Architectural Design Record

Documentation of important design decisions made during the course of the project.

Decision 1: Monolith vs Microservices

Decision: A monolith approach was chosen over a microservices approach.

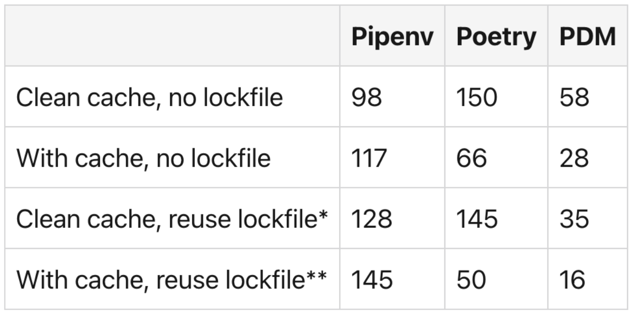
Reasoning: This was done as the project is relatively small scale and had a limited and clearly defined scope. A microservices approach would be too complex for a project of this scale, and could potentially cause more issues (eg. with communication between services). A monolith was regarded as the simpler, safer choice for a project that does not necessarily need to scale.

Decision 2: Choice of dependency manager

Decision: Using PDM as the dependency manager.

Reasoning: Conda and other package/dependency managers proved to be unreliable, as members had issues getting the project running using them. PDM proved to be more reliable and performant, which we thus used.

Related Information:



	Pipenv	Poetry	PDM
Clean cache, no lockfile	98	150	58
With cache, no lockfile	117	66	28
Clean cache, reuse lockfile*	128	145	35
With cache, reuse lockfile**	145	50	16

[Source: https://dev.to/frostming/a-review-pipenv-vs-poetry-vs-pdm-39b4]

Decision 3: Choice of Flask over JS

Decision: Use Flask

Reasoning: While the project was initially supposed to be done in JavaScript, members' inexperience with JS was a potentially large hurdle to overcome when combined with relative inexperience with front-end development and the limited time available. Flask was chosen due to teams' familiarity with python, the framework itself, and the following considerations:

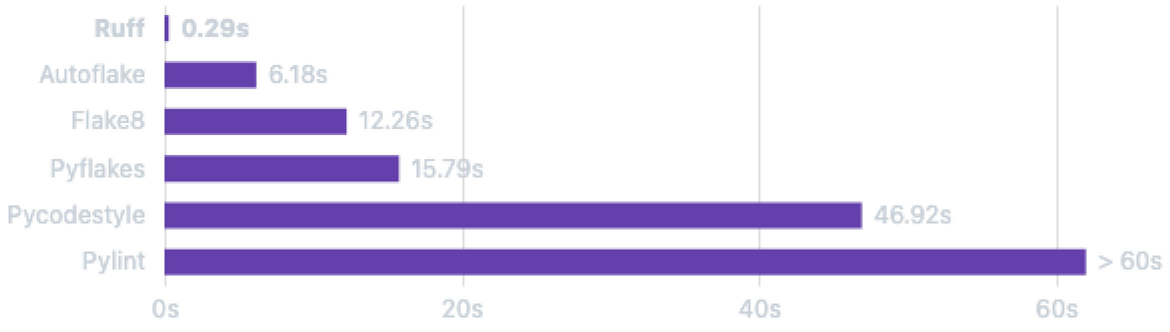
Flask offers several advantages that align with Software Engineering principles

- Separation of Concerns : Flask uses the Jinja2 templating engine, promoting the separation of business logic and presentation.
- Modularity : Developers can choose and integrate different components, libraries, and extensions based on their project requirements.
- Widely Compatible : Flask can be integrated with various databases, front-end frameworks, and other tools.
- Testing : Flask provides built-in support for testing, allowing developers to write unit tests for their applications more easily.

Decision 4: Ruff Linter over Other Python Linters

Decision: Use Ruff Linter

Reasoning: Ruff is super fast and integrates the functionalities of several linters. Therefore, it became a natural choice for us to utilize this. It is also extremely simple to set up exceptions and customisations.



Related Information: https://github.com/astral-sh/ruff

Decision 5: DJLint

Decision: Use DJLint.
Reasoning: We decided to use DJLinter instead of traditional HTML linters as DJLint has jinja templating specific linters.
Relevant Link: https://www.djlint.com/

Decision 6: Super Linter

Decision: Use the Slim image for superlinter.
Reasoning: We decided to use the slim image for super linter as it is much smaller in size, faster and doesn't run linting against some of the languages we are not using in this project.
Relevant Link: https://github.com/super-linter/super-linter

Decision 7: Automatic Documentation Generation

Decision: Use Sphinx for Automatic Documentation Generation.
Reasoning: There is plenty of discussion regarding Mkdocs vs Sphinx for automatic documentation generation. We have decided to move forward with Sphinx due its ease of usage, robustness and the feature set.
Relevant Links:

- https://github.com/encode/httpx/discussions/1220
- https://www.reddit.com/r/Python/comments/kidjd9/mkdocs_vs_sphinx_for_static_site_generation/

Decision 8: View vs Interact with Posts on Feed Amalgamator

Decision: Support only viewing posts and not interact (comment/like/etc) with posts on Feed Amalgamator.
Reasoning: We decided to limit our scope and stick to just displaying posts from various servers. Interaction will complicate the application logic and will consume more time for implementation. Interacting with posts will be a high priority in (hypothetical) future sprints

Decision 9: Containerisation

Decision: Use Docker to containerize the application.
Reasoning: We decided to containerize the application so that it becomes extremely easy to deploy the application without the "works on my machine problem". There are several advantages of containerisation that we will not mention here (you can find more info here:https://circleci.com/blog/benefits-of-containerization/). We anticipate that doing this will help us to deploy the application with ease.

Decision 10: Deployment Platform

Decision: Use Azure App Service.

Reasoning: We explored alternatives such as Railway, Heroku, Render and Fly. We did not find a free tier or the platform was not conducive for hosting our application and therefore decided to pick Azure App Service for deploying the application. We also used Azure SQL DB for hosting the DB and Azure monitoring services for doing availability checks. All these features help us to better observe the application and have everything hosted on a single platform.

Decision 11: Structured Logging

Decision: Use ECS (Elastic Common Schema) structured logging format

Reasoning: Structured logs (in json format) make querying fields in logs significantly easier. This makes monitoring of the app easier to perform as opposed to logs stored in the default string format. We chose to use the ECS as ElasticSearch+Kibana is a common way to process and monitor logs. However, should another log format be chosen in the future, the change is simple to implement: All one has to do is change the log format in the centralized logging_helper class.

Decision 12: Testing Limitations

Decision: Tests that automate the generation of new clients (bots) will not be performed.

Reasoning: If a new bot is created every time a unit test is run, Mastodon servers will get very unhappy with us.