

Технологии программирования

Лекция 9 | API и архитектура

Архитектура программного обеспечения (ПО) –

Это план или структура, описывающая, как система организована, как разные ее части (компоненты) взаимодействуют между собой и с внешними системами

Виды архитектуры:

Микросервисы

Микроядро

Клиент-Серверная

Domain-Driven Design

Event-Driven

Монолит

Модульная





Модульная архитектура

Преимущества и принципы

API

Определение и виды

Интеграция внешних библиотек

Пример для CMake

Как делиться кодом

Общие шаги

REST API

Базовые концепции



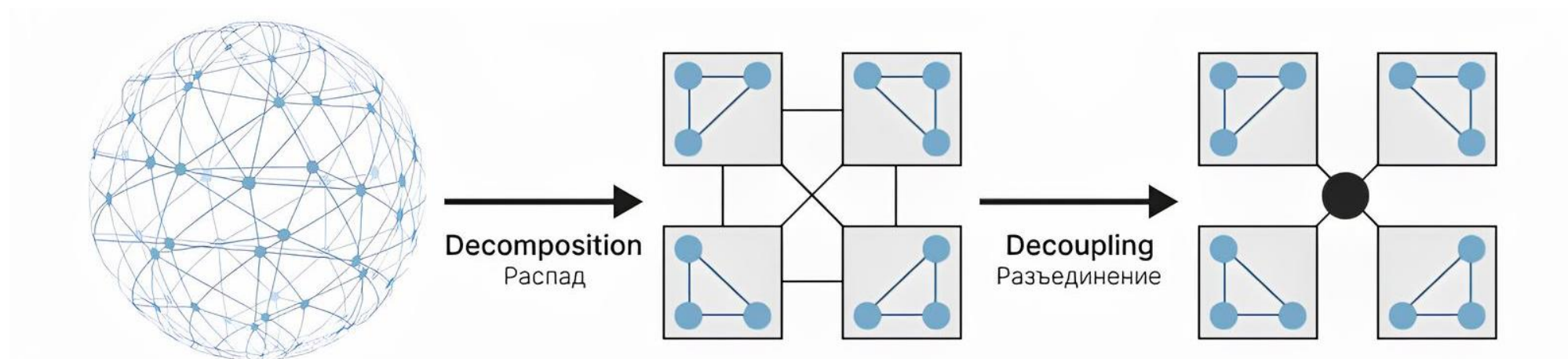
Мы здесь



Модульное приложение

Это подход к разработке программного обеспечения, в котором приложение разбивается на независимые компоненты – **модули**

Каждый модуль выполняет определенную функцию и имеет свою логику и интерфейс для взаимодействия с другими модулями



Преимущества модульной архитектуры

Возможность разработки в большой команде

Модульная архитектура позволяет разработчикам пользоваться разными модулями и работать над ними независимо друг от друга

Возможность повторного использования кода

Код, написанный для одного модуля, может быть использован в другом модуле – это позволяет использовать код повторно и уменьшает количество его дублирования

Усовершенствованная отладка и тестирование

Модули легко тестировать и запускать независимо друг от друга, чтобы существенно упрощает и ускоряет разработку

Лучшая масштабируемость

При данной архитектуре легко добавить новую функциональность в отдельный модуль, что упрощает расширение всего приложения

Что напоминает?

Что напоминает?



Идеи ООП в данной концепции

Модули можно воспринимать как отдельные пакеты, а классы – как библиотеки

ООП	Модули
Инкапсуляция	
Соккрытие данных внутри класса	Изоляция логики внутри компонента (например, API модуля)
Полиморфизм	
Интерфейсы позволяют заменять реализации объектов	API позволяет независимо развивать проекты, подменяя модули при необходимости
Наследование	
Повторное использование кода через иерархию	Модули собираются в систему, как конструктор
Жесткая связь, сложность изменения родительских классов	

Как добиться модульной архитектуры?

Определение границ модуля

Определение зависимостей

Использование интерфейсов

Инъекция зависимостей

Использование общего ядра

Определение границ модуля

! Каждый модуль должен иметь четко обозначенную границу, в пределах которой он выполняет свою функцию

```
// Модуль "матрицы" (math/matrix.h)
namespace math::matrix {
    class Matrix {
    public:
        Matrix multiply(const Matrix& other) const;
    };
}

// Модуль "оптимизация" (math/optimization.h)
namespace math::optimization {
    class GradientDescent {
    public:
        void minimize();
    };
}
```

Определение зависимостей

! Модуль должен зависеть только от необходимых компонентов

```
// Модуль "solver" зависит только от "matrix",  
а не от "visualization"  
#include "math/matrix.h"  
  
namespace math::solver {  
    class LinearSolver {  
    public:  
        void solve(const math::matrix::Matrix& A) { ... }  
    };  
}
```

```
// Модуль "solver" использует только "matrix"  
use crate::matrix::Matrix;  
  
pub struct LinearSolver;  
impl LinearSolver {  
    pub fn solve(a: &Matrix) { ... }  
}
```

Использование интерфейсов

! Взаимодействие через абстракции, а не конкретные реализации

```
// Интерфейс "ISolver"
class ISolver {
public:
    virtual void solve() = 0;
};

// Модуль "newton_solver" реализует
интерфейс
class NewtonSolver : public ISolver {
public:
    void solve() override { ... }
};
```

```
// Трейт "Solver"
pub trait Solver {
    fn solve(&self);
}

// Модуль "newton_solver" реализует трейт
pub struct NewtonSolver;
impl Solver for NewtonSolver {
    fn solve(&self) { ... }
}
```

Инъекция зависимостей

! Зависимости передаются извне, а не создаются внутри модуля

```
// Общий интерфейс логгера
type Logger interface {
    Log(message string)
}

// Модуль "solver" принимает логгер через конструктор
type Solver struct {
    logger Logger
}

func NewSolver(logger Logger) *Solver {
    return &Solver{logger: logger}
}

func (s *Solver) Solve() { s.logger.Log("Solving...") }
```

```
# Модуль "optimizer.py"
class Optimizer:
    def __init__(self, logger):
        self.logger = logger

    def run(self):
        self.logger.log("Optimization started")

# Использование
logger = ConsoleLogger()
optimizer = Optimizer(logger)
optimizer.run()
```

Использование общего ядра

! Общие утилиты выносятся в отдельный модуль

```
// core/logging.h
namespace core::logging {
    class Logger {
    public:
        void log(const std::string& message);
    };
}

// Модуль "solver" использует общий логгер
#include "core/logging.h"

namespace math::solver {
    class LinearSolver {
    private:
        core::logging::Logger logger;
    public:
        void solve() {
            logger.log("Solving linear system...");
        }
    };
}
```

```
// src/utils/logging.rs
pub struct Logger;
impl Logger {
    pub fn log(&self, message: &str) { ... }
}
```

// Модуль "solver" использует общий логгер
use crate::utils::logging::Logger;

```
pub struct Solver {
    logger: Logger,
}
```

Организация проекта

API

Описывает и определяет всю функциональность и сервисы, которые предоставляет модуль

Core

Обеспечивает базовую функциональность, которая редко меняется
Как правило, модули core не зависят от конкретного проекта
Зачастую используются внутри API проекта

Utility

Обеспечивает целенаправленную, многократно используемую функциональность, которая используется в небольшом количестве связанных модулей Implementation

Implementation

Содержит классы, необходимые для реализации функций и сервисов, описанных в модулях API
Обычно модули Implementation содержат одну или несколько общедоступных точек входа

Организация проекта

Тип	Функция	Зависимости
API	Описывает что делает система (контракты, интерфейсы)	Может зависеть от Core
Core	Базовая функциональность (абстракции, редко меняется)	Независим от других типов
Utility	Утилиты для Implementation (повторно используемая логика)	Только внутри своей области (Implementation)
Implementation	Реализация сервисов из API (конкретная логика)	Зависит от API, Core, иногда Utility

Основные ошибки, которых надо избегать

1

«Божественные» объекты

Ошибка: Модуль, который делает всё (обработка данных, логика, UI)

Пример: Класс MathProcessor, который и решает уравнения, и рисует графики

2

Жесткая связность

Ошибка: Модули напрямую зависят от внутренней реализации друг друга

Пример: Модуль Solver вызывает приватные методы модуля Matrix

3

Использование глобальных состояний

Ошибка: Общие глобальные переменные для нескольких модулей

Пример: Глобальный объект Config, который меняют все модули

4

Игнорирование границ модулей

Ошибка: Неясно, за что отвечает модуль, код дублируется

Пример: Функции для работы с JSON разбросаны по 5 модулям

5

Циклические зависимости

Ошибка: Модуль A зависит от B, а B зависит от A

Пример: NetworkModule использует Logger, а Logger зависит от NetworkModule для отправки логов

6

Микро-модули без смысла

Ошибка: 50 модулей, каждый из 10 строк кода

Пример: Отдельный модуль для функции calculateSum(a, b)

Разделение на модули в C++

Пространства имён

`math::vector`, `math::matrix`, `math::operations` –
разделение функциональности

Инкапсуляция:

Приватные поля `Vector3D` (`x_`, `y_`, `z_`)
и `Matrix3x3` (`data_`)

Минимальные зависимости:

Модуль `operations` зависит от `vector` и `matrix`,
но не наоборот

Чёткие границы:

Каждый модуль отвечает за свою задачу –
векторы, матрицы, операции над ними

Разделение на модули в C++

```
// vector.cpp
#include "vector.h"
#include <cmath>

namespace math::vector {
    Vector3D::Vector3D(double x, double y, double z)
        : x_(x), y_(y), z_(z) {}

    double Vector3D::length() const {
        return std::sqrt(x_ * x_ + y_ * y_ + z_ * z_);
    }

    double Vector3D::dot(const Vector3D& other) const {
        return x_ * other.x_ + y_ * other.y_ + z_ * other.z_;
    }
} // namespace math::vector
```

```
// vector.hpp
#pragma once

namespace math::vector {
    class Vector3D {
    public:
        Vector3D(double x, double y, double z);
        double length() const; // Длина вектора
        double dot(const Vector3D& other) const; //
        Скалярное произведение

    private:
        double x_, y_, z_;
    };
} // namespace math::vector
```

Разделение на модули в C++

```
// matrix.hpp
#pragma once

namespace math::matrix {
    class Matrix3x3 {
    public:
        Matrix3x3(
            double m11, double m12, double m13,
            double m21, double m22, double m23,
            double m31, double m32, double m33
        );

        double determinant() const; // Определитель матрицы

    private:
        double data_[3][3];
    };
} // namespace math::matrix
```

```
#include "matrix.h"

namespace math::matrix {
    Matrix3x3::Matrix3x3(
        double m11, double m12, double m13,
        double m21, double m22, double m23,
        double m31, double m32, double m33
    ) {
        data_[0][0] = m11; data_[0][1] = m12; data_[0][2] = m13;
        data_[1][0] = m21; data_[1][1] = m22; data_[1][2] = m23;
        data_[2][0] = m31; data_[2][1] = m32; data_[2][2] = m33;
    }

    double Matrix3x3::determinant() const {
        return data_[0][0] * (data_[1][1] * data_[2][2] - data_[1][2] *
data_[2][1])
            - data_[0][1] * (data_[1][0] * data_[2][2] - data_[1][2] * data_[2][0])
            + data_[0][2] * (data_[1][0] * data_[2][1] - data_[1][1] * data_[2][0]);
    }
} // namespace math::matrix
```

Разделение на модули в C++

```
// matrix.hpp
#pragma once

#include "vector.h"
#include "matrix.h"

namespace math::operations {
    // Умножение матрицы на вектор
    math::vector::Vector3D multiply(
        const math::matrix::Matrix3x3& mat,
        const math::vector::Vector3D& vec
    );
} // namespace math::operations
```

```
#include "operations.h"

namespace math::operations {
    Vector3D multiply(const Matrix3x3& mat, const
    Vector3D& vec) {
        double x = mat.data_[0][0] * vec.x() +
mat.data_[0][1] * vec.y() + mat.data_[0][2] * vec.z();
        double y = mat.data_[1][0] * vec.x() +
mat.data_[1][1] * vec.y() + mat.data_[1][2] * vec.z();
        double z = mat.data_[2][0] * vec.x() +
mat.data_[2][1] * vec.y() + mat.data_[2][2] * vec.z();
        return Vector3D(x, y, z);
    }
} // namespace math::operations
```

Разделение на модули в C++

```
#include <iostream>
#include "vector.h"
#include "matrix.h"
#include "operations.h"
```

```
int main() {
    using namespace math;
```

```
    // Создаем вектор и матрицу
```

```
    vector::Vector3D v(1, 2, 3);
```

```
    matrix::Matrix3x3 m(
```

```
        2, 0, 0,
```

```
        0, 2, 0,
```

```
        0, 0, 2
```

```
    );
```

```
    // Умножаем матрицу на вектор
    vector::Vector3D result =
    operations::multiply(m, v);
```

```
    std::cout << "Result: ("
        << result.x() << ", "
        << result.y() << ", "
        << result.z() << ")\n";
```

```
    return 0;
```

```
}
```

C++ | PIMPL

! Идиома PIMPL позволяет скрыть детали реализации класса, уменьшая зависимости между заголовочными файлами и ускоряя компиляцию

```
#pragma once
#include <memory>

class MyClass {
public:
    MyClass();           // Конструктор
    ~MyClass();          // Деструктор (обязателен для unique_ptr)
    MyClass(const MyClass&); // Конструктор копирования
    MyClass& operator=(const MyClass&); // Оператор присваивания
    void publicMethod(); // Публичный метод

private:
    class Impl; // Предварительное объявление внутреннего класса
    std::unique_ptr<Impl> pImpl; // Указатель на реализацию
};
```

```
#include "MyClass.h"
#include <iostream>

// Определение внутреннего класса
class MyClass::Impl {
public:
    void privateMethod() {
        std::cout << "Private method called. Value = " << value << std::endl;
    }

    int value = 42; // Приватные данные
};

// Реализация методов MyClass
MyClass::MyClass() : pImpl(std::make_unique<Impl>()) {}

MyClass::~MyClass() = default; // Деструктор должен быть в .cpp

MyClass::MyClass(const MyClass& other)
    : pImpl(std::make_unique<Impl>(*other.pImpl)) {}

MyClass& MyClass::operator=(const MyClass& other) {
    *pImpl = *other.pImpl;
    return *this;
}

void MyClass::publicMethod() {
    pImpl->privateMethod(); // Вызов метода реализации
}
```

C++ | Modules

! Модули в C++20 предоставляют современный способ организации кода, заменяя традиционные заголовочные файлы

```
// math.ixx — файл модуля
module;

// Экспортируемый модуль
export module math;

// Экспортируемые функции и классы
export namespace math {
    class Vector3D {
    public:
        Vector3D(double x, double y, double z);
        double length() const;

    private:
        double x_, y_, z_; // Приватные данные
    };

    // Экспортируемая функция
    double dot(const Vector3D& a, const Vector3D& b);
}

// Неэкспортируемая часть (инкапсулирована)
namespace math::details {
    double square(double x) { return x * x; } // Вспомогательная функция
}
```


C++ | Modules – ключевые слова

module

Объявляет модуль, указывая его начало и название



```
module math; // Объявление модуля "math"
```

export

Делает публичным внутренние части модуля



```
export namespace math {  
    class Vector { ... }; // Класс доступен извне  
}
```

import

Импортирует другой модуль для использования в текущем файле



```
import math; // Импорт модуля "math"
```

C++ | Modules – реализация

```
import math; // Импорт модуля

int main() {
    math::Vector3D v1(1, 2, 3);
    math::Vector3D v2(4, 5, 6);

    double len = v1.length(); // ОК: публичный метод
    double product = math::dot(v1, v2); // ОК: публичная
    функция

    // math::details::square(2.0); // Ошибка: details не
    экспортирован!
    return 0;
}
```

```
module math; // Указание, что это реализация
модуля "math"
#include <cmath>
namespace math {

    Vector3D::Vector3D(double x, double y, double z)
        : x_(x), y_(y), z_(z) {}

    double Vector3D::length() const {
        using namespace details;
        return std::sqrt(square(x_) + square(y_) +
            square(z_));
    }

    double dot(const Vector3D& a, const Vector3D& b) {
        return a.x_ * b.x_ + a.y_ * b.y_ + a.z_ * b.z_;
    }
}
```

Зачем использовать модули

Инкапсуляция

Скрытие деталей реализации (приватные поля или методы не видны в интерфейсе)

Скорость компиляции

Модули компилируются один раз и кэшируются

Устранение проблем с заголовками

Нет необходимости в `#include` и макросах (`#ifndef`)

Чистые зависимости

Явное указание, какие компоненты используются

Механизм компиляции модулей

#include (заголовочные файлы)

- Препроцессор вставляет содержимое заголовочного файла напрямую в место включения
- Если один заголовок включен в 100 файлов, он обрабатывается 100 раз

Модули (C++20)

- Модуль компилируется один раз и сохраняется в предварительно обработанном виде
- При импорте используется уже готовое представление

Ускорение компиляции



Причины ускорения компиляции

1

Кэширование

Модули компилируются один раз и повторно используются, тогда как заголовки обрабатываются каждый раз при включении

2

Сокращение работы препроцессора

Модули не требуют обработки `#define`, `#ifdef` и других макросов в каждом файле

3

Инкапсуляция

В модулях экспортируется только публичный API, а детали реализации скрыты
Это уменьшает объём кода, который нужно обрабатывать

Rust Crates

! Пакет (crate) – единица компиляции в Rust
Управляется через Cargo.toml

Типы пакетов:

- Бинарный крейт
Исполняемый файл (src/main.rs)
- Библиотечный крейт
Набор функций/типов для повторного использования (src/lib.rs)

```
ristle obsidian-to-google-sync (f/obsidian_sync)* tree .
.
├── Cargo.lock
├── Cargo.toml
├── container-compose.yaml
├── diesel.toml
├── README.md
├── src
│   ├── db
│   │   ├── core.rs
│   │   ├── db.rs
│   │   └── mod.rs
│   ├── google_api
│   │   ├── google.rs
│   │   └── mod.rs
│   ├── google.rs
│   ├── main.rs
│   ├── models
│   ├── obsidian.rs
│   ├── obsidian_wrapper
│   │   ├── Cargo.toml
│   │   └── src
│   │       └── lib.rs
│   ├── schema.rs
│   ├── sync_tasks.rs
│   └── tasks.rs
└── 7 directories, 18 files
```

Rust | Модули и управление видимостью

! Модули в Rust помогают организовать код в логические блоки и управлять областью видимости

Все элементы по умолчанию приватны

<code>pub</code>	Делает элемент (модуль, структуру, метод) доступным вне текущей области видимости
------------------	---

Уровни видимости:

<code>pub(crate)</code>	Доступно только внутри текущего крейта
<code>pub(super)</code>	Доступно в родительском модуле
<code>pub(in path)</code>	Доступно в указанном модуле Например, <code>pub(in path::vector)</code>

```
// Файл: src/lib.rs
mod math {           // Объявление модуля
    pub mod vector { // Публичный подмодуль
        pub struct Vec3 { // Публичная структура
            x: f64,
            y: f64,
            z: f64,
        }

        impl Vec3 {
            pub fn new(x: f64, y: f64, z: f64) -> Self { // Публичный метод
                Vec3 { x, y, z }
            }
        }
    }
}

// Использование модуля
use math::vector::Vec3;

fn main() {
    let v = Vec3::new(1.0, 2.0, 3.0);
}
```

Rust Трейты (trait) и их расширения

! Трейты определяют общее поведение для разных типов

- Трейт-экстеншены (Trait Extensions)
- Расширение функциональности существующих типов через трейты

```
// Трейт для геометрических фигур
pub trait Area {
    fn area(&self) -> f64;
}

// Реализация для структуры Circle
struct Circle { radius: f64 }

impl Area for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }
}
```

```
// Трейт для преобразования в JSON
trait ToJson {
    fn to_json(&self) -> String;
}

// Реализация для типа из сторонней библиотеки
impl ToJson for some_external_lib::Data {
    fn to_json(&self) -> String {
        serde_json::to_string(self).unwrap()
    }
}
```




Модульная архитектура

Преимущества и принципы

API

Определение и виды

Интеграция внешних библиотек

Пример для CMake

Как делиться кодом

Общие шаги

REST API

Базовые концепции



Мы здесь



API (Application Programming Interface)

Это набор правил и протоколов, который позволяет разным программам взаимодействовать друг с другом

API может быть использован для различных целей, включая:

1

Взаимодействие с внешними сервисами

Многие приложения и веб-сервисы предоставляют API, которые позволяют другим приложениям получать доступ к их функциональности и данным

3

Интеграция с аппаратным обеспечением

API также используются для взаимодействия с аппаратным обеспечением, таким как принтеры, камеры, датчики и другие устройства

2

Расширение функциональности приложений

Например, плагины и расширения для браузеров

4

Обмен данными

API часто применяются для обмена данными между различными частями одной программы или между разными программами

Виды API

Веб-API (Web APIs)

Web-API предоставляют доступ к функциональности и данным через интернет с использованием стандартных протоколов, таких как HTTP

RESTful API и SOAP API – это примеры веб-API, которые позволяют взаимодействовать с удаленными веб-сервисами

Библиотеки и SDK (Software Development Kit)

Эти API предоставляются в виде библиотек или наборов инструментов, которые разработчики могут внедрить непосредственно в свои приложения

Они часто используются для работы с конкретными языками программирования или платформами

API операционных систем

Эти API предоставляют доступ к функциям операционных систем, таким как файловая система, сеть и управление устройствами

Примеры включают WinAPI для Windows и POSIX API для Unix-подобных операционных систем

API DB/DWH

API, предназначенные для взаимодействия с базами данных
Примеры включают JDBC для Java и SQLAlchemy для Python

Преимущества протоколов API

Стандартизация

- + Протоколы API определяют общие структуры и форматы данных, что способствует стандартизации и обеспечивает согласованность взаимодействия между разными приложениями и сервисами

Модульность

- + Протоколы API позволяют разбить сложные системы на более мелкие, независимые компоненты, что упрощает разработку и поддержку приложений

Улучшенное взаимодействие

- + Протоколы API способствуют сотрудничеству между разными командами разработчиков и организациями, позволяя им создавать приложения и сервисы, которые взаимодействуют друг с другом

Безопасность

- + Многие протоколы API предоставляют механизмы аутентификации и авторизации, что позволяет контролировать доступ к данным и функциональности

Расширяемость

- + API-протоколы могут быть расширены и изменены без необходимости изменения всей системы, что упрощает добавление новых функций и возможностей

Недостатки протоколов API

Сложность

Некоторые API-протоколы могут быть сложными и требовать глубокого понимания для их использования и реализации

Совместимость

Изменения в протоколах API могут привести к проблемам совместимости, особенно если старые версии клиентов или серверов не поддерживают новые изменения

Производительность

Некоторые протоколы могут иметь накладные расходы в виде лишнего объема данных или дополнительных запросов, что может сказаться на производительности

Пример API проекта в Python

```
geometry/
├── geometry/
│   ├── __init__.py
│   ├── shapes.py
│   └── utils.py
├── tests/
│   ├── __init__.py
│   └── test_shapes.py
├── setup.py
├── requirements.txt
└── README.md
```

Исходный код
Инициализация пакета
Классы фигур
Вспомогательные функции
Тесты

Конфигурация пакета
Зависимости
Документация

Пример API проекта в Python

```
# geometry/utils.py
```

```
def validate_positive(value: Union[int, float],
name: str) -> None:
    """Проверяет, что значение
    положительное."""
    if value <= 0:
        raise ValueError(f"{name} must be positive.
Got {value}.")
```

```
# geometry/__init__.py
```

```
from .shapes import Circle, Rectangle

__all__ = ["Circle", "Rectangle"]
```

```
# geometry/shapes.py
```

```
from typing import Union
from .utils import validate_positive

class Circle:
    """Класс для работы с окружностями."""

    def __init__(self, radius: Union[int, float]):
        validate_positive(radius, "Radius")
        self.radius = radius

    def area(self) -> float:
        """Вычисляет площадь окружности."""
        return 3.14159 * self.radius ** 2

    def circumference(self) -> float:
        """Вычисляет длину окружности."""
        return 2 * 3.14159 * self.radius
```

```
class Rectangle:
```

```
    """Класс для работы с прямоугольниками."""

    def __init__(self, width: Union[int, float],
height: Union[int, float]):
        validate_positive(width, "Width")
        validate_positive(height, "Height")
        self.width = width
        self.height = height

    def area(self) -> Union[int, float]:
        """Вычисляет площадь
        прямоугольника."""
        return self.width * self.height

    def perimeter(self) -> Union[int, float]:
        """Вычисляет периметр
        прямоугольника."""
        return 2 * (self.width + self.height)
```

Пример API проекта в Python

```
# tests/test_shapes.py
import pytest
from geometry.shapes import Circle, Rectangle
```

```
@pytest.fixture
def circle():
    return Circle(radius=5)
```

```
@pytest.fixture
def rectangle():
    return Rectangle(width=4, height=3)
```

```
def test_circle_area(circle):
    assert round(circle.area(), 2) == 78.54
```

```
def test_circle_circumference(circle):
    assert round(circle.circumference(), 2) == 31.42
```

```
def test_rectangle_area(rectangle):
    assert rectangle.area() == 12
```

```
def test_rectangle_perimeter(rectangle):
    assert rectangle.perimeter() == 14
```

```
# setup.py
from setuptools import setup, find_packages

setup(
    name="geometry",
    version="0.1.0",
    packages=find_packages(),
    install_requires=[], # Зависимости (если есть)
    author="Your Name",
    description="A library for geometric calculations",
    url="https://github.com/yourusername/geometry",
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
    ],
)
```

```
# requirements.txt
pytest>=6.0.0
```


API | Readme.md

Geometry Library

Библиотека для вычисления площадей и периметров геометрических фигур

Установка

```
```bash  
pip install .
```
```



Модульная архитектура

Преимущества и принципы

API

Определение и виды

Интеграция внешних библиотек

Пример для CMake

Как делиться кодом

Общие шаги

REST API

Базовые концепции



← Мы здесь



CMake | Варианты интеграции внешних библиотек

Использование `find_package`
(для системных библиотек)

Использование `FetchContent`
(загрузка из репозитория)

Ручное указание путей

Подключение через `add_subdirectory`
(для локальных библиотек)

Использование find_package

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Поиск Boost (требуется компонент system)
find_package(Boost 1.70 REQUIRED COMPONENTS system)

# Создание исполняемого файла
add_executable(my_app main.cpp)

# Подключение библиотеки
target_link_libraries(my_app PRIVATE Boost::boost Boost::system)
```

Использование FetchContent

```
cmake_minimum_required(VERSION 3.14)
project(MyProject)

include(FetchContent)
FetchContent_Declare(
  json
  GIT_REPOSITORY https://github.com/nlohmann/json.git
  GIT_TAG v3.11.2
)
FetchContent_MakeAvailable(json)

add_executable(my_app main.cpp)
target_link_libraries(my_app PRIVATE nlohmann_json)
```

Ручное указание путей

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Пути к заголовкам и бинарникам
set(MY_LIB_INCLUDE_DIR "/path/to/lib/include")
set(MY_LIB_LIBRARY "/path/to/lib/libmylib.a")

# Создание цели
add_executable(my_app main.cpp)


# Подключение
target_include_directories(my_app PRIVATE ${MY_LIB_INCLUDE_DIR})
target_link_libraries(my_app PRIVATE ${MY_LIB_LIBRARY})
```

Подключение через add_subdirectory

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Добавление поддиректории с библиотекой
add_subdirectory(thirdparty/mylib)

add_executable(my_app main.cpp)
target_link_libraries(my_app PRIVATE mylib)
```



Модульная архитектура

Преимущества и принципы

API

Определение и виды

Интеграция внешних библиотек

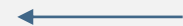

Пример для CMake

Как делиться кодом

Общие шаги

REST API

Базовые концепции



Мы здесь

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Создайте публичный или приватный репозиторий на платформе, такой как GitHub, GitLab или Bitbucket. Это обеспечит централизованное место для хранения и обмена кодом.

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Организуя код в логичную и понятную структуру каталогов
Это может включать разделение на модули, библиотеки,
примеры использования и тесты

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Создайте подробную документацию, включающую:

- Описание проекта и его назначение
- Инструкции по установке и настройке
- Руководство по использованию
- API-документацию для публичных функций и классов
- Примеры использования

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Выберите и укажите соответствующую лицензию для вашего проекта, чтобы определить условия использования и распространения кода

- **MIT**
Простая и свободная лицензия
- **Apache 2.0**
Свободная лицензия, обязательно указание первоисточника
- **BSD**
Семейство свободных лицензий
- **GPL (GNU General Public License)**
Лицензия, требующая, чтобы производные работы также распространялись под той же лицензией, обеспечивая открытость исходного кода
- **Proprietary / Closed Source**
Лицензия, при которой все права остаются за владельцем

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Добавьте модульные и интеграционные тесты, чтобы гарантировать качество и работоспособность кода

Настройте CI-pipeline, чтобы автоматически собирать, тестировать и развертывать ваш проект при каждом изменении в репозитории

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Создайте файл **CONTRIBUTING.md**, в котором опишите, как другие разработчики могут вносить свой вклад в проект (например, как создавать запросы на слияние, сообщать об ошибках и т.д.)

Основные шаги

1. Репозиторий

2. Структура проекта

3. Документация

4. Лицензия

5. Тесты

6. Continuous Integration (CI)

7. Руководство по внесению вклада

8. Лидеры проекта и контакты

Укажите основных участников проекта и их контактную информацию, чтобы другие разработчики могли обращаться с вопросами и предложениями



Модульная архитектура

Преимущества и принципы

API

Определение и виды

Интеграция внешних библиотек

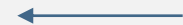
Пример для CMake

Как делиться кодом

Общие шаги

REST API

Базовые концепции



Мы здесь

Протоколы API

RESTful API (Representational State Transfer)

RESTful API основаны на принципах архитектуры REST и используют стандартные HTTP методы (GET, POST, PUT, DELETE) для взаимодействия с ресурсами

SOAP (Simple Object Access Protocol)

Это протокол на основе XML, который предоставляет строгую структуру для обмена сообщениями между клиентами и серверами

SOAP API часто используются в корпоративных приложениях и могут предоставлять расширенные возможности безопасности

GraphQL

Современный протокол, который позволяет клиентам запрашивать только те данные, которые им нужны

Вместо предоставления фиксированных точек доступа, как REST, GraphQL дает клиентам гибкость создавать свои запросы

JSON-RPC и XML-RPC

Эти протоколы используются для удаленного вызова процедур и передачи данных в формате JSON или XML

Они предоставляют простой способ взаимодействия между клиентами и серверами

REST API

Это стиль архитектуры API, использующий стандартные HTTP методы (GET, POST, PUT, DELETE) и работает с ресурсами, представленными в виде URL-адресов

Основные характеристики:

В REST API всё представлено в виде ресурсов, которые могут быть идентифицированы уникальными URL-адресами
Ресурсы могут быть любыми коллекциями данных

REST использует стандартные методы HTTP для выполнения операций над ресурсами

REST API обычно работает с данными, представленными в формате JSON или XML (также доступны и другие форматы)

RESTful сервисы не сохраняют состояние между запросами. Каждый запрос от клиента должен содержать все необходимые данные для выполнения операции – это делает их более масштабируемыми и легче в поддержке

REST API обладают унифицированным интерфейсом, что означает, что клиенты и серверы могут взаимодействовать между собой, следуя общим правилам, без необходимости знания подробностей реализации

Преимущества REST API

- + Простота использования и понимания
- + Стандартное использование HTTP методов и кодов состояния
- + Масштабируемость и гибкость
- + Возможность использования веб-кэширования для оптимизации производительности

Недостатки REST API

- Отсутствие строгой спецификации может привести к разнообразию реализаций и сложностям в согласованности интерфейсов
- Не всегда идеально подходит для сложных операций, требующих транзакций или многоэтапных процессов

Методы REST API

GET

Метод GET используется для получения данных с сервера

GET-запросы обычно не должны иметь побочных эффектов на сервере, они должны быть безопасными и повторный GET-запрос не должен изменять состояние ресурса

POST

Метод POST используется для создания нового ресурса на сервере или для отправки данных для обработки
POST-запросы могут изменять состояние сервера

PUT

Метод PUT используется для обновления существующего ресурса или для создания ресурса, если он не существует
Повторное выполнение одного и того же PUT-запроса не должно изменять состояние ресурса

DELETE

Метод DELETE используется для удаления ресурса с сервера

Клиент отправляет DELETE-запрос, указывая URL-адрес удаляемого ресурса
Повторное выполнение DELETE-запроса на уже удаленный ресурс не должно вызывать ошибку

PATCH

Метод PATCH используется для частичного обновления ресурса

В отличие от PUT, который обновляет ресурс полностью, PATCH позволяет обновлять только определенные части ресурса

Какие данные можно отправлять?

Текстовые форматы

- JSON (JavaScript Object Notation)
- YAML
- XML

Бинарные форматы

- MessagePacked
- ProtobufProtocol

Какие данные можно отправлять?

JSON

Используется в REST API, конфигах, NoSQL (MongoDB)

- + Универсальность (поддержка всеми языками)
- + Гибкость (без схемы)
- Избыточный размер

Легко читается человеком | Пример:

```
{
  "name": "Alice",
  "age": 30,
  "is_student": false
}
```

MessagePacked

Бинарный аналог JSON

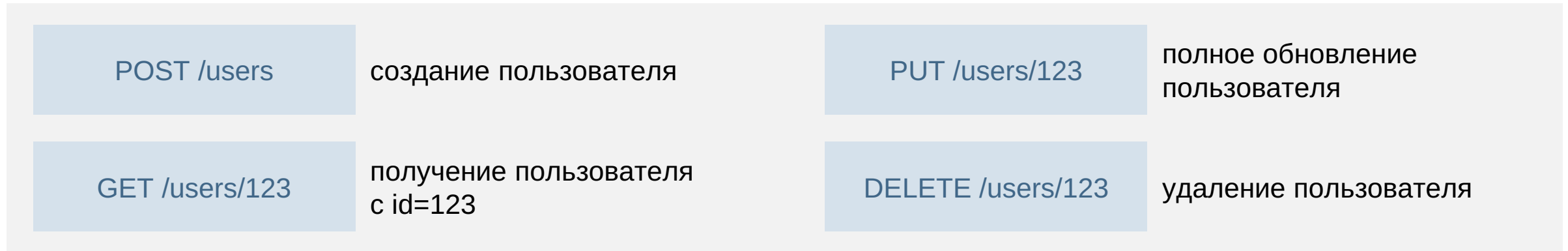
- + Меньше размер, чем у JSON
- + Широкая поддержка
- Сложность отладки (бинарные данные)

Пример в hex:

```
83 A4 6E 61 6D 65 A5 41 6C 69 63 65 A3 61 67
```

Пример хорошего API | Оформление

Оформление внутри одного сервиса:



Оформление внутри одного модуля:



Пример хорошего API | Статусные коды HTTP

2xx (Успех)

- 200 OK – успешный GET, PUT, DELETE
- 201 Created – успешное создание (POST)
- 204 No Content – успешный запрос без тела ответа

4xx (Ошибка клиента)

- 400 Bad Request – некорректный запрос
- 401 Unauthorized – требуется аутентификация
- 403 Forbidden – доступ запрещен
- 404 Not Found – ресурс не существует

5xx (Ошибка сервера)

- 500 Internal Server Error – общая ошибка сервера

Примеры

```
use actix_web::{get, post, web, App,
  HttpServer, Responder};
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize)]
struct Person {
  id: u32,
  name: String,
  age: u32,
}

#[get("/people")]
async fn get_people() -> impl Responder
{
  let people = vec![
    Person {
      id: 1,
      name: "John Doe".to_string(),
      age: 30,
    },
```

```
    Person {
      id: 2,
      name: "Jane Smith".to_string(),
      age: 25,
    },
  ];
  web::Json(people)
}
```

```
#[post("/people")]
async fn create_person(person:
  web::Json<Person>) -> impl Responder {
  println!("Created person: {:?}",
    person);
  web::Json(person.into_inner())
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
  HttpServer::new(|| {
    App::new()
      .service(get_people)
      .service(create_person)
  })
  .bind("127.0.0.1:8000")?
  .run()
  .await
}
```



Спасибо за внимание!
Ваши вопросы?

← Оставьте, пожалуйста, обратную связь

