**Титульный лист**

**Технический доклад**  
**Архитектура и работа full-stack проекта (Go + Vue 3)**

Разработчик: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Организация/Университет: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Год: \_\_\_\_\_\_\_

**Содержание**

1. Введение
2. Общая структура проекта
3. Backend — серверная часть
   * cmd/server/main.go
   * internal/handlers/
   * local.go
   * static/
   * Middleware и безопасность
   * Шаблоны кода для разных языков
   * Health-checks
4. Frontend — клиентская часть
   * main.js и App.vue
   * api.js
   * Router и Pinia
   * Компоненты и логика SPA
5. Взаимодействие Frontend ↔ Backend
6. Особенности и “приколы” проекта
7. Возможные улучшения
8. Заключение

**1. Введение**

Проект — это **веб-платформа для обучения программированию**, где пользователи могут:

* Выбирать задачи по языку и теме
* Получать шаблонный код для каждого языка
* Выполнять код прямо в браузере
* Проверять результаты своих решений

**Стек технологий:**

* Backend — Go (Golang), REST API
* Frontend — Vue 3, Pinia, Vue Router
* JSON — формат обмена данными

Цель архитектуры — **отделить серверную логику от фронтенда**, чтобы можно было масштабировать и модифицировать систему отдельно.

**2. Общая структура проекта**

project-root/

├── backend/

│ ├── cmd/server/ — запуск серверной части

│ ├── internal/handlers/ — обработка API запросов

│ ├── local.go — выполнение кода локально

│ └── static/ — скомпилированный фронтенд (опционально)

└── frontend/

├── src/

│ ├── main.js — инициализация приложения

│ ├── App.vue — корневой компонент

│ ├── api.js — клиент для общения с сервером

│ ├── router/ — маршрутизация

│ └── stores/ — управление состоянием (Pinia)

├── index.html

└── vite.config.js

**Пояснения:**

* cmd/server/main.go — стартовая точка для сервера, подключает роуты, middleware и запускает слушание порта.
* internal/handlers/ — каждый файл внутри содержит функцию для конкретного эндпоинта API, например /execute, /check, /task/....
* local.go — отвечает за выполнение кода на сервере локально. Можно расширить для безопасного sandbox’а.
* static/ — сюда кладется фронтенд-билд, чтобы сервер мог отдавать SPA как статические файлы.
* frontend/src/main.js — инициализация Vue приложения, подключение Pinia и Router.
* api.js — абстрактный клиент для всех запросов к серверу.
* router/ — маршруты SPA.
* stores/ — хранит глобальное состояние приложения, например текущий выбранный язык или задачу.

**3. Backend — серверная часть**

**3.1 cmd/server/main.go**

* **Задача:** запустить сервер и подключить все обработчики запросов.
* Подключает **middleware** для CORS и логирования.
* Определяет **порты, таймауты** для безопасности и стабильности.
* Определяет маршруты API: /api/execute, /api/check, /api/task/..., /api/health, /health.

**Принцип работы:**

1. Сервер получает HTTP-запрос
2. Middleware проверяет заголовки и логирует запрос
3. Запускается соответствующий handler
4. Возвращается JSON ответ

**3.2 internal/handlers/**

* Каждый файл/функция — это **обработчик конкретного запроса**:
  + TasksHandler — получает список задач
  + ExecuteHandler — выполняет код пользователя
  + CheckHandler — проверяет решение на тестах
  + LoginHandler / GuestAuthHandler / RegisterHandler — авторизация
* **Почему важно:** всё распределено по функциональным единицам, легко добавлять новые эндпоинты.

**3.3 local.go**

* **Задача:** выполнять код на сервере.
* Для каждого языка есть отдельная логика: Python, JavaScript, C++, Java.
* Используется как локальный “sandbox” для быстрого тестирования.
* **Пример:** пользователь отправил Python-код → local.go запускает интерпретатор → возвращает stdout/stderr.

**3.4 static/**

* Если деплой монолитный, фронтенд собирается и кладется сюда.
* Сервер через Go отдает SPA как статические файлы.
* Если статических файлов нет, сервер работает в режиме **API-only**.

**3.5 Middleware и безопасность**

* **CORS middleware:** разрешает запросы с разных доменов.
* **Logging middleware:** логирует все запросы кроме health-check, фиксирует время выполнения.
* **Security headers:** X-Frame-Options, X-Content-Type-Options, X-XSS-Protection — защищает от базовых web-атак.

**3.6 Шаблоны кода для разных языков**

* Python → print("Hello World")
* JavaScript → console.log("Hello World")
* C++ → простой main.cpp с cout << "Hello World"
* Java → класс Main с System.out.println("Hello World")

**Пояснение:** удобно для новичков, чтобы сразу видеть стартовую точку.

**3.7 Health-checks**

* /health → базовая проверка системы, без CORS, для балансировщика нагрузки.
* /api/health → проверка состояния API, возвращает JSON с портом, версией и средой.
* **Прикол:** можно мониторить uptime сервера и использовать в CI/CD.

**4. Frontend — клиентская часть**

**4.1 main.js и App.vue**

* **main.js** инициализирует Vue, Pinia и Router, монтирует приложение в DOM.
* **App.vue** — корневой компонент с <router-view />, куда подставляются текущие маршруты.

**4.2 api.js**

* Абстрактный клиент для всех HTTP-запросов.
* Функции:
  + executeCode(code, language)
  + checkSolution(code, language, taskId)
  + getTask(lang, topic, taskId)
* Все функции возвращают **JSON** и обрабатывают ошибки.

**4.3 Router и Pinia**

* **Router:** маршруты /tasks, /execute, /check, /login, /guest.
* **Pinia:** хранит глобальное состояние приложения, например:
  + текущий выбранный язык
  + текущая задача
  + результат выполнения кода

**4.4 Компоненты SPA**

* Каждая страница — отдельный компонент Vue.
* <router-view /> динамически подставляет нужный компонент.
* Frontend полностью отделен от backend, работает как SPA.

**5. Взаимодействие Frontend ↔ Backend**

**Принцип:**

[Vue SPA] → [api.js] → [Go сервер] → [Handler] → [JSON]

**Пример:**

1. Пользователь открывает задачу Python
2. api.js вызывает getTask("python", "loops", "1")
3. Сервер отдает JSON: описание задачи + шаблон кода
4. Пользователь пишет решение → executeCode
5. Сервер через local.go выполняет код и возвращает результат

**6. Особенности и “приколы” проекта**

* Middleware для CORS, логирования и безопасности
* Каждый язык имеет отдельную функцию для шаблонного кода
* Health-checks для мониторинга и балансировщиков
* Возможность работать как **API-only** или **монолит (API + SPA)**
* Все запросы обрабатываются JSON — удобно для фронтенда

**7. Возможные улучшения**

* JWT авторизация
* Подключение базы данных (PostgreSQL, SQLite)
* Централизованное логирование и метрики
* Unit-тесты и CI/CD

**8. Заключение**

Проект демонстрирует **модульную архитектуру full-stack приложения**, где frontend и backend разделены.

* Middleware делает систему безопасной и прозрачной
* Шаблоны кода и health-checks упрощают работу с задачами
* Легко масштабировать, добавлять новые языки и расширять функционал