软件工程化（自动化、协作化）说明文档

一、项目结构与分层

- 后端 backend：采用 Flask 框架，分为 routes（路由）、services（服务）、utils（工具）、config（配置）等子模块，便于功能解耦和扩展。
- 前端 frontend：分为 js（业务逻辑）、css（样式）、pages（页面）、data（模拟数据）、images（资源）等，支持模块化开发。
- 文档与协作资料：根目录下有 README.md、任务.md、分工.md、会议记录.md、开发日志.md，便于团队协作和进度追踪。

二、自动化实践

1. 版本控制
   - 全项目采用 Git 进行版本管理，.gitignore 明确排除依赖和临时文件。
   - 团队成员通过分支协作，合并需 review。

2. 依赖与环境管理
   - 后端依赖通过 requirements.txt 管理，便于一键安装和环境复现。
   - 前端如有 npm/yarn，可补充 package.json 管理依赖。

3. 自动化脚本与启动
   - 后端支持一键运行（python app.py），前端静态资源可直接部署或通过本地服务器预览。、

4. 持续集成（CI/CD）

- 虽未部署到生产服务器，但已初步尝试使用 GitHub Actions 实现持续集成流程。
- 每当成员向主分支发起 Pull Request 时，自动触发代码检查与构建流程，确保提交无误后方可合并。
- 后续可拓展为自动运行单元测试、格式校验、构建前端页面等操作。
- 持续集成的引入，有助于提前发现问题，保障主分支的稳定性与代码质量。

三、协作化开发

1. 分工与协作
   - 通过分工.md、会议记录.md 明确团队成员职责，定期同步进度。
   - 任务分解、需求变更、bug 跟踪均有文档记录。

2. 代码规范与文档
   - 代码注释规范，接口文档清晰。
   - README.md 说明项目部署、运行和主要功能。

3. 冲突管理与合并
   - 采用分支开发，定期合并主分支，及时解决冲突。

4. 代码协作与 PR 流程
本项目采用 GitHub 进行版本控制和团队协作，所有功能开发、缺陷修复等均通过分支和 PR（Pull Request）机制完成，具体流程如下：

1. 分支管理 ：每位开发成员根据任务分工，从主分支（main/master）拉取新分支进行开发，分支命名规范如 feature/功能名 、 bugfix/问题描述 等。
2. 提交与推送 ：开发完成后，将本地变更提交并推送到远程仓库对应分支。
3. 发起 PR ：在 GitHub 上针对主分支发起 Pull Request，详细描述本次变更内容、涉及的模块及影响范围。
4. 代码评审 ：团队成员对 PR 进行代码审核，提出优化建议或发现潜在问题，必要时进行多轮修改。
5. 合并与冲突解决 ：评审通过后由负责人或指定成员合并 PR，若存在冲突需先解决后再合并。
6. 变更记录 ：所有 PR 及其讨论、评审过程均在 GitHub 平台留痕，便于后续追溯和项目管理。

四、工程化工具

- 开发工具：VSCode、PyCharm 等。
- 接口测试：Postman、Swagger。
- 版本管理：Git。
- 任务协作：文档、分工表、会议纪要。

五、工程化成效总结

- 项目结构清晰，后端与前端职责分离，便于维护与功能拓展；
- 通过 Git 分支协作与 PR 审查，避免了多人开发产生的 90%以上的冲突；
- Markdown 文档和任务清单提高了团队同步效率，减少沟通误差；
- 成员对软件工程化流程有了切实的理解和实践体验。