ImpactCV Technical Documentation

Table of Contents

Project Structure

The ImpactCV project follows a standard client-server architecture with separate frontend and backend components:

```
ImpactCV-master/
├── public/                 # Static assets
├── server/                 # Backend Express.js server
│   ├── routes/             # API route definitions
│   ├── controllers/        # Request handlers
│   ├── models/             # Database models
│   ├── middleware/         # Express middleware
│   ├── schema.sql          # Database schema
│   ├── setup-db.js         # Database initialization
│   └── server.js           # Main server entry point
├── src/                    # Frontend React application
│   ├── components/         # React components
│   │   ├── sections/       # Resume section components
│   │   └── ui/             # Reusable UI components
│   ├── hooks/              # Custom React hooks
│   ├── lib/                # Utility functions and types
│   ├── styles/             # CSS and styling files
│   ├── App.tsx             # Main React component
│   └── main.tsx            # React entry point
├── .env                    # Environment variables
├── package.json            # Project dependencies
└── vite.config.js          # Vite configuration
```

Frontend Architecture

Component Structure

The frontend is built with React and TypeScript, following a component-based architecture:

1. **Core Components**:
   - `ResumeEditor` : Main editor interface
   - `ResumePreview` : Live preview of the resume
   - `ThemeSelector` : Theme selection and customization
   - `SectionManager` : Manages adding/removing sections

2. **Section Components**:

    - `ExperienceSection` : Work history entries
    - `EducationSection` : Educational background
    - `SkillsSection` : Technical and soft skills
    - `ProjectsSection` : Project showcase
    - `CustomSection` : User-defined sections

3. **UI Components**:

    - Form elements (inputs, dropdowns, etc.)
    - Buttons and action elements
    - Modal dialogs
    - Drag-and-drop interfaces

State Management

The application uses React's Context API for global state management:

- `ResumeContext` : Stores the complete resume data structure
- `ThemeContext` : Manages the current theme and customizations
- `UIContext` : Controls UI state (sidebar open/closed, active section, etc.)

Local component state is managed with React hooks for component-specific concerns.

Data Flow

1. User interactions trigger state updates in React components
2. State changes are propagated through the component tree
3. API calls are made to persist changes to the backend
4. The UI updates to reflect the current state

Backend Architecture

Server Setup

The backend is built with Express.js and provides:

- RESTful API endpoints
- Database connectivity via `pg` (PostgreSQL client)
- File upload handling with `multer`
- Environment configuration with `dotenv`

API Structure

The API follows RESTful conventions with these main resource endpoints:

- `/api/users` : User account management
- `/api/resumes` : Resume CRUD operations
- `/api/upload` : File upload handling
- `/api/generate-pdf` : PDF generation

Middleware

Custom middleware is used for:

- Authentication and authorization

- Request validation
- Error handling
- CORS configuration

Database Schema

The PostgreSQL database consists of two main tables:

Users Table

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Resumes Table

```sql
CREATE TABLE resumes (
  id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
  title VARCHAR(255) NOT NULL,
  theme VARCHAR(50),
  data JSONB NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

API Reference

Authentication Endpoints

```
POST /api/users/register
```

- Creates a new user account
- Request body: `{ email, password }`
- Response: User object with token

```
POST /api/users/login
```

- Authenticates a user
- Request body: `{ email, password }`
- Response: User object with token

Resume Endpoints

```
GET /api/resumes
```

- Returns all resumes for the authenticated user
- Headers: `Authorization: Bearer {token}`
- Response: Array of resume objects

```
GET /api/resumes/:id
```

- Returns a specific resume by ID
- Headers: `Authorization: Bearer {token}`
- Response: Single resume object

```
POST /api/resumes
```

- Creates a new resume
- Headers: `Authorization: Bearer {token}`
- Request body: Resume data object
- Response: Created resume object with ID

```
PUT /api/resumes/:id
```

- Updates an existing resume
- Headers: `Authorization: Bearer {token}`
- Request body: Updated resume data
- Response: Updated resume object

```
DELETE /api/resumes/:id
```

- Deletes a resume
- Headers: `Authorization: Bearer {token}`
- Response: Success message

File Upload Endpoint

```
POST /api/upload
```

- Uploads a profile photo
- Headers: `Authorization: Bearer {token}`
- Request: Form data with 'photo' field
- Response: URL to the uploaded file

PDF Generation Endpoint

```
POST /api/generate-pdf
```

- Generates a PDF from resume data
- Headers: `Authorization: Bearer {token}`
- Request body: Resume data and formatting options
- Response: URL to download the generated PDF

Authentication Flow

1. **Registration**:

    - User submits email and password
    - Password is hashed using bcrypt
    - User record is created in the database
    - JWT token is generated and returned

2. **Login**:

    - User submits email and password
    - Password hash is verified against database
    - JWT token is generated and returned

3. **Authentication**:

    - JWT token is included in the Authorization header
    - Token is verified on protected routes
    - User ID is extracted from the token for database queries

Development Guidelines

Code Style

- Use TypeScript for type safety
- Follow ESLint configuration for code style
- Use Prettier for code formatting

Git Workflow

1. Create feature branches from `main`
2. Use descriptive commit messages
3. Submit pull requests for review
4. Squash commits when merging

Testing

- Write unit tests for utility functions
- Write component tests for React components
- Write API tests for backend endpoints

Deployment

1. Build the frontend: `npm run build`
2. Start the backend: `node server/server.js`
3. Access the application at the configured port

Environment Variables

Required environment variables:

- `DB_USER` : PostgreSQL username
- `DB_HOST` : Database host address
- `DB_NAME` : Database name
- `DB_PASSWORD` : Database password
- `DB_PORT` : Database port (default: 5432)
- `PORT` : Server port (default: 5000)
- `JWT_SECRET` : Secret key for JWT tokens