связь реализована через внешний ключ (`author_id`).
- Нет поведения — только данные.

Проблемы, вызванные семантическим разрывом

- **Ручное преобразование объект ↔ строка.** При каждом сохранении/загрузке нужно вручную маппить поля объекта на столбцы таблицы и наоборот.
- **Потеря поведения.** Методы объекта (например, `getFullName()`) не хранятся в БД — их нужно воссоздавать после загрузки.
- **Сложность с наследованием.** Как хранить иерархию классов (например, `User→Admin, Guest`) в таблицах? Требуются специальные стратегии (одна таблица на иерархию, одна на класс и т.д.).
- **Жёсткая привязка к структуре БД.** Изменение схемы БД (например, переименование столбца) ломает код, если маппинг не абстрагирован.

Object-Relational Mapping (ORM)

ORM

Семантический разрыв — это не просто техническая деталь, а **глубокое концептуальное различие** между двумя моделями данных:

- ООП: объекты с состоянием и поведением,
- Реляционная модель: наборы кортежей без поведения.

ORM — это инструмент, который минимизирует этот разрыв, позволяя писать чистый, объектно-ориентированный код, не думая постоянно о таблицах и SQL.

ORM как инструмент устранения разрыва

ORM (Object-Relational Mapper) выступает в роли «переводчика» между двумя мирами:

1. Автоматическое маппинг

Класс → таблица, свойство → столбец, объект → строка.

2. Поддержка связей

ORM автоматически загружает связанные объекты (например, `$author->books`).

3. Абстракция от SQL

Разработчик работает с объектами, а ORM генерирует нужные SQL-запросы.

4. Поддержка наследования (в продвинутых ORM, например Doctrine)

ORM может реализовать стратегии хранения иерархий.

ORM как инструмент устранения разрыва

Пример с Eloquent (Laravel):

```
php
1 $author = Author::with('books')->find(1); // ORM сам делает JOIN
2 echo $author->name;
3 foreach ($author->books as $book) {
4     echo $book->title; // $book – полноценный объект Book
5 }
```

Здесь семантический разрыв скрыт: вы работаете с объектами, как будто они «живут» в памяти, хотя на самом деле данные приходят из таблиц.

1. Лёгкие (Thin / Lightweight) ORM

Цель: упростить выполнение SQL-запросов и маппинг результатов на объекты, не скрывая SQL полностью.

Характеристики:

- Минимальная абстракция над SQL.
- Разработчик по-прежнему пишет запросы (часто в виде строк или через query builder).
- Автоматическое преобразование строк БД → объекты (и обратно).
- Нет сложной логики отслеживания изменений (change tracking).

Примеры:

- RedBeanPHP (PHP)
- SQLAlchemy Core (Python)
- Dapper (.NET)

2. Полнофункциональные (Full-featured / Heavyweight) ORM

Цель: полностью абстрагировать разработчика от SQL и реляционной модели

Характеристики:

- Работа только с объектами.
- Автоматическая генерация SQL.
- Поддержка сложных связей (один-ко-многим, многие-ко-многим).
- **Отслеживание изменений** (change tracking): ORM сам определяет, какие объекты изменились, и сохраняет их.
- Поддержка наследования, кэширования, ленивой загрузки (lazy loading)
- Часто требует конфигурации моделей (аннотации, YAML, XML).

Примеры:

- Doctrine ORM (PHP)
- Hibernate (Java)
- Entity Framework (.NET)
- SQLAlchemy ORM (Python)

3. Генеративные ORM (Code-Generating ORM)

Цель: сгенерировать классы моделей на основе схемы БД (или наоборот).

Характеристики:

- Используют инструменты генерации кода.
- Часто применяются в enterprise-средах.
- Модели — не пишутся вручную, а создаются автоматически.

Примеры:

- **Propel** (PHP) — генерирует классы из XML-схемы.
- **MyBatis Generator** (Java)
- **jOOQ** (Java) — генерирует типобезопасные классы на основе БД.

4. Active Record

Цель: объединить **данные и поведение** в одном классе: объект сам знает, как себя сохранить, обновить или удалить.

Характеристики:

- Каждый объект модели — это одновременно **данные + методы работы с БД** (`save()` , `delete()` , `find()`).
- Простота и интуитивность.
- Часто используется в веб-фреймворках.

Примеры:

- Eloquent ORM (Laravel, PHP)
- Active Record (Ruby on Rails)
- CakePHP ORM



5. Data Mapper (противоположность Active Record)

Цель: строго разделить доменные объекты и логику доступа к данным.

Характеристики:

- Доменные объекты — «глупые» (POPO / ПОСО): только данные, без методов БД.
- Отдельные классы-репозитории или мапперы отвечают за сохранение/загрузку.
- Соответствует принципам чистой архитектуры и DDD (Domain-Driven Design).

Примеры:

- Doctrine ORM (PHP, по умолчанию использует Data Mapper)
- Hibernate (Java)

Основные виды ORM

Тип ORM	Уровень абстракции	Контроль над SQL	Сферы применения	Примеры
Active Record	Средний	Низкий	Простые веб-приложения	Eloquent, Rails AR
Data Mapper	Высокий	Очень низкий	Сложные системы	Doctrine, Hibernate
Легкий ORM	Низкий	Высокий	Производительность	RedBean PHP, Dapper
Генеративный ORM	Средний и высокий	Средний	Enterprise, типизация	Propel, jOOQ

Реализация ORM через паттерн Active Record

Active Record как форма ORM

Active Record — это шаблон проектирования (паттерн), впервые описанный Мартином Фаулером в книге «*Patterns of Enterprise Application Architecture*».

“Суть паттерна:

Объект, который обёртывает строку таблицы базы данных, инкапсулирует как данные, так и поведение для доступа к этой строке — включая методы для сохранения, обновления, удаления и поиска записей.”

Другими словами:

“Каждый объект модели — это одновременно и данные, и «дверь» в базу данных.”

Active Record как форма ORM

sql



```
1 CREATE TABLE users (  
2     id INTEGER PRIMARY KEY,  
3     name TEXT,  
4     email TEXT  
5 );
```

Реализация через Active Record (на примере Eloquent из Laravel):

php



```
1 // Модель User – наследуется от базового класса Eloquent\Model  
2 class User extends Illuminate\Database\Eloquent\Model  
3 {  
4     protected $table = 'users';  
5     // $fillable, $guarded и т.д. – опционально  
6 }  
7  
8 // Использование:  
9 $user = new User();  
10 $user->name = 'Алиса';  
11 $user->email = 'alice@example.com';  
12 $user->save(); // ← объект сам сохраняет себя в БД!
```

Active Record как форма ORM

```
14 // Поиск:
15 $user = User::find(1); // ← статический метод поиска
16 echo $user->name;
17
18 // Обновление:
19 $user->name = 'Алиса Новая';
20 $user->save(); // снова – сам себя обновляет
21
22 // Удаление:
23 $user->delete();
```

Здесь:

- Класс `User` — это **Active Record**.
- Он знает, в какой таблице хранится (`users`).
- Он умеет сохраняться, обновляться, удаляться.
- Он предоставляет статические методы для запросов (`find`, `where`, и т.д.).

Преимущества Active Record

1. Простота и интуитивность

Новичок быстро понимает: «создал объект → заполнил → сохранил».

2. Минимум кода

Не нужны отдельные репозитории или мапперы.

3. Быстрая разработка

Идеален для CRUD-приложений (блоги, админки, MVP).

4. Хорошо интегрирован во фреймворки

Например, Laravel (Eloquent), Ruby on Rails, Yii, CakePHP.

Недостатки Active Record

1. Нарушение принципа единственной ответственности (SRP)

Объект отвечает и за данные, и за сохранение — две разные зоны ответственности.

2. Жёсткая связь с БД

Трудно протестировать логику без подключения к базе (требуется мокать всю ORM).

3. Сложность при сложной бизнес-логике

Если у вас DDD (Domain-Driven Design) или сложные агрегаты — Active Record мешает.

4. Проблемы с наследованием и полиморфизмом

Не все реализации хорошо поддерживают наследование моделей.

ORM RedBean PHP

Что это такое

RedBeanPHP — это легковесная, динамическая и простая в использовании **ORM (Object-Relational Mapping)** библиотека для языка PHP. Её главная особенность — **автоматическое создание и адаптация структуры базы данных под ваш код**, без необходимости писать миграции, схемы или конфигурационные файлы.

RedBeanPHP следует принципу "NoSQL-стиля для SQL-баз": вы пишете код, а RedBean автоматически создаёт таблицы, столбцы и связи в базе данных **на лету**, в зависимости от того, какие свойства вы присваиваете объектам.

Это особенно удобно для:

- быстрого прототипирования,
- обучения,
- небольших проектов,
- случаев, когда структура данных часто меняется.

1. Установка

Через Composer:

```
bash
```

```
1 composer require gabordemoij/redbean-php
```

Или просто скачайте один файл `rb.php` с [официального сайта](#).

2. Подключение к базе

```
php
```

```
1 require 'rb.php';  
2  
3 R::setup('mysql:host=localhost;dbname=mydb', 'user', 'password');  
4 // Для SQLite: R::setup('sqlite:/tmp/db.sqlite');
```

3. Создание записи

```
php
1 // Автоматически создаётся таблица 'book', если её нет
2 $book = R::dispense('book');
3 $book->title = '1984';
4 $book->author = 'George Orwell';
5 $book->price = 12.99;
6
7 $id = R::store($book); // Сохраняет в БД и возвращает ID
```

“RedBean создаст таблицу `book` со столбцами `id`, `title`, `author`, `price` автоматически.”

4. Чтение записи

```
php
1 $book = R::load('book', $id);
2 echo $book->title; // "1989"
```

5. Поиск записей

```
php 📄 ⬇  
1 $books = R::find('book', 'price > ?', [10]);  
2 foreach ($books as $b) {  
3     echo $b->title . "\n";  
4 }
```

Или с сортировкой:

```
php 📄 ⬇  
1 $books = R::findAll('book', 'ORDER BY title ASC');
```

6. Связи между таблицами (One-to-Many)

```
php
1 $author = R::dispense('author');
2 $author->name = 'J.K. Rowling';
3
4 $book1 = R::dispense('book');
5 $book1->title = 'Harry Potter 1';
6
7 $book2 = R::dispense('book');
8 $book2->title = 'Harry Potter 2';
9
10 // Связываем книги с автором
11 $author->ownBookList[] = $book1;
12 $author->ownBookList[] = $book2;
13
14 R::store($author); // Сохранит автора и книги, добавит author_id
```

“RedBean автоматически добавит столбец `author_id` в таблицу `book`.”

7. Many-to-Many связи

```
php
1 $tag1 = R::dispense('tag');
2 $tag1->name = 'fantasy';
3
4 $tag2 = R::dispense('tag');
5 $tag2->name = 'adventure';
6
7 $book->sharedTagList[] = $tag1;
8 $book->sharedTagList[] = $tag2;
9
10 R::store($book);
```

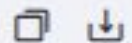
“RedBean создаст промежуточную таблицу `book_tag` автоматически.”

Режимы работы

RedBean имеет два основных режима:

- **Режим разработки (по умолчанию)** — автоматически изменяет структуру БД.
- **Режим продакшена** — фиксирует схему и запрещает изменения:

```
php
```



```
1 R::freeze(); // После этого RedBean не будет менять структуру БД
```

Достоинства RedBean PHP

- Нет необходимости писать миграции.
- Минимум кода для CRUD-операций.
- Отлично подходит для прототипов и MVP.
- Всё в одном файле (можно использовать без Composer).
- Поддержка транзакций, индексов, внешних ключей (вручную при необходимости).

Слабые места RedBean PHP

- Не подходит для сложных enterprise-приложений.
- Менее гибкий, чем Doctrine или Eloquent.
- Автоматические изменения схемы могут быть опасны в продакшене (если не вызвать `freeze()`).
- Меньше контроля над SQL-запросами.