DASS Quiz 6

Manvith Reddy
Project - Reddit for Research
TA Mentor - Vijayraj Shanmugaraj

Separation of Concerns

A concern is a feature or behavior that is specified as part of the requirements model for the software. The fundamental concept of separation of concerns is achieved at multiple levels within our project. At the highest level, our project is divided into numerous sub-projects :

1. Reddit for Research Webapp (MEAN Stack)
2. ElasticSearch Webapp (MERN Stack)
3. Machine Learning based Search Webapp (Flask microframework)
4. Data Science Pipelining based on Kubeflow and Nuclio.io
5. Continuous Integration using AWS

There is a further "Separation of Concerns" within each of these sub-projects.

Reddit For Research Webapp - The primary webapp is split into multiple components to enable different team members to work on different problems. This eases development difficulties like inter-team communications and speeds up the development life cycle. The main product is fundamentally split into the PDF Viewer, Annotation, Highlighting, User Interaction and the Search Engine. These concerns are further split into various smaller, and therefore more manageable pieces, meaning that the problem takes less effort and time to solve.

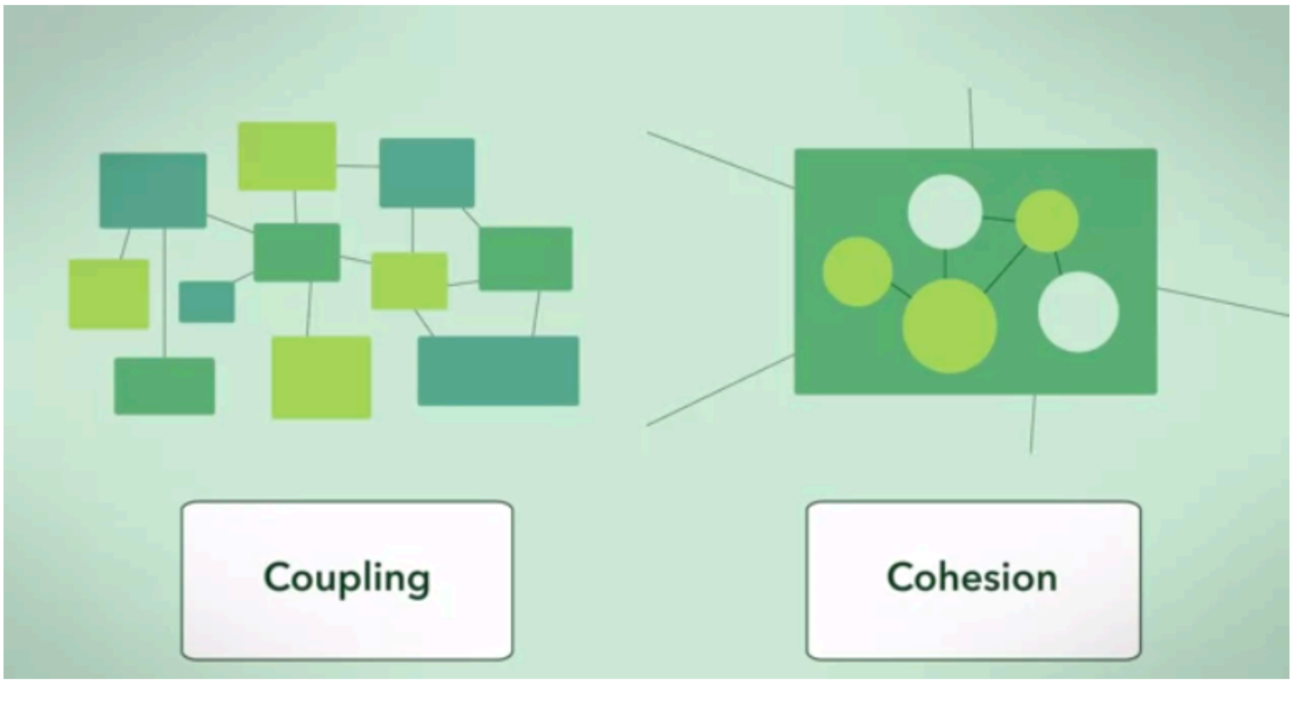
Search - The Search concern is further split into two different approaches. The ML Search and ElasticSearch. Within these Search Engines the concerns are further split into basic search, database management, advanced search filters, etc. This level of modularity speeds up development time as well as debugging.

Pipelining - Pipelining was explored using two different approaches (Nuclio.io and Kubeflow). Pipelining has various sub-concerns like Storage Management, Data Flow, Cloud Computation, etc. All these various concerns were split and treated as individual components of the pipeline.

Continuous Integration/Deployment - Various sub-concerns in CI/CD include setting up a code pipeline, integrating with Git WebHooks, writing unit tests and system tests, and deploying on cloud service providers.

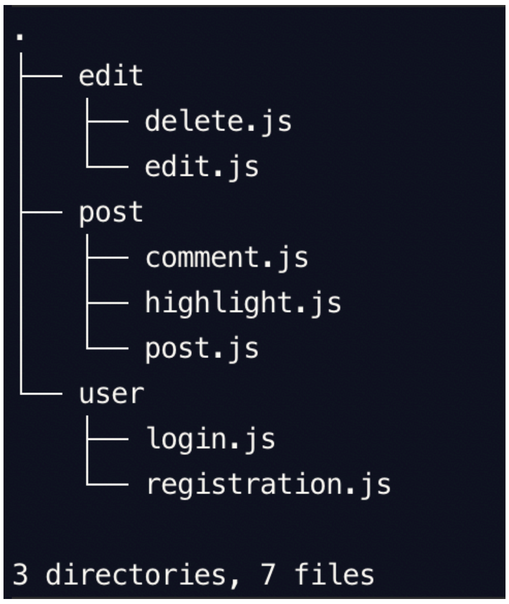
High Cohesion and Low Coupling

The image below describes the fundamental understanding of Coupling and Cohesion. It is important to have low coupling
(less dependencies between various classes) and high cohesion (A stronger relationship between methods of the same class).



Given below are various examples of low coupling and high cohesion best practices that are followed in our code.

- Usage of different directories for routes, static items, database models, etc
- Each file in the routes directory contains APIs that share a common purpose
    - APIs for login/registration
    - APIs for posting/commenting/highlighting
    - APIs for editing/deletion, etc
- Different (unrelated) aspects of the frontend are separated into different Angular components
- Inter-related aspects of the frontend are combined into individual Angular components



```
.
├── edit
│   ├── delete.js
│   └── edit.js
├── post
│   ├── comment.js
│   ├── highlight.js
│   └── post.js
└── user
    ├── login.js
    └── registration.js

3 directories, 7 files
```

The image represents the API directory structure.

These policies extend to the search apps as well. However, those aren't listed, as the Reddit-for-Research webapp is the primary focus of the project.
In this manner, elements within particular modules are all directly related to the functionality that module is meant to provide. Duplication of knowledge in the modules is reduced, as related code modules are all close to each other.

Low coupling allows changes to the internal workings of particular modules without impacting other modules in the system. It also allows for reusable modules. Bugs/issues are also isolated to self-contained units of code.

Abstraction

In the Reddit for Research webapp, the backend uses API endpoints which are exposed to the users. The functioning of the backend is not exposed to the end-users, who are only allowed to interact with the frontend (which, in turn, interacts with the APIs). In this manner, the alterations or improvements to the infrastructure behind the endpoint aren't noticed by the users (frontend) that rely on that API.

Similar levels of abstraction are achieved on the database layer. Records are stored as fields and attributes along with their data types in a MongoDB database. This is the **logical level** of the database layer.

At the **view level**, users just interact with the front-end, and aren't aware of how and where the data is stored, and what format the data is stored in. In this manner, data abstraction is achieved at the database layer.

The ML Search app is built using Flask and Python3. The APIs and the nature of Flask (classes) lead to logical data abstraction.
The Elastic Search app is built using MERN. The usage of React components (which are self-made classes) is an example of data abstraction. It allows for reusable frontend systems and abstractions.

Finally, the data science pipeline using Nuclio (which is part of a sub-project to evaluate multiple tools' viability for data science pipeline) deploys serverless functions which are independent of each other. Each serverless function could represent one step in the pipeline, and the output of one step is fed into the next step using one Jupyter notebook for orchestration. Each function is independent from the others and the end-user is not allowed access to any of the functions, which is how data is abstracted in the pipeline.