ChatGPT

Проектирование стека и архитектуры фронтенд-портфолио

Исходя из заданных критериев (Vercel Design System, шрифт Geist/Inter, Swiss Style дизайн, структура контента как в Notion), ниже предложены варианты стека и подробная архитектура проекта. Решение структурировано по требуемым пунктам, с четкими списками и описаниями для прямого использования в проекте.

Варианты стека

Вариант A – быстрый старт

- **Next.js (Pages Router)** – классический роутер Next.js с директории `pages`. Обеспечивает статическую генерацию (SSG) страниц без SSR, что идеально для портфолио [1] [2]. Страницы собираются на этапе билда и раздаются через CDN, обеспечивая высокую скорость загрузки и SEO.
- **React + TypeScript** – современный фронтенд-стек: React для создания UI-компонентов, TypeScript для статической типизации.
- **UI-библиотека: Tailwind CSS + shadcn/ui** – используется утилитарный CSS-фреймворк Tailwind для быстрого стилирования. Компоненты берутся из набора shadcn/ui [3] – это коллекция шаблонов React-компонентов на базе Radix UI и Tailwind. Они копируются в проект (без внешних зависимостей) и сразу доступны для кастомизации, при этом **доступность** и логика уже встроены (ARIA-атрибуты, фокус-менеджмент) [4]. Такой подход дает гибкость дизайна и отсутствие лишнего кода в рантайме.
- **Темизация (Light/Dark)** – с самого начала заложена поддержка светлой и темной темы. Можно использовать библиотеку `next-themes` для сохранения предпочтений пользователя. Базовая стилистика – минималистичная, в духе Vercel: светлая тема (белый фон, черный текст) и темная тема (темный фон, светлый текст) с единым дизайном компонентов. *(Примеры современных шаблонов включают автоматическое переключение темы и MDX-контент [5].)*
- **Контент как MDX/JSON** – проекты, навыки и прочий контент хранятся в локальных файлах (например, Markdown/MDX для страниц и JSON для структурированных данных). MDX позволяет оформлять контент гибко (Markdown + JSX) по аналогии с блоками в Notion. При сборке приложения эти файлы парсятся и генерируются статические страницы. Никакой внешней CMS не требуется, контент легко версионируется вместе с кодом.
- **Форма контактов: mailto** – форма обратной связи реализована как простая ссылка `mailto:` на email, без серверной обработки (для быстроты запуска). Endpoint `/api/contact` при этом планируется и описывается в документации (см. **Backend** ниже), но в данной минимальной реализации не вызывается.

Когда использовать Вариант A: когда нужен максимально быстрый результат с минимальной конфигурацией. Next.js Pages Router привычен и прост, SSG дает отличное быстродействие для статичного портфолио, а комбинация Tailwind+shadcn/ui позволяет быстро собрать UI без разработки дизайна с нуля.

Вариант B – более “показательный” стек

- **Next.js (App Router)** – современный подход с директорией `app`. Используются React Server Components и вложенные layouts Next 13+. Это дает более гибкую архитектуру и высокую производительность за счет рендеринга на сервере на этапе билда и разделения кода. App Router упрощает создание общих макетов страниц и маршрутов. Несмотря на повышенную сложность, этот вариант демонстрирует владение новейшими практиками Next.js [6].
- **React 18+ и TypeScript** – актуальная версия React (с поддержкой Concurrent Features) и строгий TypeScript для надежности кода. Включены новые возможности (хуки, Context API) для грамотного управления состоянием (например, для переключения тем).
- **Tailwind CSS + shadcn/ui с кастомизацией** – подключается тот же набор компонентов shadcn, но кастомизируется под Vercel Design System. Определяются дизайн-токены (цвета, отступы, шрифты) в Tailwind темах, чтобы компоненты соответствовали фирменному стилю. Swiss Style принципы (строгая модульная сетка, много белого пространства, крупная типографика) реализованы через кастомные utility-классы и макеты. Используются шрифты **Geist Sans** / **Inter** в связке с **Geist Mono** для кода – эти шрифты разработаны Vercel и основаны на принципах швейцарской типографики [7], обеспечивая чистый современный вид. Благодаря shadcn/ui все компоненты изначально доступны и *доступны* (Radix UI под капотом гарантирует фокусировку, навигацию с клавиатуры и т.д.), QA фокусируется только на бизнес-логике [4].
- **Статическая генерация с Contentlayer** – для управления контентом подключается **Contentlayer** (библиотека, генерирующая типобезопасный контент). Она автоматически парсит MDX/Markdown файлы (проекты, статьи и пр.) и предоставляет их как готовые данные с TypeScript-типами [8]. В результате, добавление нового проекта в папку контента автоматически отражается на сайте. Все страницы генерируются статически (SSG) при билде, что дает максимум скорости и SEO, а при необходимости можно настроить Incremental Static Regeneration для обновления контента. *(По сути, как и в варианте A, мы избегаем SSR – контент предрендерится и кэшируется глобально [1].)*
- **Дополнительные технологии для демонстрации**:
- *Анимации и UX*: подключается Framer Motion для плавных анимаций, микровзаимодействий (например, анимированное появление проектов, плавный скролл и т.д.) – добавляет «живости» портфолио.
- *Иконки*: используется набор иконок (например, **Lucide** или **Heroicons**) для визуализации ссылок, технологий и пр.
- *Валидация и утилиты*: для формы контактов и других частей может использоваться библиотека **Zod** или **Yup** для схем валидации данных (демонстрация заботы о корректности данных).
- *Тестирование*: для показательного стека можно предусмотреть настройку Jest + React Testing Library для тестирования компонентов и функций, а также ESLint/Prettier для поддержания качества кода. (Эти инструменты не влияют на функциональность портфолио, но демонстрируют профессиональный подход.)
- **Форма контактов с API** – хотя форма по-прежнему представлена как `mailto:` для пользователя, на бэкенде уже реализован черновой вариант API endpoint (`/api/contact`), который можно задействовать в будущем. Это показывает умение проектировать full-stack функциональность: endpoint написан с учетом валидации и защиты от спама (см. раздел **Backend /api/contact** ниже).

Когда использовать Вариант B: когда цель – продемонстрировать глубину навыков и современный подход. Такой стек сложнее и дольше в настройке, зато отражает владение Next.js

13+, контентными решениями без CMS, анимациями и общей архитектурой приложений. Портфолио получится не только содержательным, но и технически впечатляющим.

Архитектура проекта

Проект построен многослойно, с четким разделением обязанностей, что облегчает поддержку и масштабирование. Ниже описаны основные слои и части приложения, выбор роутинга, структура папок и правила именования.

Слои приложения

- **Компоненты UI** – переиспользуемые презентационные компоненты (кнопки, карточки, навигационные элементы и др.). Базовые атомарные компоненты (в т.ч. из shadcn/ui) хранятся в `components/ui` и реализуют единый стиль и поведение. Над ними могут быть композиционные компоненты (организмы) в `components/` – например, шапка сайта `Header`, список проектов `ProjectsList` и т.д., собранные из базовых компонентов.
- **Страницы и Layoutы** – каждую страницу представляет компонент-страница, отвечающий за сборку контента. В App Router страницы находятся в `app/` (например, `app/projects/page.tsx` для раздела проектов). Можно определить общий layout: файл `app/layout.tsx` для обёртки (навигация, футер) на всех страницах, а также вложенные layout'ы для отдельных секций если нужно (напр. отдельный layout для страниц проектов). При Pages Router аналогично, страницы в `pages/`, а общие компоненты (Header/Footer) подключаются на каждой странице или через `_app.tsx`.
- **Логика и утилиты** – отдельные функции и hooks, не связанные с отображением. В папке `lib/` или `utils/` можно хранить helpers: например, форматирование дат, фильтрация списка проектов, конфигурация валидации формы. Также здесь могут быть контексты React (например, Context для темы оформления) и клиенты внешних сервисов (хотя внешних сервисов минимально, но например отправка email или подключение аналитики).
- **Контент и данные** – отдельный слой для хранения и загрузки контента. Статические файлы MDX/Markdown, JSON располагаются в папке `content/` (или `data/`). Специальные скрипты или библиотеки (в варианте B – Contentlayer) обрабатывают эти файлы, превращая их в структуру данных, доступную на этапе сборки. Таким образом, страницы получают необходимые данные через импорты или функции **getStaticProps** (в Pages Router) либо напрямую через асинхронные компоненты (в App Router). Структура контента вдохновлена Notion: информация организована иерархически и модульно – например, проекты в отдельных файлах с метаданными (заголовок, описание, тэги), навыки сгруппированы по категориям, страницы состоят из блоков контента (MDX позволяет вставлять заголовки, списки, изображения подобно тому, как в Notion можно комбинировать разные блоки).
- **Стили и тема** – Tailwind CSS используется для оформления, поэтому глобальные стили и токены хранятся в `styles/` (например, `globals.css` подключает базовые стили, в том числе Reset/Normalize и импортирует Tailwind). Также там могут быть файлы для кастомных тем, например, определение CSS-переменных для светлой/темной темы. Переключение темы осуществляется либо через CSS (`prefers-color-scheme`) плюс `next-themes` для хранения выбора, либо через добавление/удаление класса (например, `class="dark"` на `html`). Все отступы, размеры, цвета унифицированы по дизайн-системе (например, сетка 8px, цвета из палитры Vercel). Swiss Style подход отражается в CSS: четкая колонная сетка (например, с использованием CSS Grid или flex-колонок), большие поля (margin/padding) между блоками, преобладание черно-белого с акцентным цветом (как в Vercel DS).

Маршрутизация (Next.js Routing)

Проект использует **файловый роутинг Next.js**. Выбор сделан в пользу **Next.js App Router** (вариант B), поскольку он более современный и упрощает ряд задач: - Страницы размещаются в папке `app/` соответствуя структуре URL. Например, главная страница – `app/page.tsx`, страница проектов – `app/projects/page.tsx`, детальная страница проекта – `app/projects/[slug]/page.tsx`. Next.js сам генерирует маршруты на основе структуры папок. - App Router поддерживает вложенные layout'ы и React Server Components, что улучшает производительность и разработку. Поскольку контент статический, все эти страницы можно помечать как **статически генерируемые** (по умолчанию в App Router, если нет динамических данных без кеша, страница рендерится на билд-фазе). - *Альтернатива:* При использовании Pages Router (вариант A) маршруты определяются файлами в `pages/`. Например, `pages/index.tsx`, `pages/projects/[slug].tsx`. Вместо встроенных layout, общие части (навигация) подключаются через `_app.tsx` или на уровне компонентов. Для статического рендеринга применяются `getStaticProps` / `getStaticPaths` для страниц проектов и т.д. Оба подхода обеспечивают SEO-оптимизированный, предрендеренный контент; мы выбрали App Router для демонстрации новейших возможностей Next.js.

Маршрутизация охватывает следующие основные страницы/разделы портфолио: - **Главная /** – краткое приветствие, обзор. - **Проекты /projects** – список проектов. При наличии детальных страниц – вложенный маршрут `/projects/[slug]` для каждого проекта. - **Навыки /skills** – страница (или раздел на главной) со списком категорий навыков (может генерироваться из `skills.md`). - **Связаться /contact** – либо отдельная страница с контактной формой (mailto-ссылкой), либо просто секция на главной. Но роут `/api/contact` зарезервирован для backend обработки формы. - **Дополнительно** – возможно, страница «О себе», блог или другие разделы, если планируется.

Все маршруты настроены для корректного SEO: человекопонятные URL (например, `/projects/my-project`), генерация sitemap.xml, настроены теги `<title>` и мета-описания на каждую страницу через объект `PageMeta` (см. модель данных), либо с помощью `metadata` API Next.js App Router.

Структура папок

Ниже представлена предлагаемая структура файлов и папок проекта (основные элементы):

```
/app                  # Next.js App Router pages
├── layout.tsx        # Главный макет (навигация, темы, метатеги)
├── page.tsx          # Главная страница портфолио
├── projects/         # Раздел проектов
│   ├── page.tsx      # Страница со списком всех проектов
│   └── [slug]/       # Динамические страницы проектов
│       └── page.tsx  # Шаблон страницы отдельного проекта
├── skills/
│   └── page.tsx      # Страница со списком навыков (если не на главной)
├── api/
│   └── contact/
│       └── route.ts  # API-эндпоинт для формы контактов (POST /api/contact)
└── ... (другие разделы при необходимости, например blog/)
```

```
/components            # Переиспользуемые React-компоненты
├── ui/                # Каталог компонентов shadcn/ui (atomic primitives)
│   ├── button.tsx     # Пример: кнопка
│   ├── dialog.tsx     # Пример: диалоговое окно
│   └── ...            # и др. (input, card, navbar и т.п.)
├── Header.tsx         # Общий хедер
├── Footer.tsx         # Общий футер
├── ProjectCard.tsx    # Компонент карточки проекта
└── ...                # и др. компоненты
/content               # Контентные файлы (данные портфолио)
├── projects/          # MDX-файлы с описанием проектов (один файл – один
проект)
│   ├── project1.mdx
│   ├── project2.mdx
│   └── ...
├── skills.md          # Описание навыков по категориям (для генерации
страницы навыков)
├── skills.json        # (Вариант) те же навыки в JSON, если нужно в коде
└── ...                # другие данные (например, блог-посты, если есть)
/public                # Публичные ассеты
├── images/            # Изображения (проекты, фото иконки и др.)
├── fonts/             # Локальные шрифты (Geist Sans/Mono, Inter) если нужны
локально
└── favicon.ico
/styles                # Стили проекта
├── globals.css        # Глобальные стили, импорты Tailwind CSS
├── theme.css          # CSS-переменные для темизации (цвета для light/dark)
└── ...                # возможно, дополнительные файлы (напр.
typography.css)
/utils (или /lib)      # Утилиты и вспомогательный код
├── validation.ts      # Функции/схемы валидации (например, для
ContactMessage)
├── apiClient.ts       # Клиент для внешних API (если понадобится, например
отправка email)
└── constants.ts       # Константы (например, email получателя формы, имя
сайта и пр.)
.next.config.js        # Конфигурация Next.js (настройка MDX плагина, alias и
пр.)
tailwind.config.js     # Конфигурация Tailwind (цвета, шрифты, breakpoints,
темы)
tsconfig.json          # Конфигурация TypeScript (включая пути к папкам)
AGENTS.md              # (Документация) Описание обязанностей Frontend Agent
backlog.md             # (Документация) Беклог задач, включая планы по /api/
contact
README.md              # Общая информация о проекте, запуск, деплой
```

(Примечание: В варианте с Pages Router структура чуть отличается – папка `pages/` *вместо* `app/`*, и API-роут находится в* `pages/api/contact.ts`*. Остальное – аналогично.)*

Объяснение структуры: Файлы сгруппированы по предназначению. Например, в `components/ui` лежат низкоуровневые UI-элементы (кнопки, поля ввода) – они получены из shadcn/ui и не зависят от бизнес-логики. В корне `components` – более сложные компоненты, собирающие UI вместе (например, `Header` включает логотип и меню). Папка `content` отделяет данные от кода – это упрощает замену источника данных (например, переход на CMS) в будущем, и четко разделяет ответственность: разработка структуры vs. наполнение контентом. Папка `styles` содержит определение дизайна: благодаря Tailwind здесь можно хранить design tokens (через CSS-переменные и настройки Tailwind). Файлы конфигурации (.config.js) держатся в корне репозитория. Документы `AGENTS.md`, `backlog.md`, `skills.md` – не участвуют напрямую в работе приложения, но служат документацией и контентом для страницы навыков.

Правила именования

Единые соглашения по именованию упрощают навигацию в коде: - **Файлы и папки:** названия *в нижнем регистре*, слова разделяются дефисом (kebab-case), например `project-card.tsx`, `contact-form.tsx`. Исключение – файлы, содержащие один React-компонент, можно называть с большой буквы (PascalCase) по имени компонента, например `Header.tsx`, чтобы сразу было видно соответствие. Папки – только lowercase (например, `components`, `utils`, `images`). - **Компоненты React:** именуются в PascalCase (например, `function ProjectCard() {}` в файле `ProjectCard.tsx`). Имена должны отражать сущность компонента. Примеры: `Header`, `Footer`, `ProjectList`. Если компонент предназначен быть **клиентским** (использует состояние, эффекты), в Next 13 App Router в начале файла добавляется директива `'use client'`. - **Типы и интерфейсы (TypeScript):** именуются в PascalCase, обычно существительное, отражающее суть структуры данных – напр. `Project`, `Skill`, `ContactMessage` (см. следующий раздел). Если нужен тип для набора литералов (например, категории навыков), его можно назвать в единственном числе во множественном числе (напр. `SkillCategory` для union-типов категорий). - **Переменные и функции:** camelCase нотация. Переменные – существительные или существительное с прилагательным (`projectList`, `filteredProjects`). Функции – глагол или глагольная фраза (`fetchProjects`, `handleSubmit`). Для React-хуков: префикс `use` (`useTheme`, `useContactForm`). - **Константы:** UPPER_SNAKE_CASE, например `DEFAULT_LANG`, `MAX_MESSAGE_LENGTH`. - **Коммиты и ветки (опционально):** можно придерживаться conventional commits (`feat: ...`, `fix: ...`) и именовать ветки по функциональности (`feature/add-contact-endpoint`), хотя это организационный момент вне кода. - **Прочее:** именование должен быть самодокументирующим. Например, файл с утилитами для валидации `validation.ts`, конфиг для почты `email.config.ts` и т.п. Стараемся избегать аббревиатур без необходимости. Если используются интерфейсы для пропсов компонентов, можно именовать их с суффиксом Props (например, `HeaderProps`).

Следуя этим правилам, кодовая база остается читаемой: сразу понятно, где искать определение компонента, что представляет та или иная переменная, и как устроены данные.

Модель данных

Определим ключевые TypeScript-типы, описывающие данные портфолио. Эти интерфейсы обеспечивают типобезопасность при работе с контентом и документацию структуры данных.

```
// Проект (портфолио элемент)
interface Project {
  id: string            // Уникальный идентификатор проекта (например, слаг)
```

```
  title: string          // Название проекта
  description: string    // Описание проекта (краткое)
  details?: string       // Развернутое описание (может быть в MDX)
  tags?: string[]        // Теги или категории проекта (напр. ['React', 'UI'])
  skills?: string[]      // Ключевые навыки, использованные в проекте
(ссылаются на Skill.name)
  links: Link[]          // Связанные ссылки (демо, репозиторий и пр.)
  image?: string         // Путь к изображению проекта (скриншот)
}

// Навык / умение
interface Skill {
  id: string             // Уникальный идентификатор или ключ навыка (слэг)
  name: string           // Название навыка (например, "React")
  category: string       // Категория навыка (например, "Frontend", "Design")
  level?: string         // Опционально: уровень владения или опыт (например,
"advanced")
}

// Сообщение из формы контакта
interface ContactMessage {
  name: string           // Имя отправителя
  email: string          // Email отправителя
  message: string        // Текст сообщения
  honeypot?: string      // Поле-ловушка для ботов (должно оставаться пустым)
  timestamp?: Date       // Время отправки (устанавливается на сервере)
}

// Мета-данные страницы (для SEO и оглавления)
interface PageMeta {
  title: string          // Заголовок страницы (тег <title>)
  description?: string   // Описание страницы (meta description)
  slug?: string          // URL slug страницы (если генерируется на основе
контента)
  tags?: string[]        // Опционально: теги или ключевые слова для SEO
}

// Ссылка (для навигации или внешних ресурсов)
interface Link {
  label: string          // Текст ссылки или название
  url: string            // URL адрес
  icon?: string          // Опционально: имя иконки (если иконки отображаются
рядом с ссылкой)
}
```

(Примечание: Эти типы могут дополняться. Например, можно добавить `ProjectCategory` *или расширить* `Project` *полями вроде* `year` *(год выполнения проекта). В текущем минимальном варианте включены только необходимые свойства.)*

Модель данных отражает структуру контента: - **Project** – используется для рендеринга списка проектов и страниц проектов. Например, список проектов (`projects.json` или несколько MDX

файлов) парсится в массив `Project[]`. Каждому проекту соответствует отдельная страница (маршрут `/projects/[id]`), где `id` совпадает со `slug`. Поле `details` может хранить контент в MDX (например, импортируя MDX-файл проекта). - **Skill** – описывает навык или технологию. Может использоваться для генерации раздела "Навыки": сначала загружаем список всех навыков, группируем их по `category` и отображаем. - **ContactMessage** – структура данных, которую ожидает бэкенд `/api/contact`. Собирается из полей формы обратной связи. Включает `honeypot` (скрытое поле, которое пользователь не заполняет; если заполнено – значит бот). `timestamp` может добавляться на сервере при получении. - **PageMeta** – метаданные для страниц. Можно хранить вместе с содержанием (напр. в MDX-файлах в виде frontmatter YAML) и использовать для установки `head` (заголовок, описание, OpenGraph-мета). Это улучшает SEO и контроль над отображением ссылки в соцсетях. - **Link** – универсальная структура для ссылок. Применяется в навигации (название раздела + URL) или в списке ссылок проекта (например, `{ label: 'Demo', url: 'https://...', icon: 'external-link' }`). Иконка – условное поле, если нужно визуально отмечать ссылки.

Все эти интерфейсы хранятся, например, в файле `types.ts` или распределены по модулям (проекты, навыки и т.д.). TypeScript будет проверять соответствие данных этим типам, предотвращая ошибки несоответствия (например, отсутствие обязательных полей).

Минимальный backend для `/api/contact`

Endpoint `/api/contact` предназначен для обработки отправки контактной формы (если в будущем решено уйти от простого `mailto:` к полноценной форме). Сейчас он находится в бэкенде "про запас" и реализует базовые функции: прием данных, валидация, антиспам и логирование. Ниже описана структура этого эндпоинта:

1. **Маршрут и метод:** `POST /api/contact` – принимаются только POST-запросы с данными формы в формате JSON. Если прилетает GET или другой метод – возвращается 405 Method Not Allowed.
2. **Поля запроса:** `name`, `email`, `message` – обязательные поля (соответствуют интерфейсу `ContactMessage`). Также ожидается скрытое поле `honeypot` (должно быть пустым). Пример тела запроса:

```
{
  "name": "Ivan Petrov",
  "email": "ivan@example.com",
  "message": "Здравствуйте, хочу сотрудничать...",
  "honeypot": ""
}
```

3. **Валидация входных данных:** На сервере проверяются:
4. Заполненность обязательных полей (имя, email, сообщение не пустые).
5. Корректность email (простая проверка формата или использование готовой библиотеки/регэкспа).
6. Длина сообщения (например, не более 1000 символов, чтобы избежать очень длинных спам-сообщений).
7. Поле `honeypot` *должно* быть пустым: если нет – запрос классифицируется как бот-спам и обрабатывается особым образом (см. ниже).
8. **Анти-спам меры:**

9. **Honeypot:** как описано, скрытое поле (например, `<input type="text" name="company" style="display:none">`) не видно человеку, но будет заполнено ботами. Если `honeypot` поле не пустое, сервер может сразу вернуть успех (имитируя обработку) и не выполнять никаких действий (таким образом, бот думает, что сообщение отправлено, а мы его просто игнорируем).
10. **Rate limiting:** вводится ограничение на частоту запросов с одного IP. Например, не более 5 запросов в минуту. Для реализации можно использовать in-memory хранилище (в рамках Vercel Function – недолговечно) или внешний сервис. Минимально – хранить в глобальной переменной список последних обращений по IP с метками времени и отклонять при превышении. В продакшене лучше подключить Redis или Upstash для хранения счетчиков запросов. Но в нашем минимальном бэкенде можно просто вставить заглушку: считать количество обращений в рамках single run (понимая, что на Vercel функция без состояния между вызовами; более надежно – интеграция с **Vercel Rate Limits** либо CAPTCHA).
11. **Дополнительно:** можно предусмотреть проверку peCAPTCHA (Google reCAPTCHA v3 или hCaptcha) – для этого форма при отправке должна отправлять токен, а бэкенд проверять его на валидность. Это сложнее, поэтому пока опущено.
12. **Обработка запроса:**
13. Если валидация не пройдена (пустое поле, email некорректен и т.п.) – возвращается ответ с ошибкой 400 Bad Request с описанием проблемы (или generic "Invalid input").
14. Если запрос признан спамом (honeypot заполнен или слишком много запросов) – возвращается **200 OK** *как будто* успешно (или 204 No Content), но по факту письмо не отправляется. Это обманывает простых ботов и снижает нагрузку.
15. Если все хорошо, **в текущей минимальной реализации**: можно просто залогировать сообщение в консоль (на Vercel логи функции) или отправить в очередь. В production-варианте здесь происходит отправка письма владельцу портфолио:
    - Либо через встроенный email-сервер (например, с помощью `nodemailer` + SMTP, требующий настроек в .env: SMTP_HOST, SMTP_USER, SMTP_PASS).
    - Либо через внешнее API (например, SendGrid, Resend, AWS SES) – тогда в .env хранится API-ключ, и запрос к API осуществляется из функции.
    - Сообщение может включать тему (например, "New contact message from Portfolio"), содержать имя, email и текст, и отправляться на личную почту владельца.
16. После попытки отправки (или логирования) возвращается клиенту JSON-ответ, например `{ success: true }` или сообщение об ошибке. В случае успеха фронтенд может показать пользователю нотификацию "Ваше сообщение отправлено".
17. **Безопасность:** Endpoint не требует аутентификации (он публичный), поэтому важна вышеперечисленная защита. Также сервер должен убедиться, что вызывается с нашего сайта: можно проверить заголовок Origin/Referer, хотя это не очень надежно (но поможет отсечь некоторые запросы). В целом, ничего критичного этот endpoint не выдает – лишь принимает сообщения, поэтому угроз невелико, главное – не позволить злоупотреблять (spam/DDoS).
18. **Размещение и структура:** В App Router файл расположен по пути `app/api/contact/route.ts` и экспортирует функции `GET`, `POST` и др. (нам нужен только `POST`). В Pages Router эквивалент – файл `pages/api/contact.ts`, экспортирующий функцию-хендлер `(req, res) => {...}`. В коде организуем его аккуратно: сначала проверка метода, затем парсинг `req.body` (нужен `JSON.parse` если raw, но Next API automatically parses JSON), валидация, и затем логика отправки. Можно вынести логику валидации и отправки в утилиты (например, `utils/validation.ts` для схемы ContactMessage и `utils/mailer.ts` для функции sendEmail).

Таким образом, даже в минимальном виде бэкенд готов к боевой работе – остается только подключить реальную отправку почты и при необходимости улучшить антиспам. Пользователь

же на фронтенде пока взаимодействует с mailto (почтовым клиентом), но при масштабировании можно переключить форму на AJAX-запрос к этому endpoint'у без значительных изменений в инфраструктуре.

План деплоя

Развертывание портфолио планируется на **Vercel** – это оптимальный выбор для Next.js приложений, так как Vercel непосредственно поддерживает фреймворк (автор Next.js) и обеспечивает глобальное распространение контента.

- **Хостинг на Vercel:** Репозиторий (например, на GitHub) подключается к Vercel. Каждый push в основную ветку инициирует автоматический деплой. Приложение разворачивается на глобальной CDN Vercel, поэтому статические страницы кешируются по миру и отдаются пользователям с ближайшего региона [1]. Нет необходимости вручную настраивать сервер – Vercel предоставляет Serverless Functions для API (наш `/api/contact` будет выполняться как функция).
- **Настройки домена:** По умолчанию проект будет доступен по адресу вида `my-portfolio.vercel.app`. Можно прикрепить собственный домен через настройки Vercel (DNS конфигурация).
- **Environment Variables (.env):** В корне проекта используется файл `.env.local` для локальной разработки, где хранятся переменные окружения (не в репозитории). На Vercel эти переменные задаются через dashboard проекта (раздел **Settings -> Environment Variables**). Для данного проекта это может включать:
- `NEXT_PUBLIC_BASE_URL` – базовый URL сайта (для генерации ссылок, sitemap, и т.д.).
- `CONTACT_EMAIL_TO` – email получателя для контактной формы (если реализуется отправка).
- `SMTP_HOST`, `SMTP_USER`, `SMTP_PASS` – если настраивается прямое SMTP соединение.
- API ключи сторонних сервисов, если будут (например, аналитика, капча).
- **Важно:** префикс `NEXT_PUBLIC_` указывает, что переменная попадет в фронтенд-бандл (например, ID отслеживания аналитики). Чувствительные данные (пароли, ключи) без этого префикса доступны только на сервере (в наших функциях).
- **CI/CD и проверки:** Vercel автоматически запускает билды; стоит настроить скрипты `build` и `lint` в package.json. Хорошей практикой будет включить линтер и типизационные проверки в CI: Vercel перед деплоем может запускать `npm run build`, который упадет при ошибках. Опционально, можно подключить GitHub Actions для дополнительных проверок, но для простоты Vercel CI достаточно.
- **Мониторинг и аналитика:** Для минимального мониторинга подключаем **Vercel Web Analytics** – это встроенная в Vercel легковесная аналитика без использования cookie, которая покажет базовые метрики посещаемости (кол-во посетителей, просмотры страниц). Она не требует дополнительного кода – достаточно включить ее в настройках проекта на Vercel и вставить `<script>` тег, который Vercel сгенерирует. Альтернативы: Google Analytics (Universal Analytics или GA4) – более мощный инструмент, но требующий баннера согласия (GDPR) и тяжелее; либо простая open-source аналитика вроде Umami или Fathom – при необходимости.
- В нашем случае, **Vercel Analytics** достаточно: в `.env` добавится `NEXT_PUBLIC_VERCEL_ANALYTICS_ID`, и в файл `_app.tsx` (если Pages Router) или в `layout.tsx` (App Router) – скрипт `import { Analytics } from '@vercel/analytics/react';` и `<Analytics />` компонент.
- **Логи и ошибки:** Vercel предоставляет просмотр логов функции (для `/api/contact`) прямо на сайте. Дополнительно, можно интегрировать Sentry или аналог для

отслеживания ошибок runtime в клиентской части. В минимальном варианте можно обойтись без этого, полагаясь на консоль в браузере для отладки и логи Vercel для серверных ошибок.
- **Бэкап и откат:** Vercel хранит историю деплоев, можно быстро откатиться к предыдущему успешному билду, если что-то пошло не так. Также рекомендуется периодически экспортировать контент (наш контент и так в Git, поэтому бэкапы – это просто commits).

Резюме деплоя: Вы берете готовый репозиторий, нажимаете "Deploy to Vercel", прописываете переменные окружения (если нужны, например email), и спустя пару минут портфолио доступно онлайн. Все статические страницы раздаются максимально быстро благодаря CDN, а серверные функции масштабируются автоматически. Мониторинг встроен, поддержка Next.js на высшем уровне.

Структура файлов документации

Проект содержит несколько вспомогательных файлов документации/контента. Они оформлены в Markdown и служат как для разработки, так и для вывода части информации на сайт.

AGENTS.md

Файл `AGENTS.md` описывает, что должен уметь **Frontend Agent** – условно, "агент" фронтенда, то есть фронтенд-приложение или разработчик, отвечающий за него. Здесь перечислены все основные компетенции фронтенда в контексте данного портфолио: реализация UI, маршрутизация, SEO, доступность и др. Этот файл может использоваться как чек-лист задач для фронтенда или просто как документирование архитектурных требований.

Содержимое `AGENTS.md`**:**

```
# Frontend Agent

**Ответственности и возможности фронтенда:**

- **UI и дизайн системы** – Реализует пользовательский интерфейс в соответствии с дизайн-системой. Использует принципы Swiss Style (строгая сетка, аккуратные отступы, минимализм) [7]. Применяет шрифты Inter или Geist Sans для основного текста и Geist Mono для технических деталей. Обеспечивает единый стиль на всех страницах, переиспользуя компоненты дизайн-системы.
- **Маршрутизация и страницы** – Настраивает маршруты приложения (Next.js routing), создаёт страницы портфолио (главная, проекты, навыки, контакт). Гарантирует, что навигация интуитивно понятна, ссылки работают корректно (включая переходы между страницами без перезагрузки). В случае Next App Router – организует layout'ы для общих частей (header, footer).
- **SEO-оптимизация** – Управляет мета-данными страниц. Для каждой страницы задаёт уникальный `<title>`, описание, Open Graph и Twitter Card теги для красивого шаринга в соцсетях. Генерирует sitemap.xml и `robots.txt` для поисковых систем [9]. Следит, чтобы контент (особенно текст в тегах H1, alt у изображений) был индексируемым и отражал ключевые навыки и проекты.
- **Accessibility (a11y)** – Обеспечивает доступность интерфейса для всех пользователей. Использует семантически правильные HTML-элементы и атрибуты ARIA там, где нужно. Компоненты из shadcn/ui (на базе Radix) дают базовую
```

```
гарантию доступности (управление фокусом, клавиатурная навигация) [4], дополнительно проверяется контрастность цветов, масштабируемость текста. Навигация доступна с клавиатуры, интерактивные элементы имеют фокус-стили. Проводится тестирование с инструментами типа Lighthouse или axe-core на соответствие WCAG 2.1.
- **Производительность и оптимизация** – (Опционально) Следит за размером бандла и скоростью загрузки. Использует динамический импорт для тяжёлых частей, оптимизирует изображения (Next.js Image компонент или ручная оптимизация). Шрифты подключает через `@next/font` (оптимизированный способ загрузки шрифтов). Мониторит Web Vitals (LCP, FID, CLS) с помощью встроенной аналитики или сторонних инструментов, при необходимости улучшает показатели (например, уменьшая объем JS, используя кеширование данных).
- **Лучшие практики фронтенда** – Поддерживает читабельность и чистоту кода. Придерживается соглашений по структуре и стилю кода (см. правила именования). Реализует адаптивность (responsive design) для разных устройств: от мобильных до десктопов. Проверяет кросс-браузерную совместимость (Chrome, Firefox, Safari, Edge). Настраивает инструменты сборки и деплоя (CI/CD) совместно с backend agent (если тот имеется). Документирует компоненты и утилиты при необходимости (например, в Storybook или в виде комментариев в коде).
```

(Файл `AGENTS.md` не участвует напрямую в коде приложения, но может служить ориентиром для разработки и документирования. При желании, отдельные разделы этого документа можно вывести на сайт, например, как часть страницы «О проекте».)

skills.md

Файл `skills.md` содержит список технологий, инструментов и навыков, сгруппированных по категориям. Этот файл играет роль контентного хранилища для страницы "Навыки" – его можно парсить и отображать на сайте, либо просто поддерживать как документ для читателя. В отличие от `skills.json` (машинописного), `skills.md` удобен для быстрого редактирования и визуального представления категорий.

Содержимое `skills.md`:

```
# Навыки и стек

**Языки и фреймворки:**
- **TypeScript / JavaScript** – современный стандарт веб-разработки; используется TypeScript для надежности кода.
- **React** – основа фронтенда, библиотека для построения UI.
- **Next.js** – фреймворк на базе React для серверного рендеринга и статической генерации; используется для маршрутизации и структуры проекта.

**UI/Дизайн:**
- **Tailwind CSS** – утилитарный CSS-фреймворк для быстрого и единообразного стилирования.
- **shadcn/ui (Radix UI)** – набор готовых UI-компонентов (модальные окна, выпадающие меню, формы и др.) с акцентом на доступность [4]. Позволяет быстро реализовать интерфейс, соответствующий дизайн-системе.
```

```
- **Design System & Swiss
Style** – собственная дизайн-система, основанная на принципах швейцарского
дизайна: модульная сетка, минимализм, акцент на типографике. Используются
фирменные шрифты (Inter или Geist Sans, моноширинный Geist Mono) для
консистентности с Vercel Design System.
- **Adaptive & Responsive Design** – опыт разработки адаптивного интерфейса,
обеспечивающего удобство на мобильных, планшетах и десктопах. Использование
Flexbox/Grid, медиазапросов Tailwind для реализации responsive-правил.
- **Accessibility (a11y)** – знание и применение принципов доступности:
семантичная разметка, правильные роли и aria-метки, контраст цветов,
доступность без мыши.

**Контент и разметка:**
- **MDX (Markdown + JSX)** – используемый формат для контента. Позволяет
писать статьи/страницы в Markdown, расширяя JSX-компонентами (для вставки
динамических элементов).
- **HTML5, CSS3** – базовые технологии веба. Семантичный HTML, современный
CSS (Grid, переменные, анимации) для реализации макета и стилей.
- **JSON/
YAML** – форматы для структурированных данных. В проекте применяются для
хранения списков (навыки, проекты) и метаданных (например, frontmatter у
MDX).

**Инфраструктура и инструменты:**
- **Vercel** – платформа для деплоя приложений. Используется для хостинга
портфолио, обеспечивает CDN, serverless-функции и аналитику.
- **Git и GitHub** – система контроля версий и платформа для хостинга кода.
Проект версионируется с помощью Git; GitHub – для совместной работы и
интеграции с Vercel CI.
- **Node.js (Express)** – серверные технологии. В контексте портфолио –
minimal backend на Next.js API Routes (функционал Node.js). Опыт работы с
Node позволяет настраивать серверную логику (например, отправка email через
nodemailer, использование внешних API).
- **ESLint, Prettier** – инструменты статического анализа и форматирования
кода. Настроены в проекте для поддержания единого стиля и отлавливания ошибок
на этапе разработки.
- **Jest, React Testing Library** – (опционально) инструменты для
тестирования. Могут использоваться для модульного тестирования компонентов и
утилит, что повышает надежность при внесении изменений.
- **Figma / Дизайн-инструменты** – навыки работы с дизайн-макетами. Хотя
дизайн задуман в коде по принципам Swiss Style, умение пользоваться Figma
помогает при интеграции макетов или совместной работе с дизайнерами.
```

Обратите внимание, как категории в `skills.md` соответствуют различным аспектам стека разработчика. Такой файл может быть отображен на сайте в виде секций со списками (например, заголовок категории и иконки или названия технологий под ним). Он легко расширяем: можно добавить новые категории (например, “*Backend*” или “*DevOps*”) при появлении новых навыков.

Все вышеперечисленные элементы (стек, архитектура, модели, backend, деплой, документация) образуют целостный план разработки фронтенд-портфолио. Соблюдение указанной структуры и соглашений позволит быстро стартовать с вариантом A и постепенно улучшать до варианта B, демонстрируя высокий уровень качества и профессионализма проекта. Каждый раздел документа хорошо структурирован и готов к применению в реальном коде, что облегчит процесс реализации.

---

1 2 9 App Router: Static and Dynamic Rendering | Next.js
https://nextjs.org/learn/dashboard-app/static-and-dynamic-rendering

3 4 Frontend Handbook | React / Tailwind / Shadcn
https://infinum.com/handbook/frontend/react/tailwind/shadcn

5 React Templates - Portfolio
https://www.shadcn.io/template/category/portfolio

6 App Router vs Pages Router in Next.js — a deep, practical guide
https://dev.to/shyam0118/app-router-vs-pages-router-in-nextjs-a-deep-practical-guide-341g

7 GitHub - vercel/geist-font
https://github.com/vercel/geist-font

8 Posts | Blogging with MDX in Next.js - Jonah Grimes
https://www.nerdynarwhal.com/posts/2024-01-07-blogging-with-mdx-in-nextjs