Software Architecture Documentation: PyCars

1. Introduction

Project Idea: The "PyCars" project aims to develop a simplified, free web application for publishing and viewing car sale advertisements, inspired by established platforms such as Autovit or Openlane. The application will allow users to sell vehicles quickly, without fees imposed by PyCars, and allow buyers to filter and find desired cars.

Main Features: The project will be implemented as a web application, accessible from a browser.

- **For Buyers (Visitors):**
  - Viewing all active listings in a list or grid format;
  - Filtering functionality (based on make, model, year of manufacture, mileage, engine);
  - Sorting functionality for results (by price, year, mileage);
  - Detailed view of a listing;
- **For Sellers:**
  - Ability to create a listing;
  - Filling in specification fields (make, model, year, engine, mileage, power);
  - Ability to add a free-text description of the car;
  - Ability to upload up to 10 images of the vehicle;
  - Adding a phone number for contact;
  - Setting the price (fixed or negotiable value);
- **Optional Features:**
  - Auction system (to be implemented later);

Audience: The application is aimed at regular users in Romania who wish to sell or buy a car, whether new or second-hand, offering a free alternative to large platforms.

Technologies and Constraints: The application will be developed using **Python** for the backend logic. Due to its nature as a web application, we will use **HTML/CSS/JavaScript (TBD)** for the user interface (frontend). The database will be managed directly from Python.

2. System Overview

We will adopt a **Monolithic Client-Server** architecture. The main components are:

- **The Client:** This is the user's browser (Chrome, Firefox) which will render the interface.
- **The Server:** An application written in **Python**, using the **Flask** framework. It will contain all the business logic.
- **The Database:** An **SQLite** database, managed by the Flask server.

Basic Flow:

1. The seller opens the "add listing" page in the browser.
2. The browser makes a GET request to the Flask server.
3. The Flask server renders the HTML template of the form and sends it back to the browser.
4. The seller fills in the data, uploads the images, and presses "Publish".
5. The browser sends all the entered data to the server.
6. The Flask server receives the data, validates it, saves the images to disk, and inserts a new row in the listings table of the database.
7. The Flask server redirects the user to the newly published listing's page (or the main page).

Technologies:

- **Backend Language: Python.**
- **Backend Framework: Flask**. We chose Flask because it is a simpler and more approachable micro-framework for our experience level (beginners).
- **Database: SQLite**. It is included in Python, does not require a separate server, and stores the entire database in a single file, making it ideal for a project of this scope.
- **Frontend: HTML/CSS/JavaScript** (implemented at the simplest possible level, given our lack of prior experience).

3. Component Design

The system will be logically structured into three responsibilities, all orchestrated by the Flask framework:

a) Backend Component

- **Routing Module:**
  - The main component that defines the application's URLs.
  - It will manage HTTP requests.
- **Data Definition Module:**
  - To work cleanly with the SQLite database, we will use an **ORM** (Object-Relational Mapper), probably **SQLAlchemy** (via the Flask-SQLAlchemy extension).
  - We will define a Python class that represents our data structure.
  - This class will specify the fields a listing has (make, model etc), and the ORM will automatically translate this class into a database table.
- **Utility Modules:**
  - As the logic becomes more complex, we will extract it into separate functions.
  - A critical utility function will be image upload management: it will take the files sent by the user, validate them (to ensure they are .png or .jpeg images) and save them to a designated directory on the server disk.

b) Frontend Component

We will use a templating system (integrated into Flask) to dynamically generate the HTML pages.

- **Templates:**
  - We will have a collection of HTML files that act as "patterns."
  - **A base template (layout):** We will likely have a base file containing common elements to avoid repeating HTML code.
  - **The main page template:** Will contain the filter form to display each listing from a list received from the backend.
  - **The detail template:** Will display all information for a single listing.
  - **The form template:** Will contain the HTML form that the seller fills out to create a new listing.

c) Storage Component

- **Database:**
  - A single file (pycars.db) where all structured data will be stored: the listings and their details (make, model, price, description, path to images etc.).
- **File System:**
  - We need a directory (/uploads/) where we will physically save the unstructured files (the images). The database will only store the name or path to these files, not the images themselves.
  - We will also have a directory for the application's static files.

4. Deployment and Testing

Runtime Requirements:

- Python 3
- Python libraries (installable via pip from a installation_tool.txt file) in a virtual environment.
- The application will run locally on a defined port.

Testing Strategy: Testing will be performed manually by the team members. The testing steps include:

- Testing the listing publishing form (with valid data, invalid data, variable number of photos).
- Testing the filtering (verifying the correctness of the results for various filter combinations).
- Testing the sorting (ascending/descending by price, year, etc.).
- Testing the navigation (checking all links and the flow between pages).

5. Conclusions

Envisioned Complexity: The main challenge for the team will be simultaneously learning three fundamental concepts, given the limited experience:

1. Backend logic in Flask (routing, request handling, templates).
2. Interaction with the database using an ORM.

3. The basics of HTML/CSS/JavaScript for building the interface.

Estimated Work Hours: We estimate a minimum of **20+ work hours** per member, but we are aware that, including the learning curve, the actual time will likely be much higher.

Members:

- Ovidiu-Ștefan Costache
- Stefăniță Alexandru Stoica
- Vlad Năstase