Guessarium Developer Reference Manual

Getting started (local dev)

Prerequisites

- Python 3.7+
- pip (or virtualenv)
- A modern browser for UI testing

Install & run

```
git clone https://github.com/KyziaPi/Lexis_WebApp.git
cd Lexis_WebApp
python -m venv .venv
source .venv/bin/activate  # (or .venv\Scripts\activate on
Windows)
pip install -r requirements.txt  # if requirements.txt lists
dependencies; otherwise at least install Flask
flask run   # or `python app.py`
# open http://127.0.0.1:5000
```

Project layout

```
Lexis_WebApp/
├─ app.py                 # Flask app + routes
├─ repl.py                # interactive REPL (if present)
├─ Interpreter/           # Lexis interpreter source (lexer,
parser, exec)
├─ WordBanks/             # .txt word bank files used by games
├─ static/
│  ├─ js/                 # snuzzle.js, filmster.js, raildle.js,
commands.js, etc.
│  ├─ css/
│  └─ images/
├─ templates/
│  ├─ layout.html
│  ├─ snuzzle.html
│  ├─ filmster.html
│  └─ raildle.html
└─ README.md
```

Application architecture & data flow

1. **Frontend UI** (HTML/CSS/JS) — user actions call JS functions (sendCommand, sendBatchCommands, initGame, resetGame, getCommands). JS sends AJAX POSTs to Flask endpoints, receives JSON results, and updates the UI.

2. **Flask web server** (`app.py`) — routes accept requests, call the Lexis interpreter to execute commands, update/return session state, and persist minimal session history for Raildle.
3. **Lexis interpreter** (`Interpreter/`) — parses and executes Lexis commands (file, start, word, guess, max_guesses, etc.), manipulates word bank files, and produces structured results used by the UI.

Lexis (language) reference - commands & formats

Modes & word bank formats

- **Letters Mode** — each line is a single word.
- **Hints Mode** — each line `word|hint1|hint2|...`.
- **Categories Mode** — first line: `word|cat1|cat2|...`; subsequent lines: `<word>|val1|val2|...`.
  (When categories are intended, the language requires categories to be defined first; otherwise the file may be treated as hints).

Play Mode (prompt `[Play] >>>`)

- `help` — show available commands.
- `file <filename>` — load a word bank file; the system infers mode from file format.
- `start` — start session (runs initial setup commands for current file/mode).
- `word` or `word <word>` — select secret word (random if none supplied; or set explicit secret if present in bank).
- `words` — list possible secret words.
- `max_guesses <number>` — set maximum attempts.
- `guess <word>` — submit a guess (mode-specific response).
- `show` — return the secret word.
- `quit` — exit interpreter.

Example responses:

- Letters mode: returns colored tile feedback (🟩, 🟨, ⬜).
- Hints mode: returns one preset hint and on failures reveals subsequent hints.
- Categories mode: returns per-category ✅ / ❌ with the value shown.

Edit Mode (`edit` prompt)

- `create <filename>` — create file (confirm overwrite if exists).
- `file <filename>` — open file for editing.
- `deletefile <filename>` — remove file.
- `add ...` — depends on mode:
  - Letters: `add <word>`
  - Hints: `add <word> | <hint1> | ...`
  - Categories: first `categories <name1> | <name2> | ...`, then `add <word> | <v1> | ...`.
- `list` — list words (and hints/categories).
- `edit <index> | <new_values>` — change entry.
- `delete <index>` — remove entry.
- `done` — return to play mode.

Flask endpoints (what to call from client)

- `GET /` — homepage (renders `layout.html`).
- `POST /game-redirect` — redirect to selected game (form POST).
- `POST /tutorial-redirect` — redirect to tutorial page.
- `GET /<game>/tutorial` — tutorial page for `<game>`.
- `GET /<game>` — load game page and set session (server renders HTML with `page` context).
- `POST /fetch/session/<game>` — fetch stored session commands for replay (Raildle uses this to restore progress).
- `POST /run/<game>` — execute a single Lexis command (payload contains command string). Returns structured output (results).
- `POST /run/<game>/batch` — run multiple commands sequentially (used during `initGame` on startup).
- `POST /reset_game/<game>` — clears session data + server side progress for game.

Client usage pattern:

- On page load `initGame()` calls `/run/<game>/batch` to run a standard setup (`file <bank>`, `start`, `max_guesses <n>`, `word`, `show`, etc.) and the server returns a list of command results which the client parses and uses to populate its UI.

Frontend JS API (what functions do)

These are the primary JS functions (names from README + doc snippets); use them to connect UI with server:

- `sendCommand(cmd, game)` — POST single command (string) to `/run/<game>`. Example: `await sendCommand("guess apple", "snuzzle");`. Returns JSON with `results` array.
- `sendBatchCommands(commands, game)` — POST array of commands to `/run/<game>/batch`. Used for startup sequences (load file, set max_guesses, set secret word, show). Example batch in README: `["file snuzzle.txt","start","max_guesses 6","word apple","show"]`.
- `initGame(game, max_guesses, secret_word)` — sets up game client state, calls `sendBatchCommands()` and/or `getCommands()` to replay session history (Raildle). Parses returned results to populate `secretWord`, `wordBank`, `hints`, etc. (See `snuzzle.js`, `filmster.js` logic in repo).
- `getCommands(game)` — calls `/fetch/session/<game>` to retrieve server-stored command history and returns array for replay. Used for session restore.
- `resetGame(game)` — calls `/reset_game/<game>` to clear server session and resets UI.

Frontend files to inspect:

- `static/js/snuzzle.js` — Wordle-style event handling, keyboard capture, tile animation & calls to `sendCommand()` / `initGame()`.
- `static/js/filmster.js` — handles hints, choices, parsing wordbank data returned by Lexis.
- `static/js/raildle.js` — Select2 dropdown integration, table update for category feedback, session replay logic.

Expected shape of interpreter server responses

The README describes command results being returned as a structured `results` array. A typical response (conceptual) looks like:

```
{

  "results": [

    {"command": "file snuzzle.txt", "result": "<raw file
contents or status>"},

    {"command": "start", "result": "game_started"},

    {"command": "word", "result": "APPLE"},             //
secret_word

    {"command": "show", "result": ["apple","amble"...]} // words
or info

  ],

  "status": "ok"

}
```

Note: Inspect `app.py` (server) to confirm exact JSON field names before writing strict client code.

How to add a new game mode

1. **Create a word bank file** in `WordBanks/` with the proper format for the new mode.
2. **Interpreter**: Add or extend a command handler in `Interpreter/` to implement the game-specific logic (parsing, feedback formatting). Place any new mode parsing/rules there.
3. **Flask**: If your new mode needs special endpoints, add routes in `app.py`, otherwise reuse `/run/<game>` and game parameters to distinguish logic.
4. **Frontend**: Add a template in `templates/` and add client JS (new `static/js/<newgame>.js`) that calls `initGame()` and handles returned results. Hook the new page to navbar/forms.

Developer tips & pitfalls

- **Session replay (Raildle)**: The frontend expects the server to store and return a chronological list of previous commands so the client can replay and restore the UI. Confirm commands are appended server-side in `/run/<game>`.
- **Word bank formats**: Categories mode requires first-line headers and consistent column counts. If categories aren't defined first, the interpreter may treat the file as hints. Always validate file format on upload.
- **Select2 initialization**: Raildle uses Select2; don't initialize the dropdown multiple times — that freezes the UI. Ensure initialization runs once per page load. (This was a noted issue in your project bug list.)
- **Mobile/responsive**: Snuzzle tiles and keyboard have breakpoints; test at multiple viewport widths. The repo contains media queries to shrink tile sizes.
- **Testing commands**: Use `repl.py` (if present) to try Lexis commands locally in isolation before wiring them through Flask. This makes debugging, parsing & execution easier.

Common troubleshooting & debug checklist

- *Server returns 500 for `/run/<game>`* — check `app.py` stack trace (look for absolute paths, missing WordBanks file, or interpreter import errors). Confirm `Interpreter/` is on `PYTHONPATH`.
- *Session not restoring* — confirm `/fetch/session/<game>` returns the array of commands and `initGame()` is calling `sendBatchCommands()` / replays them.
- *Hints not showing sequentially (Filmster)* — client-side `hintIndex` logic or server response ordering might be wrong; inspect `filmster.js` and the `results` array returned by the interpreter.
- *Dropdown freezes (Select2)* — ensure CSS z-index and container width do not conflict and initialize Select2 once.

Recommended improvements & extension ideas

- Add **unit tests** for the interpreter (Lexer/Parser/Executor) to assert outputs of `guess`, `start`, `word`, `file` commands.
- Replace session storage for large projects with a lightweight DB (SQLite) for persistence and audit.

- Add an **admin/CLI** for managing word banks and previewing category/hint parsing.
- Add CI (GitHub Actions) to run lints and interpreter tests on push.

Where to read code

- `app.py` — Flask endpoints and server glue.
- `Interpreter/` — interpreter internals (lexer, parser, exec functions). Start here for language semantics.
- `static/js/snuzzle.js`, `filmster.js`, `raildle.js` — game logic and client-side orchestration (UI updates, `initGame`, `sendCommand`, parsing results).
- `WordBanks/` — example banks (snuzzle.txt, filmster.txt, raildle.txt) to test parser behavior.

Useful examples

Example - start Snuzzle from client

```
// startup flow (client)

await sendBatchCommands([

  "file snuzzle.txt",

  "start",

  "max_guesses 6",

  "word",

  "show"

], "snuzzle");

// parse returned results for secret_word + word bank
```

Example - server command call (curl style)

```
curl -X POST http://127.0.0.1:5000/run/snuzzle \

  -H "Content-Type: application/json" \
```

```
-d '{"command":"guess apple"}'
```

Server should return JSON with a `results` entry describing guess feedback (colors or remaining guesses). Confirm exact JSON schema by inspecting `app.py`.

References

- Repository README and file list (contains developer section & API endpoints).
- `app.py` (Flask entry points).
- `Interpreter/` folder listing (interpreter implementation).
- `static/` (JS files, CSS, templates referenced in README).
- Project documentation (uploaded `PRL10 Project Documentation.docx`) — language spec (commands, modes, word bank formats, expected outputs).