Architecture Overview

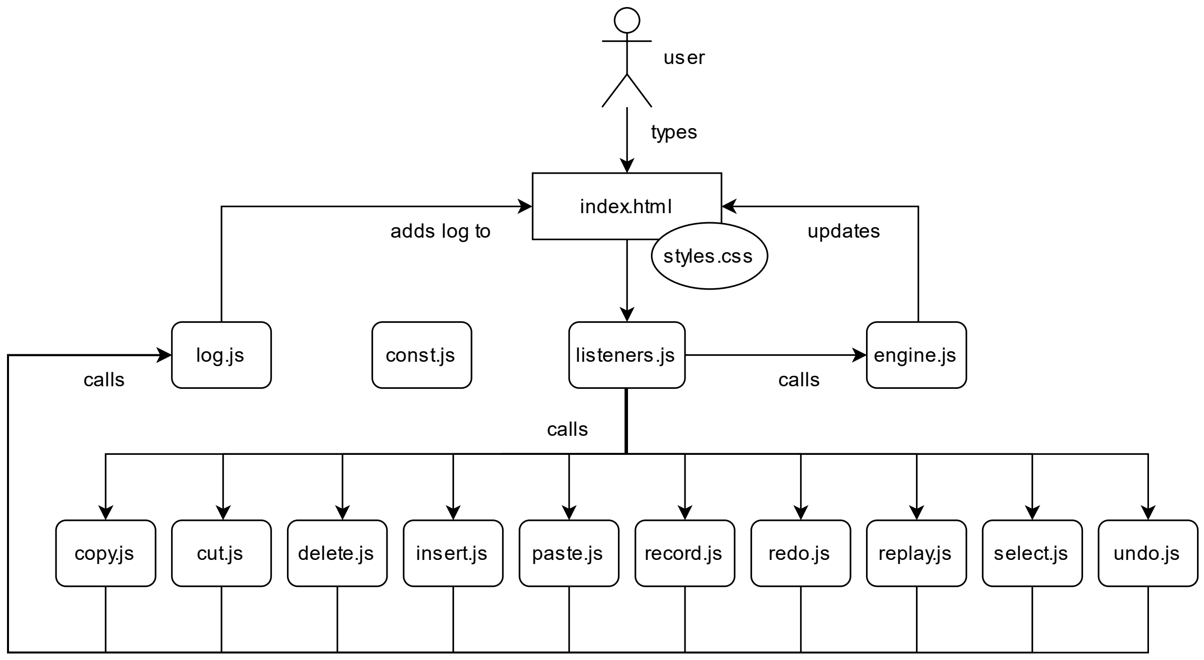
The architecture of the **Mini Text Editor** is built on a layered model, with clear separation between the frontend, REST API, and backend. The frontend handles user interaction and communicates with the backend through the REST API. The backend, written in Java, handles the core logic and state management, while the REST API serves as the interface for communication between the frontend and backend.

This section explains the overall architectural structure of the application, explaining the key components, their interactions, and the design pattern implementations used to ensure maintainability, scalability, and performance.

Click here to get back to the README.

Table of Contents

Frontend



HTML and CSS components

The `index.html` contains the graphical user interface (GUI) exposed to the user via the browser. The GUI consists of a text area to edit text, a log area to log changes of the engine state, and buttons to record and replay actions.

The `styles.css` contains all the style elements that create the visual design of the text editor frontend.

Engine components

`const.js` contains all objects that are initialized once during runtime, e.g. the editor, the engine state object, and the buttons.

Each input by the user is centrally tracked by the `listeners.js`. After each action by the user, may it be an input or a command, it calls the corresponding command function.

After successfully calling and executing a command function, `engine.js` is called to fetch the latest engine state from the server, and overwrite the local engine state object to match the editor text area with the newest engine state in the backend.

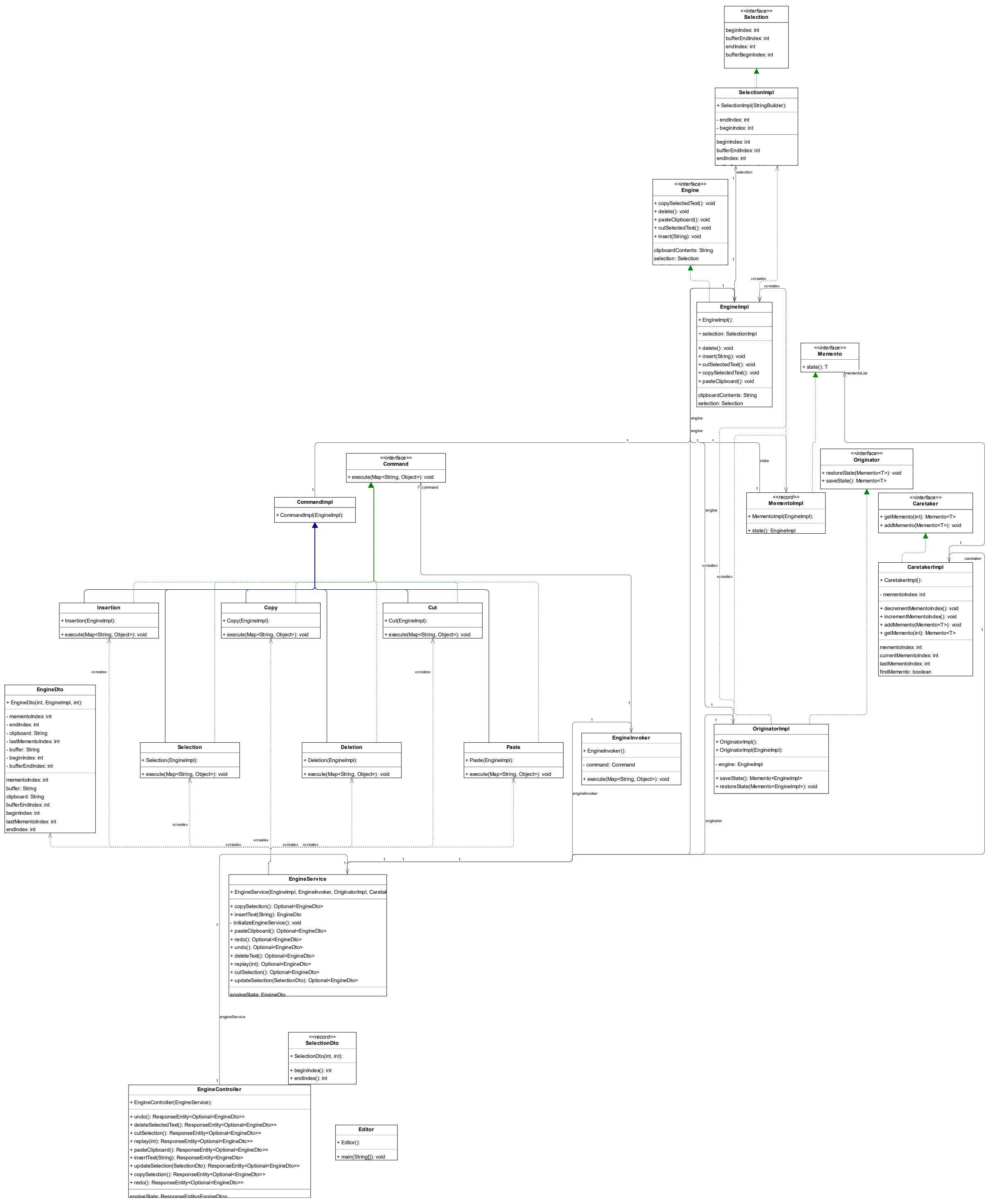
Command components

Each command, e.g. `copy.js`, interacts with the REST API by preparing and executing the call, awaiting and returning a response or handling an error.

Logging

After a command is successfully executed, a log is created using `log.js`, which is appended to the log area in the GUI.

Backend



<<interface>> Selection
beginIndex: int
bufferEndIndex: int
endIndex: int
bufferBeginIndex: int
SelectionImpl
+ SelectionImpl(StringBuilder):
- endIndex: int
- beginIndex: int
beginIndex: int
bufferEndIndex: int
endIndex: int
<<interface>> Engine
+ copySelectedText(): void
+ delete(): void
+ pasteClipboard(): void
+ cutSelectedText(): void
+ insert(String): void
clipboardContents: String
selection: Selection
EngineImpl
+ EngineImpl():
~ selection: SelectionImpl
+ delete(): void
+ insert(String): void
+ cutSelectedText(): void
+ copySelectedText(): void
+ pasteClipboard(): void
clipboardContents: String
selection: Selection
<<interface>> Memento
+ state(): T
<<interface>> Command
+ execute(Map<String, Object>): void
CommandImpl
+ CommandImpl(EngineImpl):
<<interface>> Originator
+ restoreState(Memento<T>): void
+ saveState(): Memento<T>
<<record>> MementoImpl
+ MementoImpl(EngineImpl):
+ state(): EngineImpl
<<interface>> Caretaker
+ getMemento(int): Memento<T>
+ addMemento(Memento<T>): void
CaretakerImpl
+ CaretakerImpl():
- mementoIndex: int
+ decrementMementoIndex(): void
+ incrementMementoIndex(): void
+ addMemento(Memento<T>): void
+ getMemento(int): Memento<T>
mementoIndex: int
currentMementoIndex: int
lastMementoIndex: int
firstMemento: boolean
Insertion
+ Insertion(EngineImpl):
+ execute(Map<String, Object>): void
Copy
+ Copy(EngineImpl):
+ execute(Map<String, Object>): void
Cut
+ Cut(EngineImpl):
+ execute(Map<String, Object>): void
EngineDto
+ EngineDto(int, EngineImpl, int):
- mementoIndex: int
- endIndex: int
- clipboard: String
- lastMementoIndex: int
- buffer: String
- beginIndex: int
- bufferEndIndex: int
mementoIndex: int
buffer: String
clipboard: String
bufferEndIndex: int
beginIndex: int
lastMementoIndex: int
endIndex: int
Selection
+ Selection(EngineImpl):
+ execute(Map<String, Object>): void
Deletion
+ Deletion(EngineImpl):
+ execute(Map<String, Object>): void
Paste
+ Paste(EngineImpl):
+ execute(Map<String, Object>): void
EngineInvoker
+ EngineInvoker():
- command: Command
+ execute(Map<String, Object>): void
OriginatorImpl
+ OriginatorImpl():
+ OriginatorImpl(EngineImpl):
- engine: EngineImpl
+ saveState(): Memento<EngineImpl>
+ restoreState(Memento<EngineImpl>): void
EngineService
+ EngineService(EngineImpl, EngineInvoker, OriginatorImpl, Caretak
+ copySelection(): Optional<EngineDto>
+ insertText(String): EngineDto
- initializeEngineService(): void
+ pasteClipboard(): Optional<EngineDto>
+ redo(): Optional<EngineDto>
+ undo(): Optional<EngineDto>
+ deleteText(): Optional<EngineDto>
+ replay(int): Optional<EngineDto>
+ cutSelection(): Optional<EngineDto>
+ updateSelection(SelectionDto): Optional<EngineDto>
engineState: EngineDto
<<record>> SelectionDto
+ SelectionDto(int, int):
+ beginIndex(): int
+ endIndex(): int
EngineController
+ EngineController(EngineService):
+ undo(): ResponseEntity<Optional<EngineDto>>
+ deleteSelectedText(): ResponseEntity<Optional<EngineDto>>
+ cutSelection(): ResponseEntity<Optional<EngineDto>>
+ replay(int): ResponseEntity<Optional<EngineDto>>
+ pasteClipboard(): ResponseEntity<Optional<EngineDto>>
+ insertText(String): ResponseEntity<EngineDto>
+ updateSelection(SelectionDto): ResponseEntity<Optional<EngineDto>>
+ copySelection(): ResponseEntity<Optional<EngineDto>>
+ redo(): ResponseEntity<Optional<EngineDto>>
engineState: ResponseEntity<EngineDto>
Editor
+ Editor():
+ main(String[]): void
selection
engine
state
mementoList
command
caretaker
engineInvoker
originator
engineService
«create»
1

Components

The following components form the backend of the application and handle essential functionalities like text manipulation, clipboard management, selection handling, and undo/redo operations.

Engine

The **Engine** is responsible for managing the text buffer, selection, and clipboard operations. It is the heart of the text editor, handling the core logic for editing text.

- **Interface**: `Engine`
  - Defines the methods needed for the engine of a simple text editor.
- **Class**: `EngineImpl`
  - Implements the `Engine` interface to interact with the engine state.
  - Manages the **text buffer**, where the main text is stored.
  - Handles **clipboard operations**, such as copy, cut, and paste.
  - Maintains an instance of **Selection**, the currently selected portion of the text in the buffer.
  - Supports operations like `insert`, `delete`, `copySelectedText`, `cutSelectedText`, and `pasteClipboard`.
- **Class**: `EngineInvoker`
  - Responsible for invoking commands, decoupling the actual execution of commands from the application logic.
- **Class**: `EngineDto`
  - Defines a class consisting only of primitive types and Strings to easily parse the engine state to JSON in the frontend.

Selection

The **Selection** component is responsible for managing the currently selected portion of text in the buffer. It helps tracking the start and end of the selection and interacts with the `EngineImpl` to update and use the selection.

- **Interface**: `Selection`
  - Abstracts the concept of selecting text with methods to get and set selection indices.
- **Class**: `SelectionImpl`
  - Implements the `Selection` interface to manage the start (`beginIndex`) and end (`endIndex`) of the selection.
- **Class**: `SelectionDto`
  - Defines a class consisting of primitive types for parsing the `beginIndex` and the `endIndex` when updating the selection via the API.

Command

The **Command** pattern is implemented to encapsulate text editing actions, such as copy, cut, paste, and undo, into individual command objects. This decouples the request for an operation from its actual execution.

- **Interface**: `Command`
  - Defines a common interface for all commands with a method `execute()` .
- **Class**: `CommandImpl`
  - Defines a common constructor for concrete commands`.
- **Concrete Command Classes**:
  - **`Copy`** : Copies the selected text to the clipboard.
  - **`Cut`** : Copies the selected text to the clipboard and removes it from the text buffer.
  - **`Paste`** : Pastes the clipboard content at the current selection.
  - **`Insertion`** : Inserts new text at the current selection.
  - **`Deletion`** : Removes the selected text from the buffer.
  - **`Selection`** : Updates the selection indices based on user input.

Memento

The **Memento** pattern enables undo and redo functionality by capturing and storing snapshots of the application's state.

- **Interface**: `Memento<T>`
  - Defines the state of an object at a given point in time.
- **Class**: `MementoImpl`
  - Stores snapshots of the state of `EngineImpl` . These snapshots include:
    - The current content of the text buffer.
    - The selection indices (start and end).
  - The clipboard contents are **not** saved.
- **Interface**: `Originator<T>`
  - Defines the methods required to save and restore a memento.
- **Originator**: `OriginatorImpl`
  - Responsible for creating new mementos based on the current state of the `EngineImpl` , and restoring the engine's state from a given memento.

- **Caretaker**: `Caretaker<T>`
  - Defines the essential methods for managing the history of a memento.
- **Caretaker**: `CaretakerImpl`
  - Manages a list of `Memento` objects. It tracks the current position in the undo/redo history and provides methods to navigate backward (undo) or forward (redo) through the history of mementos.

Service and Controller Layers

The **Service** and **Controller** layers act as the intermediary between the core engine and the frontend, providing a structured way to interact with the backend.

EngineService

The **EngineService** handles the execution of commands and manages undo/redo functionality by receiving requests from the **Controller** and interacting the **Engine** accordingly.

- **Class**: `EngineService`
  - Executes commands using the `EngineInvoker`.
  - Manages the undo/redo history using the `CaretakerImpl` and `OriginatorImpl`.
  - Provides higher-level operations such as managing the clipboard, updating selections, and replaying commands.

EngineController

The **EngineController** is responsible for exposing the core functionality of the text editor as a RESTful API. It receives HTTP requests, delegates them to the service layer, and returns the results.

- **Class**: `EngineController`
  - Exposes various endpoints like `/copy`, `/undo`, and `/redo`.
  - Routes incoming requests to the appropriate methods in `EngineService`.

Component relationships

- **Engine and Selection**:
  - The `EngineImpl` depends on `SelectionImpl` to manage the selection within the text buffer, ensuring that the correct portion of the text is

selected during operations like copy, cut, and paste.

- **Commands and Engine**:
    - Each command (e.g., `Copy`, `Cut`) interacts with `EngineImpl` to execute its specific task. Commands are executed through `EngineInvoker`, allowing for decoupled execution.
- **Undo/Redo Management**:
    - `CaretakerImpl` manages the history of `Memento` objects, allowing the application to navigate through previous states for undo and redo operations.
    - `OriginatorImpl` creates and restores mementos based on the current state of `EngineImpl`.
- **REST API**:
    - `EngineController` communicates with `EngineService`, which in turn interacts with the `EngineInvoker` to execute commands. The current state is managed by the `CaretakerImpl` and `OriginatorImpl`.