Hangman Web App Architecture (Flask) - JSON Persistence

This document describes the Flask-based Hangman web application architecture, with an external JSON-based persistence store. It lists the functions and responsibilities for each container/component.

# Flask Web App Container - app.py

Purpose: Create and configure the Flask app, register blueprints, and handle app-level configuration.

- create\_app(config\_overrides=None)

- configure\_logging(app)

- register\_blueprints(app)

- load\_initial\_data(app)

# Flask Web App Container - routes.py

Purpose: Define HTTP endpoints and orchestrate interactions between the Game logic, Persistence, and Templates.

- index()

- new\_game()

- guess()

- admin()

- resume\_game()

# Templates (templates/)

Files: base.html, index.html, game.html, game\_over.html, admin.html

# Static (static/)

Files: style.css, main.js (optional), hangman\_stage\_0.png ... hangman\_stage\_6.png

# Game Logic (hangman/game.py)

Purpose: Core game rules and state management. Keep free of Flask imports for easy testing.

- class Game(word, max\_attempts=6)

- Game.start\_game()

- Game.make\_guess(letter)

- Game.display\_word()

- Game.is\_won()

- Game.is\_lost()

- Game.remaining\_attempts()

- Game.to\_dict()

- Game.from\_dict(data)

# Word Selection (hangman/word\_selection.py)

Purpose: Load and sanitise word lists and provide get\_random\_word().

- load\_words(source='data/words.json')

- sanitize\_word(word)

- filter\_words(words, min\_length=4, max\_length=10)

- get\_random\_word()

# Persistence Store (External - JSON files)

Purpose: Durable storage for word lists, saved games and optional scores. Implemented as JSON files under /data/.

- save\_game\_state(game\_id, data, path='data/games.json')

- load\_game\_state(game\_id, path='data/games.json')

- delete\_game\_state(game\_id, path='data/games.json')

- list\_saved\_games(path='data/games.json')

- load\_word\_list(path='data/words.json')

- save\_word\_list(words, path='data/words.json')

# Tests (tests/)

Purpose: Pytest-based unit and integration tests. Use fixtures for predictable test state.

- test\_game.py - unit tests for Game class

- test\_word\_selection.py - tests for word loading & filtering

- test\_routes.py - integration tests using Flask test\_client

# Data Flow Example

1. Player submits a guess via the browser to /guess (POST).

2. routes.py retrieves current game state from session or persistence via persistence helper.

3. routes.py calls Game.make\_guess(letter) to update state.

4. If persistence is enabled, routes.py invokes persistence.save\_game\_state() to persist the updated state.

5. routes.py renders game.html with the updated Game display data.

# Diagram – See Below

A diagram of a diagram

AI-generated content may be incorrect.