# 

**Документация**

**по проект на тема**

**„Personal Cloud“**

**Web технологии,**

**учебна година 2024/2025**

**Екип №7**

Александър Тагамлицки - 2MI0700171  
Иван Стойнев - 6MI0700164

Десислава Добрева - 0MI0700129

Поли Янева - 2MI0700143

Севда Иванова - 3MI0700109

# Архитектура на проекта

PersonalCloudProject/

├── public/ # Статични файлове за визуализация

│ ├── index.html # Главна HTML страница

│ ├── index.js # JavaScript логика за клиента (ако има)

│ └── style.css # CSS стилове за потребителския интерфейс

├── src/ # Основна логика на приложението (backend)

│ ├── app.ts # Инициализация на Express приложението

│ ├── server.ts # Стартиране на сървъра и връзка с MongoDB

│ ├── controllers/ # Контролери с бизнес логика

│ │ ├── fileController.ts # Операции с файлове и директории

│ │ ├── userController.ts # Регистрация, вход, профил и др.

│ │ └── pageController.ts # Отговаря за визуализиране на страници

│ ├── routes/ # Дефинирани API маршрути

│ │ ├── authRoutes.ts # Регистрация, вход, изход

│ │ └── fileRoutes.ts # Работа с файлове и директории

│ ├── models/ # Mongoose модели

│ │ ├── userModel.ts # Модел на потребител

│ │ ├── fileModel.ts # Модел на файл/директория

│ │ └── storageQuotaModel.ts # Модел за следене на използвано дисково пространство

│ ├── db/ # Връзка и инициализация на базата данни

│ │ ├── connection.ts # Свързване с MongoDB чрез Mongoose

│ │ └── dbInit.ts # Начална конфигурация (напр. индекси)

│ ├── config/ # Глобални конфигурации

│ │ └── db.ts # Конфигурации за връзка с базата данни

│ ├── middlewares/ # Middleware функции

│ │ └── errorHandler.ts # Проверка за автентикация и обработка на грешки

│ ├── utils/ # Помощни функции

│ │ └── fileSearch.ts

# 

# server.ts

Път: [server.ts](http://server.ts)

## Описание:

Файлът server.ts е основната точка за стартиране на сървъра на приложението. Той конфигурира Express.js, установява връзка с базата данни и стартира HTTP сървър, който слуша на зададен порт. Основната му цел е да осигури инфраструктура за обработка на HTTP заявки и да гарантира, че базата данни е свързана преди стартиране на сървъра.

## Детайлно описание на компонентите:

### Импорти:

* express: Основната библиотека за създаване на HTTP сървър.
* connectDB: Функция за връзка с MongoDB, дефинирана в [connection.ts](http://connection.ts).
* dotenv: Зарежда променливи на средата от .env файл.

Роля:

* express се използва за създаване на сървъра и обработка на HTTP заявки.
* connectDB гарантира, че приложението е свързано с базата данни преди стартиране.
* dotenv позволява зареждане на конфигурационни параметри от .env файл.

### Зареждане на конфигурация:

Зарежда променливите на средата чрез dotenv.config(). Това позволява използването на конфигурационни параметри като PORT и DB\_CONN\_STRING.

Роля:

* Осигурява достъп до конфигурационни параметри, които са дефинирани в .env файл.
* Например: PORT за задаване на порт за сървъра и DB\_CONN\_STRING за връзка с базата данни.

### Инициализация на Express приложението:

Създава инстанция на Express приложението чрез const app = express().

Задава порт за слушане, като използва стойността от .env или 5001 по подразбиране.

Роля:

* app е основният обект на Express приложението, който обработва HTTP заявки.
* PORT дефинира порта, на който сървърът ще слуша.

### Middleware за обработка на заявки:

* app.use(express.json()): Позволява обработка на JSON заявки.
* app.use(express.urlencoded({ extended: true })): Позволява обработка на URL-кодирани заявки.

Роля:

* Осигурява обработка на входящи данни от клиента.
* express.json() обработва JSON заявки, а express.urlencoded() обработва заявки с URL-кодирани данни.

### Връзка с базата данни:

* Използва функцията connectDB за установяване на връзка с MongoDB.
* При успех логва съобщение за успешна връзка.
* При неуспех логва грешка и прекратява процеса.

Роля:

Гарантира, че приложението е свързано с базата данни преди стартиране.

При грешка прекратява процеса, за да се избегнат проблеми с липсваща връзка.

### Основен маршрут:

Дефинира основен GET маршрут, който връща съобщение за работещ API.

Роля:

Осигурява основен тестов маршрут, който показва, че сървърът работи.

### Стартиране на сървъра:

### Стартира HTTP сървър, който слуша на зададения порт.

* Логва съобщение с URL адреса на сървъра.

Роля:

* Започва слушане на заявки от клиенти.
* Логва информация за порта, на който сървърът работи.

### Примерен сценарий:

1. Стартира се файлът server.ts.
2. Зареждат се променливите на средата.
3. Установява се връзка с MongoDB.
4. Стартира се HTTP сървър на зададения порт.
5. Клиентите могат да изпращат заявки към API.

# app.ts

Път: app.ts

## Описание:

Файлът app.ts дефинира основната конфигурация на Express приложението. Той включва маршрути, статични файлове и middleware за обработка на заявки. Основната му цел е да предостави готово приложение, което може да бъде монтирано към сървъра.

## Детайлно описание на компонентите:

### Импорти:

* express: Основната библиотека за създаване на HTTP сървър.
* fileRoutes: Маршрути за работа с файлове, дефинирани в fileRoutes.js.
* path: Модул за работа с файлови пътища.
* fileURLToPath: Позволява конвертиране на URL към файлов път.

Роля:

Осигурява основните зависимости за конфигуриране на приложението.

### Инициализация на Express приложението:

* Създава инстанция на Express приложението чрез const app = express().
* Дефинира \_\_dirname за работа с файлови пътища.

Роля:

* app е основният обект на Express приложението.
* \_\_dirname се използва за задаване на пътища към статични файлове.

### Middleware за обработка на заявки:

* app.use(express.json()): Позволява обработка на JSON заявки.
* app.use(express.urlencoded({ extended: true })): Позволява обработка на URL-кодирани заявки.

Роля:

Осигурява обработка на входящи данни от клиента.

### Статични файлове:

* Дефинира директории за статични файлове:
* public: Съдържа публични ресурси като изображения, CSS и JavaScript.
* dist: Съдържа компилирани файлове от фронтенд приложението.

Роля:

Осигурява достъп до статични ресурси.

### Маршрути за файлове:

Монтира маршрути за работа с файлове под /api/files.

Роля:

Осигурява API за работа с файлове.

### Експорт:

### Експортира конфигурираното приложение за използване в други файлове, като например server.ts.

Роля:

Позволява приложението да бъде използвано от сървъра.

## Примерен сценарий:

* Файлът app.ts се импортира в server.ts.
* Конфигурираното приложение се монтира към сървъра.
* Клиентите могат да изпращат заявки към API, включително маршрути за файлове.

## Връзки между файловете

* server.ts:

Стартира сървъра и установява връзка с базата данни.

Импортира и използва конфигурираното приложение от app.ts.

* app.ts:

Конфигурира маршрути, статични файлове и middleware.

Експортира приложението за използване в server.ts.

## Предимства на архитектурата

* Модулност:

Разделя конфигурацията на приложението (app.ts) от стартирането на сървъра (server.ts).

* Гъвкавост:

Лесно добавяне на нови маршрути и middleware.

* Скалируемост:

Позволява разширяване на функционалността чрез добавяне на нови модули.

* Сигурност:

Зарежда конфигурация от .env, което предпазва чувствителни данни като портове и връзки към базата данни.

## Примерен работен поток

* Инициализация на приложението:

app.ts конфигурира маршрути, статични файлове и middleware.

* Стартиране на сървъра:

server.ts установява връзка с базата данни и стартира HTTP сървъра.

* Обработка на заявки:

Клиентите изпращат заявки към API, които се обработват от маршрути в app.ts.

# 

# **userModel.ts**

## Описание:

Файлът дефинира схемата и модела за потребителите в базата данни. Моделът представлява структурата на потребителския документ в MongoDB и включва основни данни за идентификация, автентикация и управление на потребителите.

## Интерфейс: IUser

Интерфейсът дефинира типовете данни за потребителския документ:

* **firstName**: Име на потребителя (тип: String, задължително).
* **lastName**: Фамилия на потребителя (тип: String, задължително).
* **username**: Уникално потребителско име (тип: String, задължително).
* **email**: Уникален имейл адрес (тип: String, задължително).
* **password**: Хеширана парола (тип: String, задължително).
* **lastLogin**: Дата на последно влизане (тип: Date, незадължително).

## Схема: UserSchema

Схемата дефинира структурата на потребителския документ:

* **firstName**: Задължително поле от тип String.
* **lastName**: Задължително поле от тип String.
* **username**: Задължително и уникално поле от тип String.
* **email**: Задължително и уникално поле от тип String.
* **password**: Задължително поле от тип String.
* **lastLogin**: Незадължително поле от тип Date.

## Опции:

* **timestamps**: Автоматично добавя полета createdAt и updatedAt към документа.

## Индекси:

* **email**: Индекс за бързо търсене по имейл.
* **username**: Индекс за бързо търсене по потребителско име.

## Експорт:

Моделът се експортира като User и е свързан със схемата UserSchema.

# **storageQuotaModel.ts**

## Описание:

Файлът дефинира схемата и модела за квотите за съхранение на потребителите. Моделът представлява структурата на квотните документи в MongoDB и включва данни за управление на пространството за съхранение.

## Интерфейс: IStorageQuota

Интерфейсът дефинира типовете данни за квотния документ:

* **userId**: ID на потребителя (тип: mongoose.Types.ObjectId, задължително, уникално).
* **totalSpace**: Общото пространство за съхранение (тип: number, задължително, по подразбиране: 15 GB).
* **usedSpace**: Използвано пространство (тип: number, задължително, по подразбиране: 0).
* **plan**: План за съхранение (тип: string, стойности: free, premium, business, по подразбиране: free).

## Схема: StorageQuotaSchema

Схемата дефинира структурата на квотния документ:

* **userId**: Уникално поле, което свързва квотата с потребителя.
* **totalSpace**: Задължително поле, което дефинира общото пространство за съхранение.
* **usedSpace**: Задължително поле, което дефинира използваното пространство.
* **plan**: Поле, което дефинира плана за съхранение (например: free, premium, business).

## Опции:

* **timestamps**: Автоматично добавя полета createdAt и updatedAt към документа.

## Индекси:

* **userId**: Индекс за бързо търсене по потребител.
* **plan**: Индекс за бързо търсене по план.

## Експорт:

Моделът се експортира като StorageQuota и е свързан със схемата StorageQuotaSchema.

# fileModel.ts

## Описание:

Файлът дефинира схемата и модела за файловете и директориите в системата. Моделът представлява структурата на файловите документи в MongoDB и включва данни за управление на файлове, директории и техните свойства.

## Интерфейс: IFile

Интерфейсът дефинира типовете данни за файловия документ:

* **name**: Име на файла/директорията (тип: string, задължително).
* **type**: Тип на елемента (тип: string, задължително, стойности: file или directory).
* **path**: Път до файла/директорията (тип: string, задължително).
* **content**: Съдържание на файла (тип: string, незадължително).
* **mimeType**: MIME тип на файла (тип: string, незадължително).
* **size**: Размер на файла (тип: number, незадължително, по подразбиране: 0).
* **owner**: ID на собственика (тип: mongoose.Types.ObjectId, задължително).
* **parent**: ID на родителската директория (тип: mongoose.Types.ObjectId, незадължително).
* **isPublic**: Дали файлът/директорията е публичен (тип: boolean, по подразбиране: false).
* **lastModified**: Дата на последна промяна (тип: Date, по подразбиране: текуща дата).
* **originalName**: Оригинално име на файла/директорията (тип: string, задължително).
* **storageUrl**: URL за съхранение (тип: string, незадължително).
* **sharedWith**: Списък с потребители, с които е споделен файлът (тип: mongoose.Types.ObjectId[]).
* **starred**: Дали файлът е маркиран като важен (тип: boolean, по подразбиране: false).

## Схема: FileSchema

Схемата дефинира структурата на файловия документ:

* Полетата съответстват на интерфейса IFile.

**Опции:**

* **timestamps**: Автоматично добавя полета createdAt и updatedAt към документа.

## Индекси:

* **owner и path**: Уникален индекс за бързо търсене на файлове по собственик и път.
* **parent**: Индекс за бързо търсене по родителска директория.
* **owner**: Индекс за бързо търсене по собственик.
* **sharedWith**: Индекс за бързо търсене на споделени файлове.
* **starred**: Индекс за бързо търсене на маркирани файлове.

## Експорт:

Моделът се експортира като File и е свързан със схемата FileSchema.

## Връзки между моделите:

1. **User**:
   * Свързан с **StorageQuota** чрез userId.
   * Свързан с **File** чрез owner.
2. **StorageQuota**:
   * Уникално свързан с потребител чрез userId.
3. **File**:
   * Свързан с потребител чрез owner.
   * Свързан с други файлове чрез parent.

## Примерен сценарий:

1. **User**: Създава се нов потребител с уникални данни.
2. **StorageQuota**: За всеки потребител се създава квота за съхранение.
3. **File**: Потребителят може да създава файлове и директории, които се свързват с неговия owner.

# **db.ts**

Път: db.ts

## Описание и предназначение

Този файл дефинира и експортира конфигурационен обект за връзка с MongoDB, както и параметри за квоти за съхранение според различни потребителски планове. Той централизира всички настройки, свързани с базата данни, и гарантира, че приложението няма да стартира без валидни параметри за връзка.

## Основни компоненти

### Зареждане на променливи на средата

Зарежда променливите от .env файл, за да се използват в конфигурацията.

### Интерфейс за опции на Mongoose

Разширява стандартните опции на Mongoose, за да позволи по-детайлна конфигурация.

### Конфигурационен обект dbConfig

* connectionString: Връзка към MongoDB, взета от променливите на средата.
* dbName: Име на базата данни.
* webAppName: Име на приложението (ако се използва).
* options: Опции за връзка с базата данни, включително име на база, нов парсър, унифицирана топология и автоматични индекси.
* storageQuotas: Квоти за съхранение по планове (free, premium, business) в байтове.

### Валидация на конфигурацията

Ако липсва връзка или име на база, процесът се прекратява с грешка.

### Експорт

Експортира конфигурационния обект за използване в други части на приложението.

# **connection.ts**

## Път: connection.ts Описание и предназначение:

Този файл предоставя функция за установяване на връзка с MongoDB чрез Mongoose, използвайки конфигурацията от db.ts. Осигурява обработка на събития, свързани с връзката, и гарантира коректно затваряне на връзката при прекъсване на приложението.

## Основни компоненти:

### 2.1. Импорти и конфигурация

* Импортира mongoose и конфигурация за връзката.

### 2.2. Функция за връзка с базата данни

* Опитва да се свърже с базата данни.
* При успех логва съобщение.
* При неуспех логва грешка и прекратява процеса.

### 2.3. Обработка на събития

* Логва грешки и прекъсвания на връзката.

### 2.4. Обработка на прекъсване на приложението

* При прекъсване на процеса (Ctrl+C), затваря връзката и излиза чисто.

### 2.5. Експорт

* Експортира функцията за връзка.

## 

# dbInit.ts

**Път: dbInit.ts**Описание и предназначение:  
Този файл съдържа скрипт за първоначална инициализация на базата данни. Използва се за създаване на административен потребител, задаване на квота за съхранение и създаване на основна директория, ако базата данни е празна. Скриптът е предназначен да се изпълнява еднократно при първоначално стартиране на системата.

### Основни компоненти:

#### 3.1. Импорти

Импортира всички необходими зависимости: връзка, модели, конфигурация, Mongoose и bcrypt за хеширане на пароли.

#### 3.2. Основна функция: initializeDatabase()

Подробно описание на стъпките:

##### Връзка с базата данни:

* + Използва connectDB() за установяване на връзка.
  + Проверка за съществуващи потребители:
  + Ако има поне един потребител, инициализацията се прекратява, за да се избегне дублиране.

##### Създаване на администратор:

* + Паролата се хешира с bcrypt.
  + Създава се нов потребител с фиксирани данни (име, фамилия, потребителско име, имейл, парола, статус).
  + Потребителят се записва в базата.

##### Създаване на квота за съхранение:

* + Създава се нов документ в StorageQuota за администратора.
  + Квотата е с бизнес план и максимално пространство.

##### Създаване на основна директория:

* + Създава се директория с име root, собственост на администратора, без родител.
  + Логване на резултатите:
  + След всяка основна операция се логва потвърждение.

##### Затваряне на връзката:

* + Винаги затваря връзката към базата, независимо от резултата.

#### 3.3. Стартиране като скрипт

Позволява файлът да се изпълнява директно като standalone скрипт.

#### 3.4. Експорт Експортира функцията за инициализация за евентуално използване в други части на приложението.

### Връзки и работен поток

* db.ts предоставя конфигурация и квоти за съхранение.
* connection.ts използва тази конфигурация, за да установи и управлява връзката с MongoDB.
* dbInit.ts използва функцията за връзка, моделите и конфигурацията, за да създаде начални данни при първоначално стартиране.

### Примерен сценарий на изпълнение

1. Стартира се dbInit.ts (директно или чрез npm скрипт).
2. Установява се връзка с базата данни.
3. Проверява се дали има потребители.
4. Ако няма:
   * Създава се администратор.
   * Създава се квота за администратора.
   * Създава се основна директория.
5. Връзката се затваря.
6. Всички грешки се логват, а при критични проблеми процесът се прекратява.

### Предимства на архитектурата

* Централизирана конфигурация: Всички параметри са на едно място и лесно се управляват.
* Сигурност: Паролите се хешират, а връзката се валидира.
* Гъвкавост: Лесно добавяне на нови планове и промяна на квоти.
* Надеждност: Грешките се обработват, а връзката се затваря коректно при прекъсване на приложението.
* Автоматизация: Позволява автоматично инициализиране на базата данни при първо стартиране.

Backend частта на проекта е модулно организирана, като всяка функционалност (автентикация, файлове, директории) е разделена в отделни файлове. Използването на MongoDB като основна база данни позволява гъвкаво и мащабируемо съхранение на файловата структура. Middleware-ите гарантират сигурност и правилен достъп, а multer осигурява ефективна обработка на файлове в паметта. Всички компоненти работят под един Express сървър на порт 3000.

# fileRoutes.ts (Маршрути за управление на файлове)

Файлът fileRoutes.ts дефинира основните API маршрути за работа с файлове в приложението Personal Cloud. Той използва Express.js и осигурява възможност за качване, изтегляне, изтриване, визуализация и преглед на директории, както и за достъп до всички файлове, споделени или публични.

## Основни зависимости:

import express from "express";

import multer from "multer";

import { uploadFile, downloadFile, deleteFile, ... } from "../controllers/fileController.js";

import { isAuthenticated } from "../middlewares/errorHandler.js";

import fileModel from "../models/fileModel.js";

* express – за създаване на API маршрути
* multer – middleware за обработка на multipart/form-data (качване на файлове)
* fileController.js – съдържа логиката за всяка операция с файлове
* isAuthenticated – middleware за проверка на сесията
* fileModel.js – модел на файла (MongoDB)

## Middleware:

1. **router.use(isAuthenticated)** – гарантира, че всички маршрути са достъпни само за автентикирани потребители.
2. **Добавяне на потребител в req.user** – извлича \_id от сесията и го прикачва към заявката, за да е достъпен във всички контролери.

router.use((req, res, next) => {

if (req.session?.user) {

req.user = { \_id: req.session.user.id };

}

next();

});

Основни маршрути:

POST /upload

* Middleware: upload.single("uploadFile")
* Описание: Качва файл в системата.
* Контролер: uploadFile

GET /download/:id

* Описание: Изтегля файл по дадено id.
* Контролер: downloadFile

DELETE /delete/:id

* Описание: Изтрива файл по ID. Променен от GET на DELETE за съответствие с REST принципите.
* Контролер: deleteFile

GET /directory/:directoryId?

* Описание: Показва съдържанието на директория (по избор с ID).
* Контролер: showDataInDirectory

GET /preview/:id

* Описание: Визуализира предварително даден файл (например PDF или изображение).
* Контролер: previewFile

POST /directory

* Описание: Създава нова директория.
* Контролер: createDirectory

GET /path/:directoryId

* Описание: Извежда пътя към директорията по дадено ID.
* Контролер: getDirectoryPath

Допълнителен маршрут:

GET /all

* Описание: Връща всички файлове, които:
  + са собственост на потребителя,
  + са споделени с него или
  + са публични (isPublic: true)
* Търси чрез fileModel.find(...)
* Извежда само определени полета: name, type, path, size, lastModified

Заключение:

Файлът fileRoutes.ts играе ключова роля в бекенд частта на проекта, като дефинира ясно разделени маршрути за всички действия, свързани с файлове. Всеки маршрут използва специфичен контролер, което улеснява поддръжката и мащабирането на кода.

# authRoutes.ts

## Цел:

Файлът authRoutes.ts дефинира маршрутите за автентикация на потребители – регистрация, вход, проверка на сесия и изход.

Използвани модули:

* express – за дефиниране на REST API маршрути
* bcryptjs – за хеширане и сравнение на пароли
* express-session – за управление на сесии чрез cookies
* userModel – Mongoose модел за съхранение на потребителите в MongoDB

## Разширения за TypeScript: Файлът разширява типовете на express-session, за да може req.session.user да съдържа информация за потребителя (id, username, email).

POST /api/auth/register

* Проверява дали всички полета са попълнени
* Проверява дали паролите съвпадат
* Проверява дали имейлът или потребителското име вече съществуват в базата
* Хешира паролата с bcrypt
* Създава нов документ в MongoDB и връща съобщение за успех

Примерен отговор:  
 {  
 "message": "Registered successfully"  
 }

POST /api/auth/login

* Изисква username и password
* Проверява дали потребител съществува в базата
* Сравнява подадената парола с хешираната в базата чрез bcrypt.compare()
* Ако е валидно:  
  + записва потребителя в сесията (req.session.user = {...})
  + връща съобщение за успех

Примерен отговор:  
 {  
 "message": "Login successful"  
 }

GET /api/auth/me

* Връща информация за текущия логнат потребител от сесията
* Ако няма сесия – връща грешка 401

Примерен отговор:  
 {  
 "user": {  
 "id": "...",  
 "username": "...",  
 "email": "..."  
 }  
 }

POST /api/auth/logout

* Унищожава потребителската сесия с req.session.destroy()
* Връща съобщение за успех

Примерен отговор:  
 {  
 "message": "Logged out"  
 }

Заключение:  
 authRoutes.ts съдържа пълна логика за управление на потребителски акаунти чрез сесии. Работи с MongoDB за съхранение и използва bcrypt за сигурност. Системата е лесна за разширяване и интегриране в по-големи проекти.

# userController.ts

## Цел:

Файлът userController.ts съдържа основната логика за регистрация и вход на потребители в системата. Той използва bcrypt за защита на пароли и работи с MongoDB чрез Mongoose модел User.

Описание на функциите:

Функция: register(req: Request, res: Response)

* Път: POST /api/auth/register
* Описание: Регистрира нов потребител в системата.
* Стъпки:
  1. Извлича входните данни от тялото на заявката (firstName, lastName, username, email, password).
  2. Проверява дали всички полета са попълнени.
     + Ако липсва поле, връща статус 400 с подходящо съобщение.
  3. Проверява дали email вече е използван чрез User.findOne({ email }).
     + Ако съществува, връща статус 400 с грешка.
  4. Проверява дали username вече е използван чрез User.findOne({ username }).
     + Ако съществува, връща статус 400 с грешка.
  5. Хешира паролата с bcrypt.hash(password, 10).
  6. Създава нов потребител с хеширана парола и запазва в базата данни.
  7. Връща статус 201 и съобщение "Registration successful."

Примерен отговор при успех: { "message": "Registration successful." }

Функция: login(req: Request, res: Response)

* Път: POST /api/auth/login
* Описание: Удостоверява потребител и създава сесия.
* Стъпки:
  1. Извлича username и password от тялото на заявката.
  2. Проверява дали и двете стойности са налични.
     + Ако липсват, връща статус 400.
  3. Проверява дали потребител съществува чрез User.findOne({ username }).
     + Ако не съществува, връща статус 401.
  4. Сравнява подадената парола с хешираната в базата чрез bcrypt.compare().
     + Ако не съвпада, връща статус 401.
  5. Обновява полето lastLogin с текуща дата.
  6. Записва потребителските данни в req.session.user.
  7. Връща статус 200 и съобщение "Login successful."

Примерен отговор при успех: { "message": "Login successful." }

Забележки:

* Хеширането на пароли осигурява защита при евентуален пробив в базата.
* Сесиите позволяват проследяване на автентикация без използване на JWT.
* Важно е да се защитят всички чувствителни данни чрез HTTPS и ограничения на достъп в продукционна среда.

# api/auth.ts

## Цел:

Файлът auth.ts съдържа асинхронни функции за комуникация с бекенд сървъра, отговарящи за регистрация, вход, изход и проверка на текущ логнат потребител. Тези функции се използват във frontend частта, например при изпращане на заявки от форми.

API базов адрес: const API\_BASE = 'http://localhost:3000'; Всички заявки се изпращат към този адрес.

## Описание на функциите:

1. login(email, password)

* Тип заявка: POST
* Път: /login
* Изпраща имейл и парола към сървъра за вход
* В заявката се включва заглавие Content-Type: application/json
* credentials: 'include' осигурява изпращане на cookies (сесии)

1. register(firstName, lastName, username, email, password, confirmPassword)

* Тип заявка: POST
* Път: /register
* Изпраща всички полета, нужни за регистрация на потребител
* Данните се преобразуват с JSON.stringify
* credentials: 'include' гарантира, че ста се установява веднага

1. logout()

* Тип заявка: POST
* Път: /logout
* Не изпраща тяло на заявката
* Целта е да се прекрати сесията на потребителя на сървъра

1. getUser()

* Тип заявка: GET
* Път: /me
* Извършва проверка дали има логнат потребител по активна сесия
* Ако отговорът е успешен (res.ok), резултатът се връща като JSON
* В противен случай връща null

Заключение: Файлът auth.ts е мост между потребителския интерфейс (формите за вход и регистрация) и бекенда. Чрез тези функции се изпращат и получават данни от сървъра, като се използва fetch API и сесии чрез cookies. Това е централен елемент при изграждането на frontend логиката за автентикация.

# fileSearch.ts

## Описание:

Файлът дефинира функция за търсене на файлове в списък от файлови обекти. Функцията позволява търсене по име на файл или пълния път до файла, като използва зададен шаблон за търсене.

## Интерфейс:

Интерфейсът IFile е вмъкнат от models/fileModel.ts и дефинира структурата на файловите обекти:

* name: Име на файла (тип: String, задължително).
* path: Пълен път до файла (тип: String, задължително).
* type: Тип на файла (например file или directory, тип: String, задължително).
* owner: Собственик на файла (тип: ObjectId, задължително).
* isPublic: Дали файлът е публичен (тип: Boolean, задължително).
* lastModified: Дата на последна модификация (тип: Date, задължително).
* Други свойства, дефинирани в IFile.

## Функция searchFiles:

Функцията searchFiles извършва търсене на файлове в списък от файлови обекти.

### Аргументи:

1. files: Списък от файлови обекти (тип: Partial<IFile>[]).
2. pattern: Шаблон за търсене (тип: String).

### Логика:

* Премахване на празни пространства: Шаблонът за търсене се обработва чрез trim() и се преобразува в малки букви чрез toLowerCase().
* Търсене по пълен път: Ако шаблонът съдържа /, функцията филтрира файловете по пълния път (path).
* Търсене по име: Ако шаблонът не съдържа /, функцията филтрира файловете по име (name).

### Връщана стойност:

* Списък от файлови обекти, които съответстват на шаблона за търсене (тип: Partial<IFile>[]).

### Пример за използване:

| import { IFile } from "../models/fileModel"; // Импортиране на интерфейса IFile import mongoose from "mongoose"; // Импортиране на mongoose за тип ObjectId  // Примерни данни за файлове, съответстващи на MongoDB схемата export const mockFiles: Partial<IFile>[] = [  {  \_id: '1',  name: 'nodemon.json',  type: 'file',  path: '/nodemon.json',  owner: new mongoose.Types.ObjectId(),  isPublic: false,  mimeType: 'application/json',  size: 123,  lastModified: new Date(),  originalName: 'nodemon.json',  storageUrl: undefined,  sharedWith: [],  starred: false,  },  {  \_id: '2',  name: 'package.json',  type: 'file',  path: '/package.json',  owner: new mongoose.Types.ObjectId(),  isPublic: false,  mimeType: 'application/json',  size: 456,  lastModified: new Date(),  originalName: 'package.json',  storageUrl: undefined,  sharedWith: [],  starred: false,  },  {  \_id: '3',  name: 'public',  type: 'directory',  path: '/public',  owner: new mongoose.Types.ObjectId(),  isPublic: false,  lastModified: new Date(),  originalName: 'public',  storageUrl: undefined,  sharedWith: [],  starred: false,  },  {  \_id: '4',  name: 'index withspace.ts',  type: 'file',  path: '/public/index withspace.ts',  owner: new mongoose.Types.ObjectId(),  isPublic: false,  mimeType: 'text/typescript',  size: 789,  lastModified: new Date(),  originalName: 'index.ts',  parent: new mongoose.Types.ObjectId(),  storageUrl: undefined,  sharedWith: [],  starred: false,  },  {  \_id: '5',  name: 'README.md',  type: 'file',  path: '/README.md',  owner: new mongoose.Types.ObjectId(),  isPublic: true,  mimeType: 'text/markdown',  size: 100,  lastModified: new Date(),  originalName: 'README.md',  storageUrl: undefined,  sharedWith: [],  starred: false,  }, ];  /\*\*  \* Функция за търсене на файлове по зададен шаблон.  \* @param files - Списък от файлове, в който се извършва търсенето.  \* @param pattern - Шаблон за търсене (може да бъде част от името или пълния път).  \* @returns Списък от файлове, които съответстват на шаблона за търсене.  \*/ export function searchFiles(files: Partial<IFile>[], pattern: string): Partial<IFile>[] {  const searchTerm = pattern.trim().toLowerCase(); // Премахване на празни пространства и преобразуване в малки букви  if (!searchTerm) return []; // Ако шаблонът е празен, връща празен списък   // Проверка дали шаблонът съдържа '/'  if (searchTerm.includes('/')) {  // Филтриране по пълния път  return files.filter(file => file.path?.toLowerCase().includes(searchTerm));  }   // Филтриране по име на файл  return files.filter(file => file.name?.toLowerCase().includes(searchTerm)); }  // Пример за използване: const searchTerm = "index"; // Шаблон за търсене const results = searchFiles(mockFiles, searchTerm); // Извършване на търсене console.log("Шаблон за търсене:", searchTerm); // Показване на шаблона за търсене console.log("Резултати:", results); // Показване на резултатите от търсенето  /\*\*  \* Примерен вход:  \* searchTerm = "index";  \* mockFiles = [  \* { name: "nodemon.json", path: "/nodemon.json", ... },  \* { name: "package.json", path: "/package.json", ... },  \* { name: "public", path: "/public", ... },  \* { name: "index withspace.ts", path: "/public/index withspace.ts", ... },  \* { name: "README.md", path: "/README.md", ... },  \* ];  \*  \* Примерен изход:  \* Шаблон за търсене: "index"  \* Резултати: [  \* { name: "index.ts", path: "/public/index bimbim.ts", ... }  \* ]  \*/ |
| --- |

## Функция: renderResults

Функцията рендира резултатите от търсенето в HTML контейнер.

### Аргументи:

1. results: Списък от файлове, които съответстват на шаблона за търсене (тип: Partial<IFile>[]).
2. resultsContainer: HTML елемент, в който ще се показват резултатите (тип: HTMLElement).

### Логика:

Изчистване на предишните резултати.

Създаване на HTML елементи за всеки файл и добавянето им към контейнера.

## Функция: handleSearch

Функцията обработва търсенето с debounce, като слуша за промени в полето за търсене.

### Аргументи:

1. searchInput: HTML елемент за въвеждане на шаблон за търсене (тип: HTMLInputElement).
2. resultsContainer: HTML елемент, в който ще се показват резултатите (тип: HTMLElement).
3. files: Списък от файлове, в който ще се извършва търсенето (тип: Partial<IFile>[]).

### Логика:

* Слушане за промени в полето за търсене.
* Използване на debounce за забавяне на търсенето.
* Извикване на searchFiles за филтриране на файловете.
* Извикване на renderResults за показване на резултатите.

### Пример за използване:

| import { handleSearch } from './utils/fileSearch'; import { mockFiles } from './utils/fileSearch';  document.addEventListener('DOMContentLoaded', () => {  const searchInput = document.getElementById('search-input') as HTMLInputElement;  const resultsContainer = document.getElementById('search-results') as HTMLElement;   handleSearch(searchInput, resultsContainer, mockFiles); }); |
| --- |

# 

# Index.js

**Frontend** File Manager & Authentication

Този едностраничен JavaScript модул задвижва лек и функционален personal cloud‑storage интерфейс. Той осигурява:

* Удостоверяване на потребители чрез REST API (вход, регистрация, изход, проверка на сесия)
* CRUD операции с файлове (списък, качване, създаване на директория, изтегляне, изтриване) с навигация тип „хлебни трохи“
* Отзивчиво управление на UI състоянието, което превключва между публични страници и табло за автентицирани потребители

Бек‑енд предпоставка: Node/Express сървър на http://localhost:3000, който експонира описаните по‑долу маршрути и използва сесии, базирани на cookies.

**Работещ бек‑енд със следните маршрути:**

* POST /api/auth/login
* POST /api/auth/register
* POST /api/auth/logout
* GET /api/auth/me
* GET /api/files/directory/:id?
* GET /api/files/path/:id
* GET /api/files/storage
* POST /api/files/upload
* POST /api/files/directory
* GET /api/files/download/:id
* DELETE /api/files/delete/:id

## Поток на удостоверяване

## Клас AuthService

* login(username, password)
  + Заявка: POST /api/auth/login с { username, password }.
  + В отговор се очаква обект user.
  + При успех методът запазва потребителя в localStorage, а cookie‑то със сесията се задава от сървъра.
* register(formData)
  + Заявка: POST /api/auth/register с данните на формуляра (firstName, lastName, username, email и пароли).
  + Получава се нов обект user при успех.
* logout()
  + Заявка: POST /api/auth/logout без тяло.
  + При успех сървърът изтрива сесията, а клиентът се връща в състояние „гост“.

Всички методи използват credentials: "include", така че session cookie‑то да се изпраща автоматично.

### Проверка на сесията

При зареждане на страницата се изпраща GET /api/auth/me. Ако сървърът върне 200 и обект user, интерфейсът веднага се превключва към таблото (чрез updateUIForAuthState).

## Навигация и управление на състоянието (UI)

* showPage(pageId) – прост рутер; скрива всички елементи с клас .page и показва само желания.
* updateUIForAuthState(user) – централизира toggling между:
  + „Гост“ режим – вижда се #guest-nav и публичните страници.
  + „Потребител“ режим – вижда се #user-nav, таблото и breadcrumb‑ите.

Навигационни секции:

* #guest-nav: Home, Login, Register, About.
* #user-nav: Бутони за качване/директория и бутон Logout.

## Модул за управление на файлове

### Основни функции

* loadFiles(directoryId = null) – зарежда съдържанието на директория (или root), обновява currentDirectoryId, извиква renderFilesList и updateBreadcrumb.
* renderFilesList(files) – вгражда HTML за всеки файл или показва „празна директория“.
* updateBreadcrumb(directoryId) – генерира навигация тип „хлебни трохи“; при null показва само „Root“.

Индикатор за заето пространство

* updateStorageInfo() – взема общ и зает обем чрез /api/files/storage, пресмята процент и обновява прогрес лента и стойности в GB.

Операции върху файлове

* downloadFile(id) – GET /api/files/download/:id, създава blob, тригърва изтегляне.
* deleteFile(id) – DELETE /api/files/delete/:id; след потвърждение презарежда списъка.

## Работа с форми и модални прозорци

* Login форма (#login form) – при submit се извиква AuthService.login; при успех таблото се показва и формата се занулява.
* Register форма (#register form) – при успешна регистрация се връща към екрана за вход.
* Upload форма (#uploadForm) – изпраща POST /api/files/upload (multipart). При успех модалът се затваря и списъкът се презарежда.
* Create Directory форма (#directoryForm) – изпраща POST /api/files/directory с име и родител‐директория.

Помощни функции: showUploadModal, showCreateDirectoryModal, closeModal.

## Помощни функции

* formatFileSize(bytes) – конвертира байтове в KB / MB / GB (до 2 знака).
* createBreadcrumbElement() – създава контейнера за breadcrumb при първото му извикване.
* Други утилити: showPage, getCurrentDirectoryId, handleSuccessfulLogin.

## Глобални експорти

В края на файла няколко функции се добавят към window, за да са достъпни от вграден HTML/onclick атрибути: showPage, showUploadModal, showCreateDirectoryModal, closeModal, downloadFile, deleteFile, handleLogout, loadFiles.

## Обработка на грешки

Всички асинхронни операции са оградени с try…catch. При грешка:

1. Детайли се логват в конзолата за дебъг.
2. Потребителят вижда диалог alert() с кратко описание.

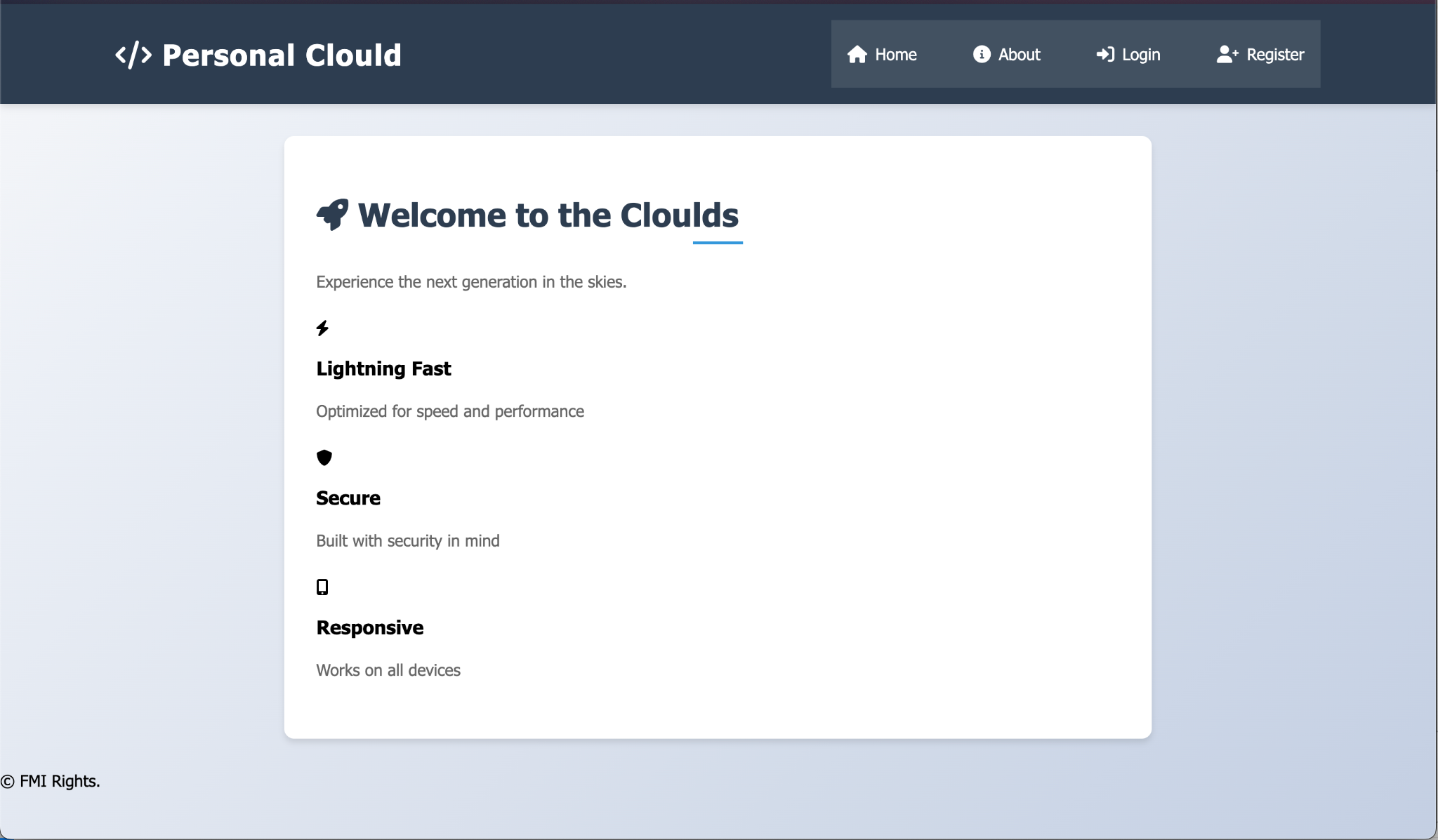
(За по‑добро UX заменете alert() с toast/snackbar.)

## Разширяване и персонализация

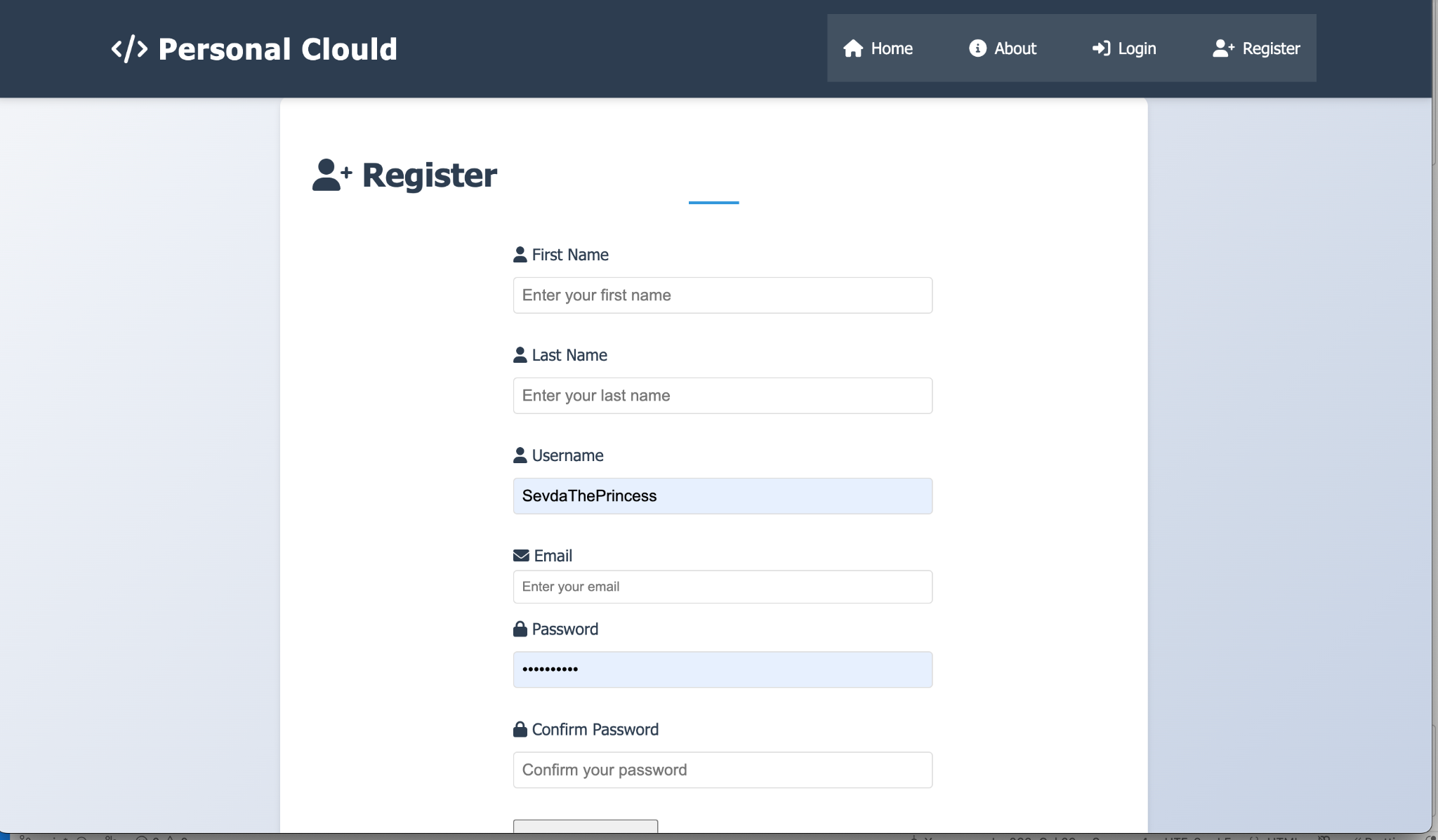
* Пагинация и търсене при големи директории.
* Drag‑and‑drop качване с прогрес бар.
* Премествай/преименувай файлове чрез допълнителни API маршрути.
* Интернационализация – извеждане на текстови ресурси в отделен файл

# Екранни снимки от интерфейса

## Начална страница

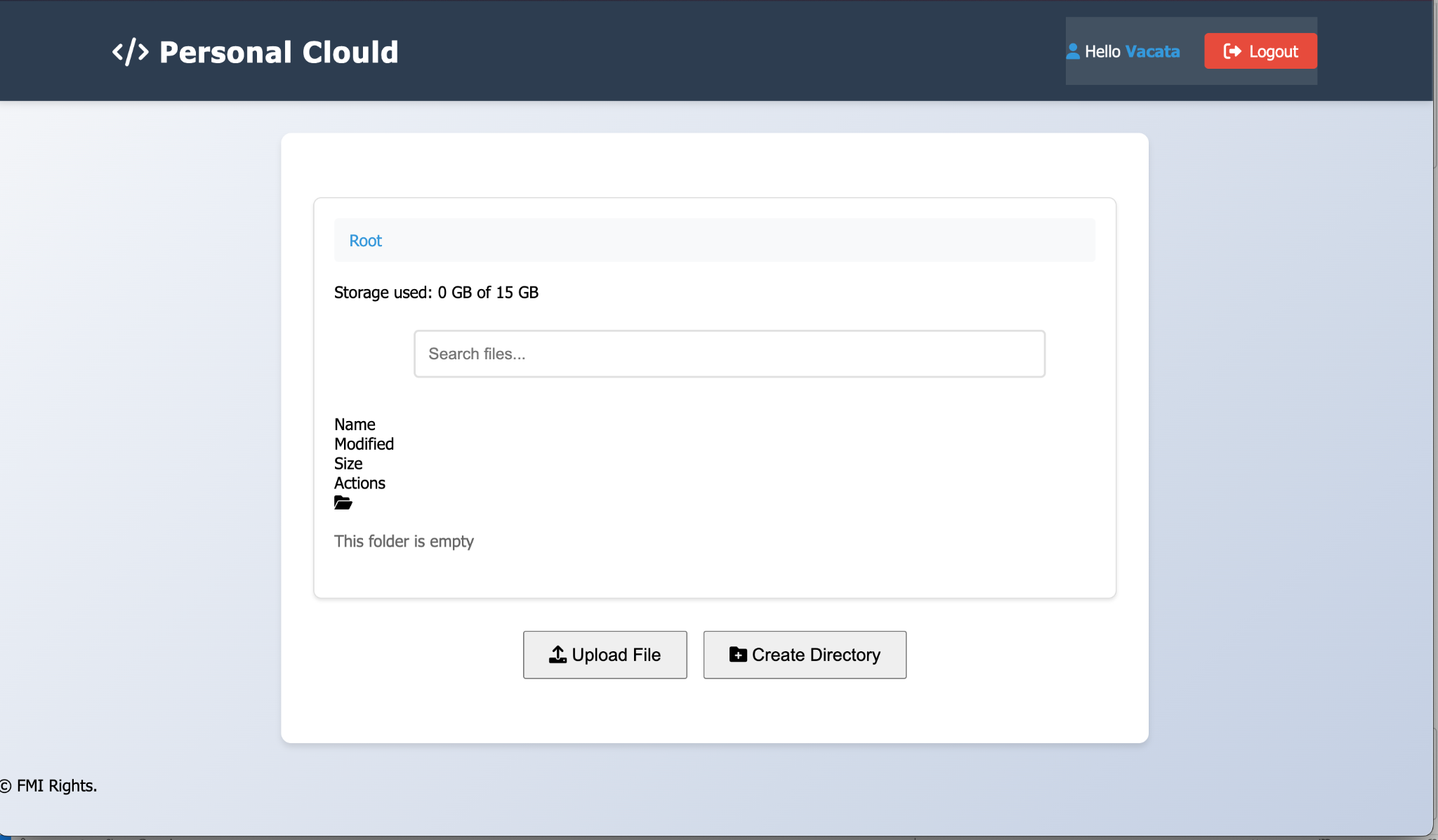


## Форма за регистрация на нов потребител



Използва: /api/auth/register

**Страница за потребителски вход**

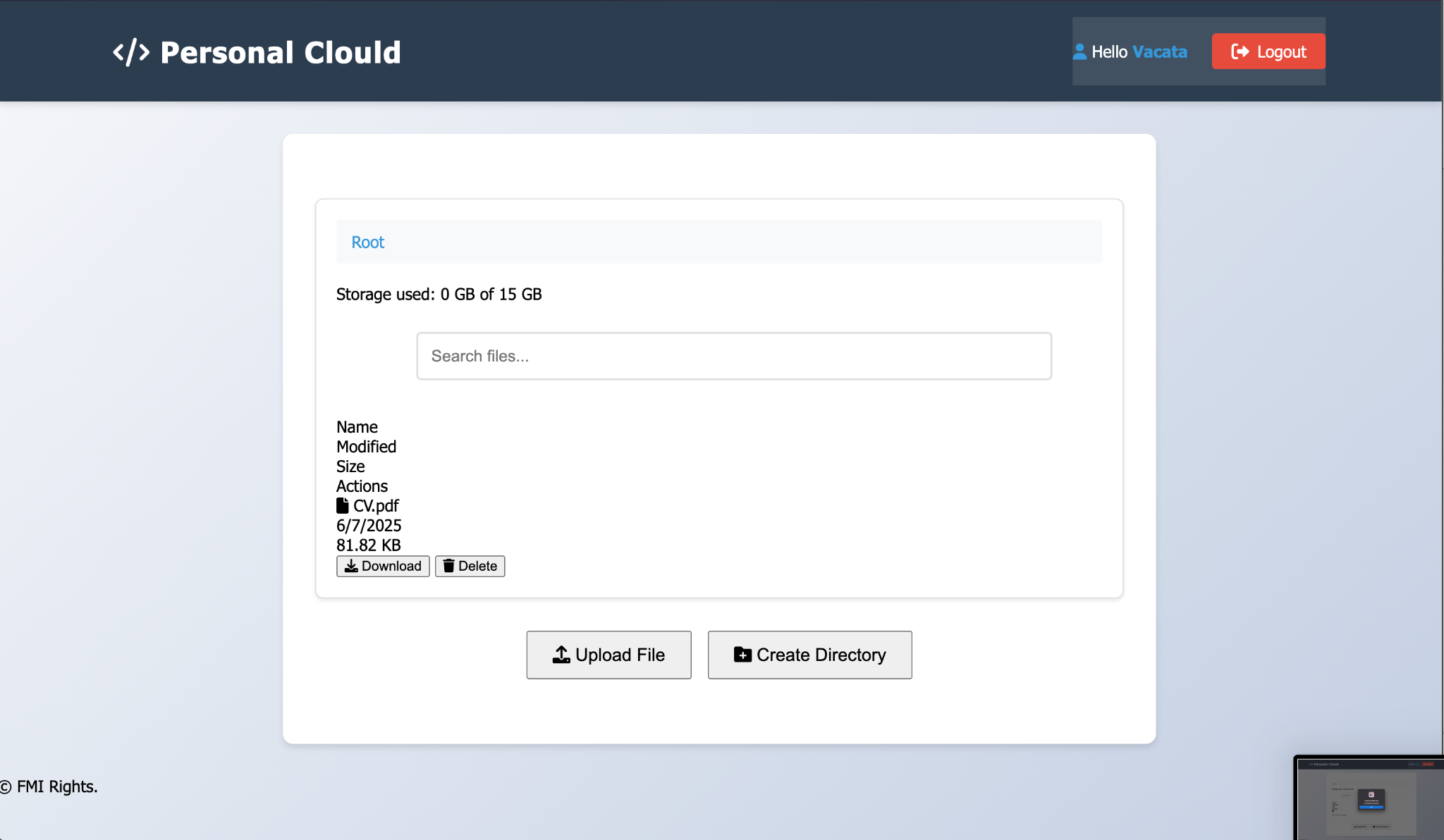
****

Използва: /api/auth/login

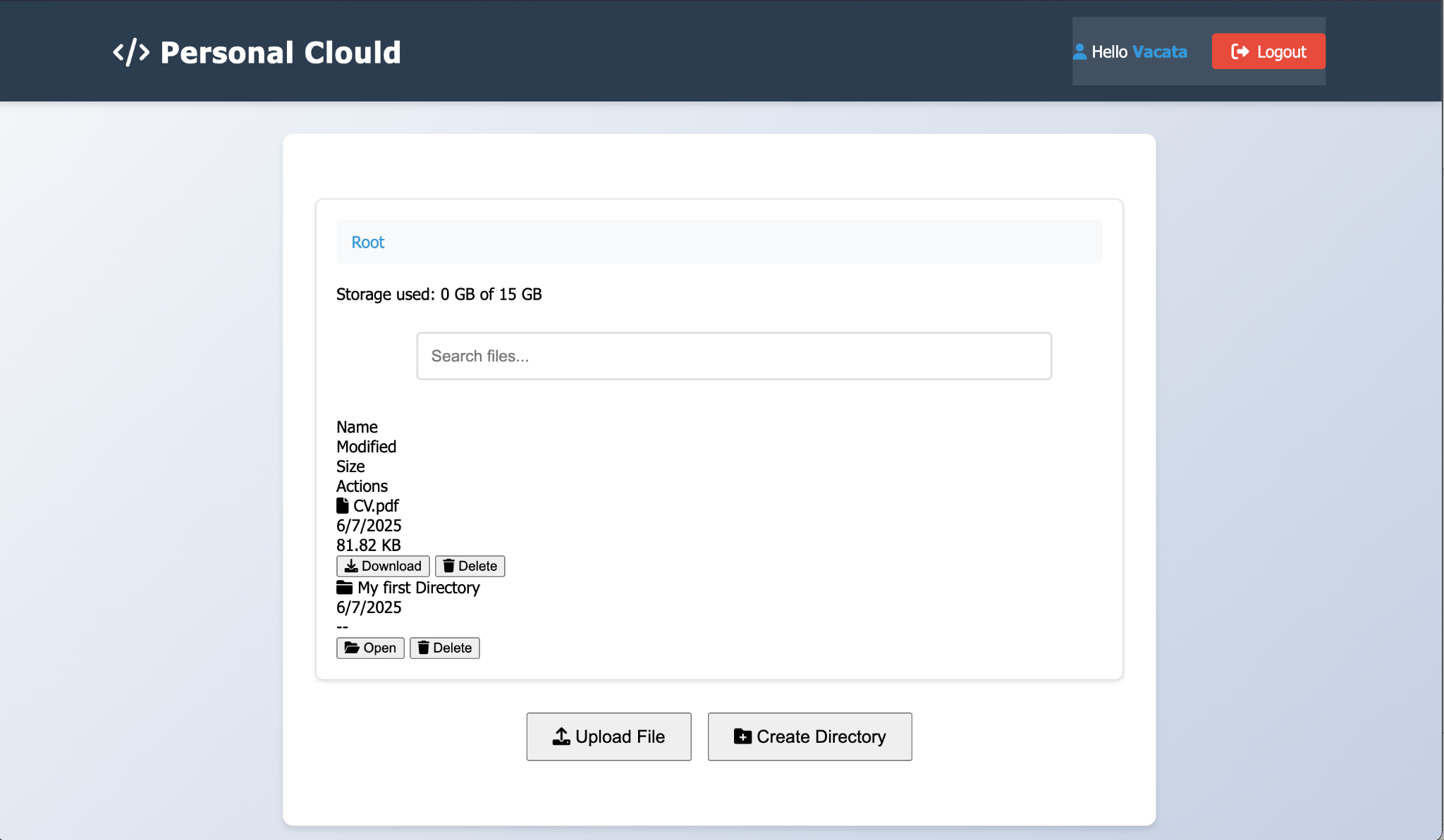
## Модул за качване на потребителски файлове

Използва: /api/files/upload

## Модул за управление на успешно добавени файлове

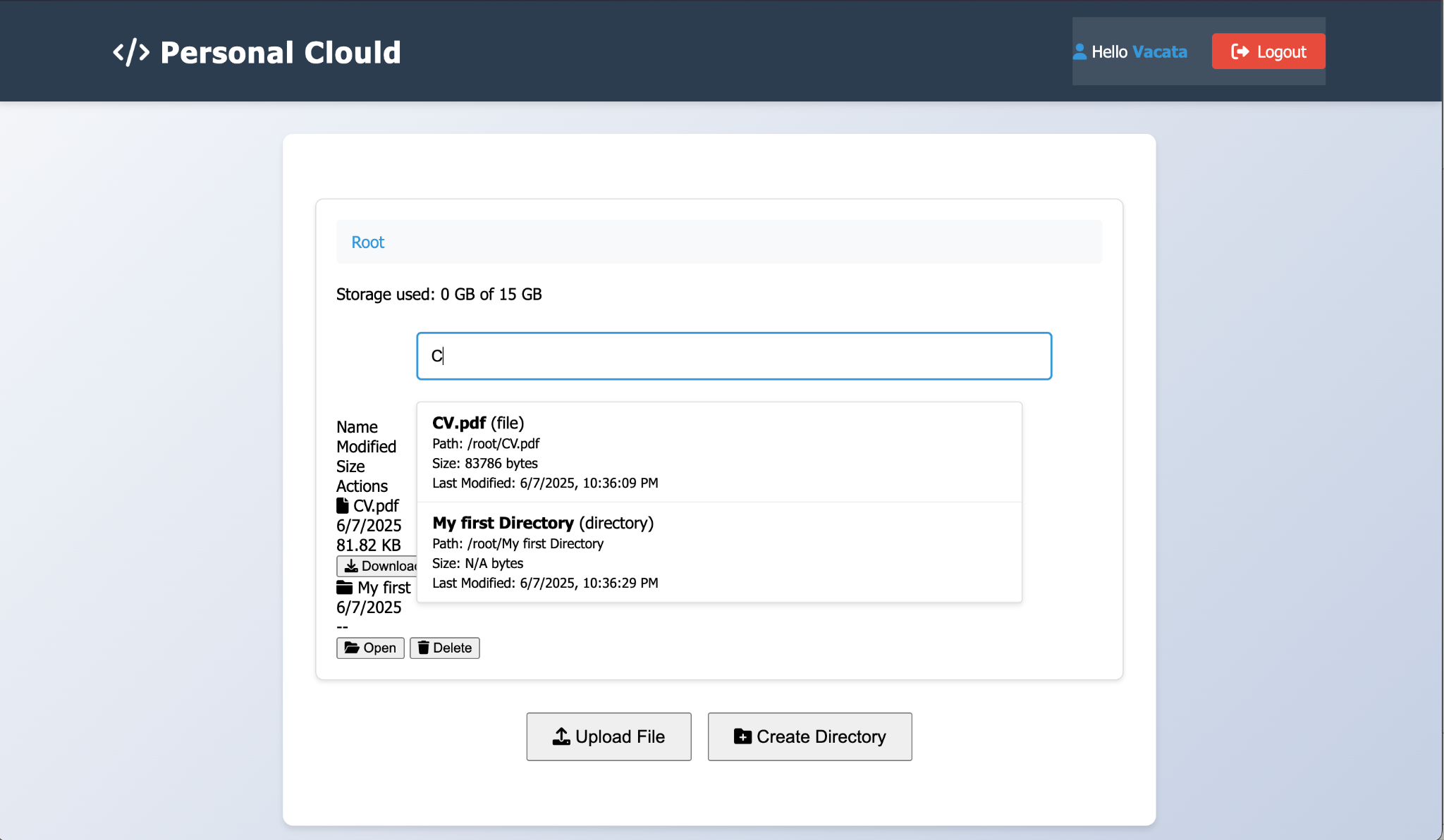
****

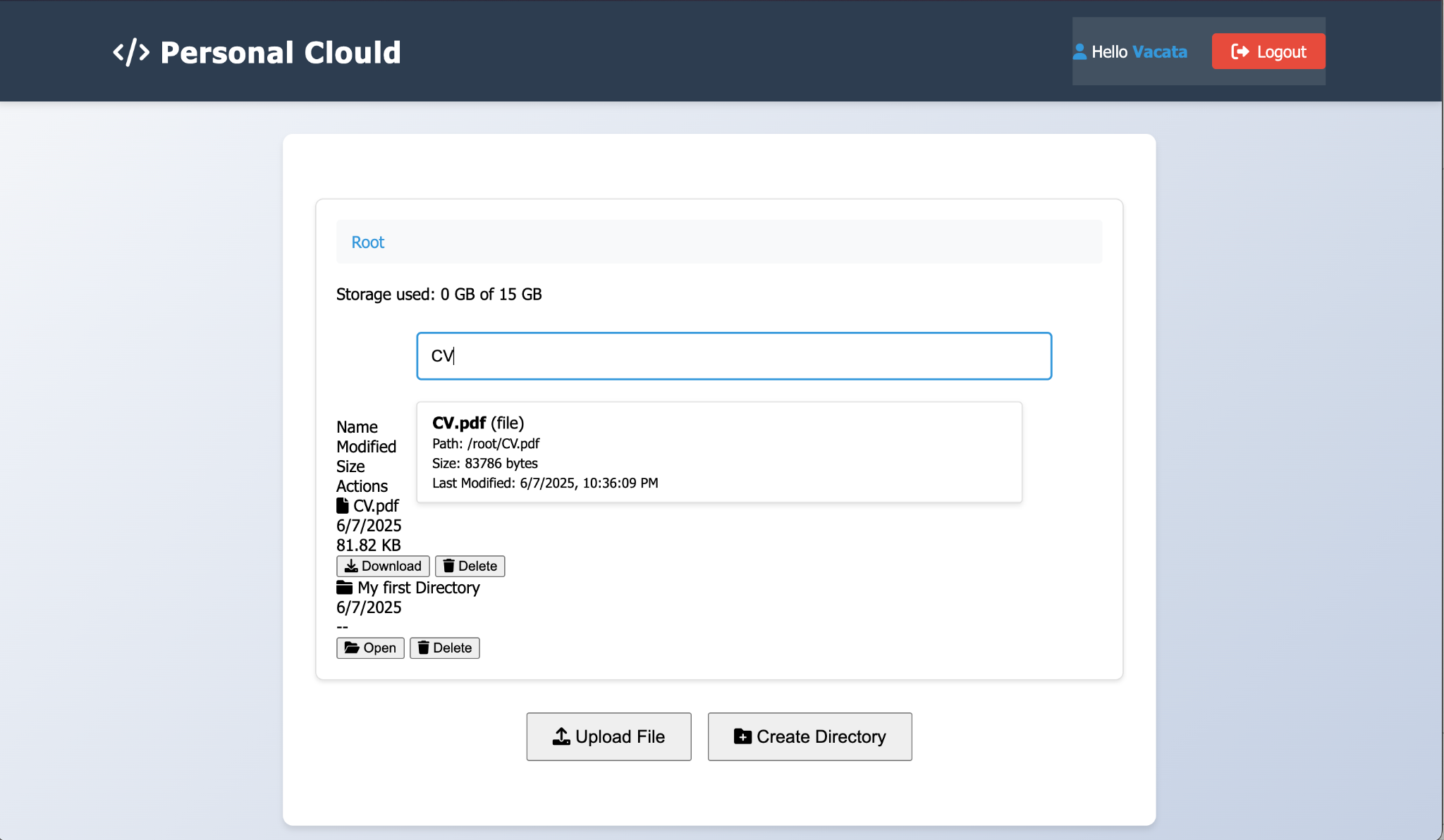
## Визуален изглед с добавена нова директория в работната среда

****

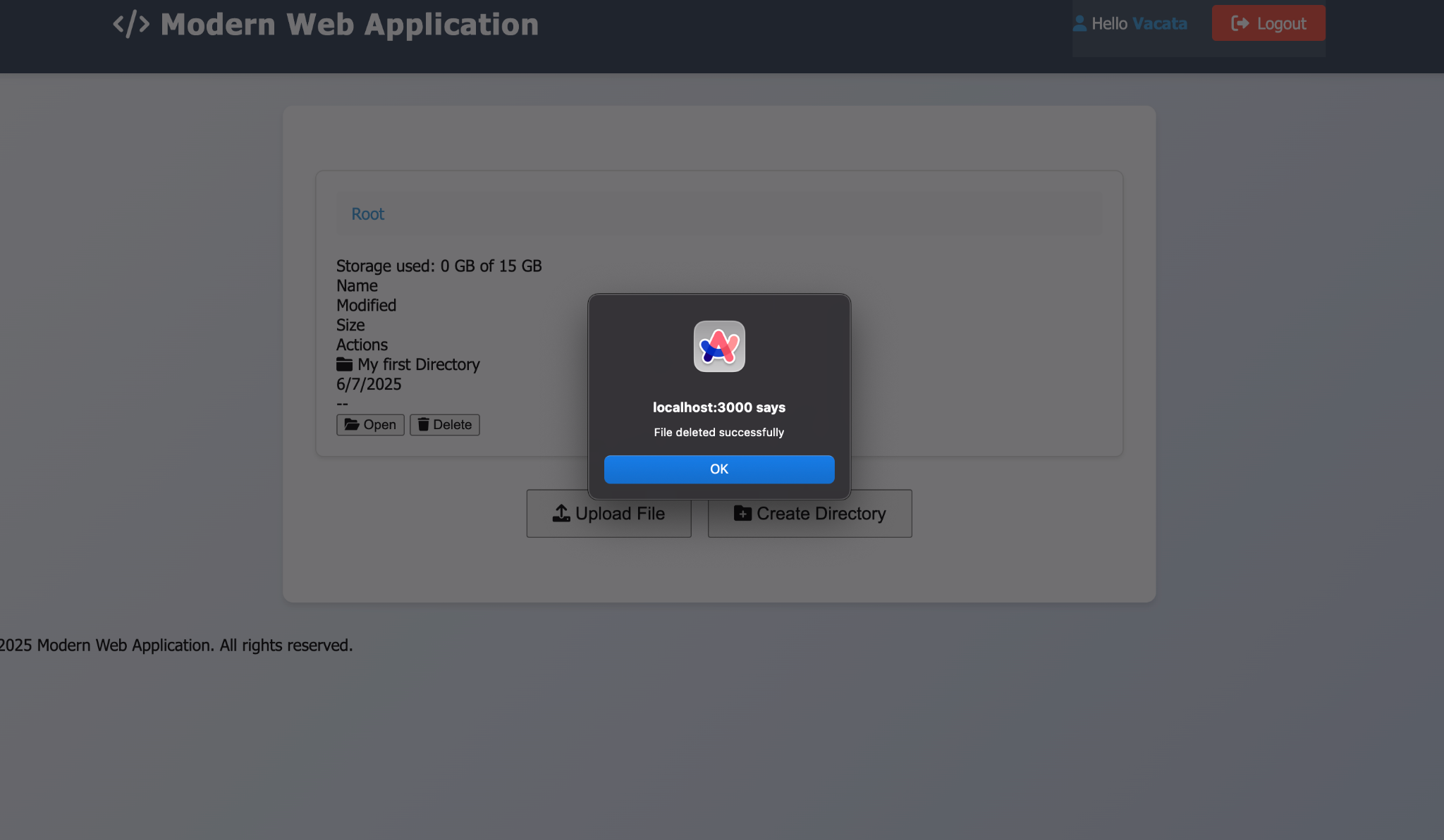
Използва: /api/files/directory

## Търсене с частично въведени символи от името на файл

****

****

## Информационно съобщение за успешно изтрит файл

****

Използва /api/files/delete/:id