FPO-Editor Documentation

Documentation for the Fachprüfungsordnung Editor

2026-03-31

Contents

1 Overview

Documentation for the Fachprüfungsordnung editor.

- Setup
- Backend (haddock)
- Frontend (pursuit)
- Language

2 Configure

All services are orchestrated using Docker Compose.

Deployment of the service should happen by using the `docker-compose.ssl.yaml` as an override to have SSL properly set up, see the appropriate section below

2.1 Technologies

- Docker

- PostgreSQL for the database
- pgweb as a database client
- nginx

2.2 Overview

- the frontend user interface is available at `/`, e.g., /
- the backend api is available at `/api/`, e.g., /api/
- internally, the services are reachable by their respective hostnames:
  - Postgres: `postgres`
  - Backend: `api`
- a collection of development tools and links is available at `/dev/`, e.g., /dev/
  - a simple database client is available at `/pgweb/`, e.g., /pgweb/. For deployment, this should be password protected, see Environment
  - the API documentation is available at `/swagger/`, e.g., /swagger/
  - the documentation hosted in its entirety at `/docs/`, e.g. /docs/
    - the hosted backend documentation is available at `/dev/haddock/`, e.g. /dev/haddock/
    - the hosted frontend documentation is available at `/dev/purs/`, e.g. /dev/purs/
  - backend logs can be found at `dev/logs.html`, e.g. /dev/logs.html. You can login here with any credentials that are also superadmins within the application.

2.3 Quick Setup

1. Install Docker.
2. Create a `.env` file similar to `.env.example` in the root directory.
3. Start all services via `docker compose up`.

2.4 Environment

The `.env` file should contain all settings. An exemplary `.env` file for can be found in the root directory within `.env.example`. The `.env` file should set the following environment variables:

2.4.1 General

- `PORT` defines the port at which the web application will be available

2.4.2 Postgres

- `POSTGRES_DB` defines the name of the postgres database
- `POSTGRES_USER` defines the name of the postgres user account
- `POSTGRES_PASSWORD` defines the password for the postgres user account

2.4.3 Mail configuration (optional)

These secrets are needed if you want to use the mail service within the backend. Currently, only STARTTLS is supported.

- `MAIL_ADDRESS` defines the mail address that the mails will be sent from
- `MAIL_HOST` defines the host for the mail server
- `MAIL_PORT` defines the STARTTLS port of the mail server
- `MAIL_USERNAME` defines the username for the specified mail servef
- `MAIL_PASSWORD` defines the password for the specified user

2.4.4 Pgweb (optional)

The following settings enable basic auth for pgweb. To disable basic auth, leave these variables unset. In production, these should be set.

- `PGWEB_AUTH_USER` defines the username for the pgweb database client
- `PGWEB_AUTH_PASS` defines the password for the pgweb database client

2.5 Certbot and SSL

To set up SSL certificates, you can use `scripts/install-certs` to request a certificate from the certificate authority and `scripts/renew-certs` to update these. Note you have to set up the certificates once manually yourself using these scripts. The certificates are then used by the Nginx container in a production environment by using the `docker-compose.ssl.yaml` override. You can invoke the override via

```
docker compose -f docker-compose.yaml -f docker-compose.ssl.yaml up --build
```

`scripts/renew-certs` should optimally be set up to be executed by a cronjob periodically.

3 GitHub Actions

The repository contains GitHub actions to test and deploy the code. The actions are configured to run on self-hosted runners. We used the Docker runners from myoung34 to run the actions. Deployment happens via SSH to two different machines (a dev environment and a production enviroment, which is only deployed to on tagged commits on `main`). The setup works, although it could be improved to use a Docker container registry in the future.

3.1 Secrets

To get the actions up and running, there are a few secrets and variables to set up:

3.1.1 Deployment server secrets

The current `main` branch is, pending passing tests, autodeployed to the dev server via SSH. To get this to work, you'll need the following secrets.

- `DEV_REMOTE_HOST`: The hostname of the development server to SSH into
- `DEV_REMOTE_PASSWORD`: The development server SSH password
- `DEV_REMOTE_USER` The development server SSH user
- `DEV_SSH_PORT`: The development server SSH port
- `DEV_SERVER_HOST`: The URL the development server will be reachable at

Similar properties exist for the production deployment server, namely:

- `REMOTE_HOST`
- `REMOTE_PASSWORD`
- `REMOTE_USER`
- `SSH_PORT` and
- `SERVER_HOST`

Deployment to the production server will only happen with commits on `main` tagged with the `prod` tag. Use the `scripts/deploy-prod` script to quickly trigger a deployment.

3.1.2 Mail configuration settings

These secrets are needed if you want to use the mail service within the backend. These are currently supplied by the action, but could also be passed via the `.env` file.

- `MAIL_ADDRESS`
- `MAIL_HOST`
- `MAIL_PASSWORD`
- `MAIL_PORT`
- `MAIL_USERNAME`

4 Backend

The backend consists of the following components, the functionality of which the frontend can access via the REST API: - LTML Parser with HTML and PDF Backends, following the Language Design - Version Control & Document Management - User Management

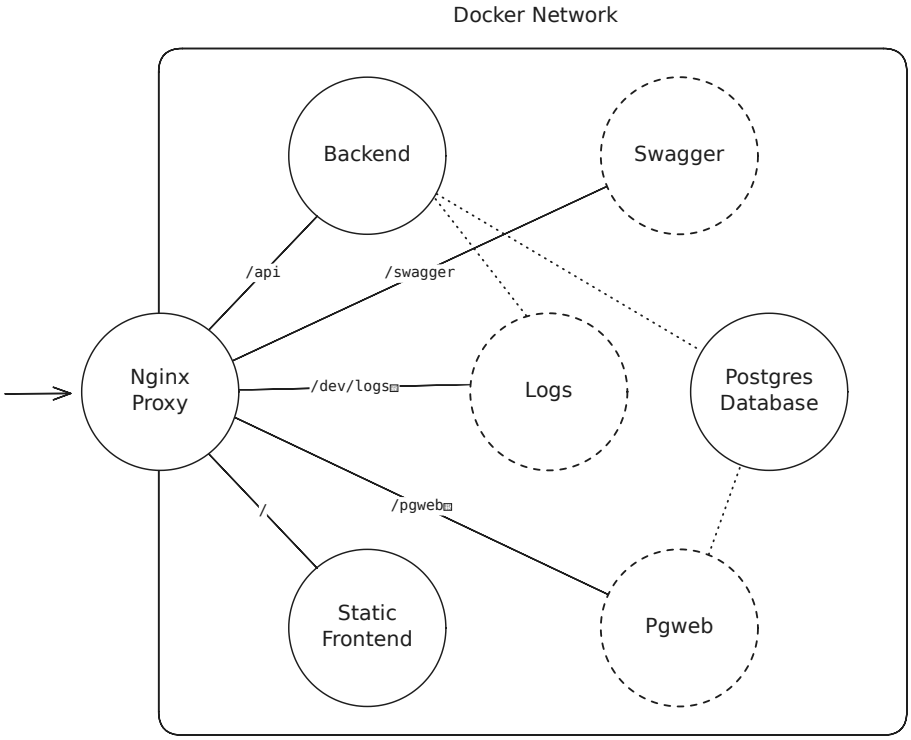
For code documentation, see haddock.

For API documentation, see swagger.

4.1 Architecture

All services are behind an nginx reverse proxy.

The following figure shows how the services are orchestrate. Dottet lines represent internal communication, solid lines are externally reachable through the nginx reverse proxy. The labels represent the locations.



5 Version Control & Document Management

The Backend provides an API to create, edit, manage and version documents. The documents can be stored in any database, the current implmenetation uses PostreSQL.

5.1 Document Management

A FPO instance can manage multiple documents. Each document is owned by a *group*. Documents are visible and editable by each *member* of the group. External reviewers and editors can be manually added to a document (this is not yet added to the system, but planned).

Admins and Superadmins can manage their groups and, consequently, the projects within those groups. Core functionalities include:

- Creating new projects
- Archiving projects (TODO)

- Deleting projects
- Assigning document permissions to users

Regular users of the application receive an overview of the projects they are assigned to upon login and can access them based on their permissions.

- Documents are assigned to a single group
- All group members always have full access to all documents within the group
- Individual external users can be added to collaborate on specific documents of the group
  - The following permission levels exist: `Read`, `Review`, `Edit` (those rights are also planned, but not yet added)

	Read	Review	Edit
Read	**yes**	**yes**	**yes**
Comment	no	**yes**	**yes**
Write	no	no	**yes**
Rearrange Paragraphs	no	no	**yes**

Commenting also includes suggesting changes.

For more detailed information, see user management.

5.2 Documents

A document consists of a *tree structure* and *text elements*. The tree structure is closely resembled by the Table of Contents. The structure of a document and the contents of the text elements are strictly separated. Text elements are referenced in leaf nodes of the structure tree only. The leaf nodes only contain a reference to a text element, but do not contain any actual text.

5.3 Version Control

Consequently, the overall structure of the document, the structure tree, has a revision history, but all text elements also have their own revision histories. Thus, there are two kinds of revision: *tree revisions* and *text revisions*. The API also exposes a third kind of revision, a *document revision*. Document revision however do not exist as such, each document revision is either a tree or a text revision. Document revisions are used to obtain all revisions made to a document in chronological order. For more information, see GET /docs/{documentID}/history

5.3.1 Noteworthy Implications

This version control model has some implications worth noting:

A revision to the structure tree does not contain any information on specific text element revisions. It is however possible to obtain the full document including text element revision at the state of a specific revision, either tree or text (see GET /docs/rev/{revision} and all other endpoints at `/docs/rev`).

A text element has to be valid for each tree version it is part of. Thus, a text elements *kind* and *type* (for more information, see language) are immutable. To guarantee this immutability, the type and kind of a text element are not part of the text revisions but the text element itself.

A revision is always restricted to either a documents structure tree or a specific text element. Thus, it is not possible to edit multiple text elements in one revision. It is also not possible to edit a text element and the structure tree in one revision. This forces the user to make more atomic revisions, but it might also add potentially unnecessary noise in some cases.

5.3.2 Reasons for the Choosen Model

This model was chosen for the following reasons:

Due to the restrictiveness and atomicy of revisions, it is easilly possible to create revisions automatically. To prevent unnecessary pollution of the revision history, subsequent revisions by the same author to the same element are squashed within a set time frame.

Forcing the user to restrict changes to one text element while automatically creating new revisions frequently, thus keeping revisions as atomic as possible, lowers the possibility of a conflict.

Adding real-time collaborative features for text editing later does not require any additional separations of text buffers, since it is already present in the underlying model.

6 User Management

The system contains at least one Superadmin, who can:

- Create users
- Delete users
- Edit users
- Grant Superadmin status to users
- Assign users to groups and assign roles (Admin or Member)

6.1 User Creation

User creation is possible by a (Super) Admin.

When creating a user, an Admin enters a username, an email address and a temporary password. Afterward, the user can log in and has to change their

passsword.

The reasons that this is not done via a mail service is that at the time this was implemented, we were still missing a mail service and had no time to revisit this after.

Admins create users within one of their groups, while a Superadmin can also create users without assigning them to a group.

6.1.1 Users in Groups

An Admin has an overview of all users in their group and can:

- Create users within the group
- Grant and revoke Admin rights within the group
- Add existing users to the group
- Remove users from the group
- Create and delete new groups (Admins are authorized to do this, not just Superadmins)

6.2 Role System

Roles: `Member`, `Admin`, `Superadmin`

An `Admin` is a `Member` with special privileges. `Members` are assigned to `Groups`, while `Superadmins` operate globally. `Admins` always hold their Admin rights in relation to a specific group.

6.2.1 Example:

Username	Group 1	Group 2
User123	`Member`	`Admin`
Admin4	`Admin`	`Member`

6.3 Role Permissions

Admins only have rights (for example, promoting users) within the groups where they are Admins.

	Member	Admin	Superadmin
Create users	no	**yes**	**yes**
Edit users	no	**yes**	**yes**
Remove users	no	**yes**	**yes**
Delete users	no	no	**yes**
View full user data	no	no	**yes**
Promote to Member	no	**yes**	**yes**

	Member	Admin	Superadmin
Promote to Admin	no	**yes**	**yes**
Promote to Superadmin	no	no	**yes**
View user list	no	**yes**	**yes**
Create groups	no	**yes**	**yes**
Delete groups	no	**yes**	**yes**
Create documents	no	**yes**	**yes**
Delete documents	no	**yes**	**yes**
Change document rights	no	**yes**	**yes**

6.4 Password Reset System

To trigger a password reset, users can request a password reset over the website, which will trigger a request to `/password-reset/request`. This will trigger a mail containing a link with a one-time token to their mail, should that mail exist. The link can be used to reset their password, as would be expected of such functionality.

Note that for this to work, a valid mail configuration has to be provided.

7 REST API

The API makes the functionality from the previous chapters accessable to the frontend application. The public API of the backend is a REST API. Authentication is handled via JWT tokens. See Swagger for more detailed information.

8 Frontend Dev Documentation

Everything related to the frontend source code is located in the `/frontend/src/FPO` folder.

A simple introduction to the underlying web framework is found in the first 'chapter': - **Halogen**.

There are 6 subfolders in there for the different parts of our application. Those are

- **Data** - Core data types, routing, and application state management
- **Page** - Individual page components and their logic
- **Components** - Reusable/ important UI components
- **Translations** - Internationalization and language support
- **Dto** - Data Transfer Objects for API communication
- **UI** - UI utilities, styling, and design system components

8.1 Halogen

Halogen is the name of the 'web framework' we chose for this project. In halogen you define components for everything that should be shown in the resulting web page. Halogen components have the data type

```
component :: forall m. H.Component Query Input Output m
```

where `Query` is the data type for requesting information about the compoent from another component `Input` is the data that the components needs from other components and the `Output` type specifies what the component can output.

That is the base functionality. You don't have to use any of them and have a standalone component or use everything if the component is well connected to other ones.

Then there are a few key functions to handle the display of the component as well as the functionality. Nearly every creation of a component looks like this:

```
component =
  H.mkComponent
    { initialState
    , render
    , eval: H.mkEval H.defaultEval
        { handleAction = handleAction
        , handleQuery = handleQuery
        , initialize = Just Initialize
        , receive = Just <<< Receive
        , finalize = ...
      }
    }
where
initialState :: Input -> State
render :: forall m. State -> H.ComponentHTML Action () m
handleAction :: forall m. Action -> H.HalogenM State Action Slots Output m Unit
handleQuery :: forall a m. Query a -> H.HalogenM State Action Slots Output m (Maybe a)
```

Every component needs a `State`, that reflects the current information of a component. Like the method declarations indicate `initialState` is building a state from the input and render renders the UI of the component based on the current state. Those two functions are pure, that means we don't have access to the AppMonad and thus not on the application wide store, API requests and other things.

`receive` and `initialize` are there so that you can use monadic expressions when the component is created or the input changes. `finalize` is not used in our system, but can be used when a component is destroyed.

For the functionality and interaction of the components there are the other

functions mentioned in the `eval` field. `handleAction` handles actions triggered by elements inside of the component itself, while `handleQuery` handles actions triggered by other components that requested this information.

One not explored type until now is `Slots`. This type defines all the children components, that are embedded in this component. Like you will see later, out `Splitview` component contains many different children

```
type Slots =
  ( comment :: H.Slot Comment.Query Comment.Output Unit
  , commentOverview :: H.Slot CommentOverview.Query CommentOverview.Output Unit
  , editor :: H.Slot Editor.Query Editor.Output Int
  , preview :: H.Slot Preview.Query Preview.Output Unit
  , toc :: H.Slot TOC.Query TOC.Output Unit
  )
```

If there are open questions, look at the official documentation here: https://purescript-halogen.github.io/purescript-halogen/.

8.2 Base Functionality

8.2.1 Entrypoint

The frontend application has one entrypoint, that is the `Main.purs` file.

This file defines the root component that manages routing and page rendering based on URL changes.

The `component` function serves as the root Halogen component that renders different pages (Home, Editor, Login, Admin panels, Profile, etc.) based on the current route, while also maintaining persistent global components like the navbar and toast notifications.

The `main` function initializes the application by setting up the initial store with user preferences (language, translator), mounting the root component to the DOM, and establishing hash-based routing listeners to handle navigation between different pages of the application.

8.2.2 AppMonad

The AppM monad in `AppM.purs` is a newtype wrapper that combines Halogen's component monad with a Store monad to manage application state and provides navigation functionality by setting URL hashes when routes change.

8.3 Data

Here all core data types and other application wide data logic (error handling, data store, etc.) are placed in here.

8.3.1 Store

In `Store.purs` we defined a global state for the application is delivered by Halogen if you want to use it. At application startup the language is loaded and the resulting translator for the website is configured. While the application is running this store keeps track of upcoming errors while doing backend requests, updates the current route of the application and this updated the shown page simultaneously. You can see all those in the `type Store = { ... }` in the source file.

8.3.2 Route

The `Route.purs` file includes all the neccessary information and mapping urls to pages for routing in the application.

8.3.3 Request

The `Request.purs` file contains our own set of functions for backend API calling. Those functions all use `Affjax` in the background and are connected to the store of our application to catch and update errors in those functions directly, so that the business logic doesn't have to keep track of all errors. There are general functions like `getJson` that returns a JSON object from a get request or specified functions like `getUser` that also encode the result into the specified user data type.

8.4 Page

Individual page components and their logic. Every file in this folder correspond to one page in the application with a dedicated url behind it (e.g. Login, EditorPage, Profile, …).

8.4.1 Home

The `Home` Page is a bit different from the others, because you get a totally different view whether you are logged in or not. This is determined via the line

```
userWithError <- Store.preventErrorHandlingLocally getUser
```

and the following logic where the state of this component will get updated.

8.4.2 Page404

This is like the sink of the application and every request resulting in a `404 - Not Found` from the backend will automatically redirected here.

8.5 Components

Here all components for possibly reusable contents on web pages are stored. Most of the time one or more components are embedded into a page because of

the distinct features or applications although the component might not be used somewhere else.

8.5.1 Splitview

The `Splitview.purs` component is our most complicated one. It consists of the three splitted views (as the name suggests) from the editor page perspective. In there lies at the left side the TOC or the comment section, in the middle the code editor itself and on the right side the preview or a comparison view. This, the size of the screens and all the transitions of data are handled through this component.

8.5.2 Table Of Content (TOC)

The `TOC.purs` file contains all the information about the table of contents of the currently viewed project. It stores the currently viewed table of contents, all the meta information which configuration of paragraphs is allowed, synchronizes the TOC with the backend and handles all the document's and paragraph's history.

8.5.3 Editor

Everything related to the code editor lies in the `FPO.Components.Editor` package. For everything to be working right here, we need to have two extra JavaScript files that allow us to configure and use the external editor library how we want to. External configurations include editor highlighting based on different font styles or comments and keybindings.

8.6 Dto

We use DTOs (or **D**ata **T**ransfer **O**bjects) mostly for handling data transmission with the backend or between components. That is, because a API reponse might not give back all the information about a user but just a subset of information that is then translated into the specific DTO in the frontend.

8.7 Translations

To be able to have our website in two languages (and to easily add another one in the future if needed) we use the `Simple.I18n.Translation` package from PureScript. With that, instead of simply putting a hard-coded string everywhere on our pages, we have a translator in our `Store` and put lines like

```
(translate (label :: _ "comment_delete") state.translator
```

in our html pages. Via the navbar component you can update the language of the page instantly through this implementation.

This package has some interesting design decisions, for example that the complete list of all labels (defined in `Labels.purs`) has to contain all application

wide used labels lexicographically ordered in a single 'list'. We have a dedicated translation file for all the pages, so that adding or editing a is easier.

8.8 UI

The `FPO.UI` package contains various utlity functions for consistent design or functions that the otherwise used `Web.HTML` library does not provide. Some of them are also directly translated from their JavaScript counterparts, therefore you see two js files also in there.

9 FPO.Components.Splitview

Splitview is the main component that is used in EditorPage. It is the parent component and has multiple child components, split into three separate views. Depending on the task, the child could be displayed either on the left, middle, or right side of the split view. The Splitview component is thus the coordinator for all the children. It either redirects data from one component to another (e.g. selecting an entry in TOC should notify the editor component by sending the corresponding ID), only gets data from its children (e.g. closing request from the Comment component), or only sends data to a child (e.g. sending a newly received tree from the backend to TOC). The middle view only has the main editor and is always visible. The other views can each only harbor one child component at a time and can also be closed by the user.

9.0.1 Splitview has the following children:

- Left view:
  - TOC (Table of contents)
  - Comment
  - CommentOverview
- Middle view:
  - Editor (0 is main editor)
- Right view:
  - Editor (1 is compare editor)
  - Preview

9.1 Important Interactions

9.1.1 Request

Splitview is the component responsible for handling the requests regarding the document tree from and to the backend. In a GET request, the tree is stored in the state as `tocEntries` and then sent to TOC. The POST is only called by TOC to update the new structure of the tree to the backend.

9.1.1.1 Request uses the following Actions:

- GET
- POST

9.1.1.2 Request uses the following state labels:

- tocEntries
- versionMapping

9.1.2 Resize

The Splitview gives the user the ability to resize the size of the three split views. It uses a DOM element referred to here as the resizer. The resizer visualizes the border of the middle view with either one of the other views. With mouse events, the user can drag and drop the resizer to resize the sizes, which are calculated in percentage. While dragging, the views cannot be smaller than a set size, which is 5%. While the middle and right views' max size is dependent on the min sizes of the other views, the left view's max size is capped at 20%.

9.1.2.1 Resize uses the following Actions:

- StartResize DragTarget MouseEvent
- StopResize MouseEvent
- HandleMouseMove MouseEvent

9.1.2.2 Resize uses the following state labels:

- mDragTarget :: Maybe DragTarget
- startMouseRatio :: Number
- startSidebarRatio :: Number
- startPreviewRatio :: Number
- sidebarRatio :: Number
- previewRatio :: Number

9.1.3 Toggle visibility

Since all other components of EditorPage are in Splitview as children, toggling their visibility is thus also a responsibility of Splitview. As the middle main editor view is always visible, there does not exist the ability to close this component. The other two views can be closed by the respective button on their resizer to close and reopen themselves. The right view additionally has a close button on its top right to close itself. However, each view can only have one component active at a time. The content of the right view can be changed depending on different button presses. The left view, however, can have "overlapping" components, which can be closed with a top-right close button. The order from bottom to top is the following: TOC, CommentOverview, and then Comment. The components are made invisible, but they are not deleted, so as

not to delete their state and reload the information again if we open them up again.

9.1.3.1 Toggle visibility uses the following Actions:

- ToggleComment
- ToggleCommentOverview Boolean Int Int
- ToggleSidebar
- TogglePreview

9.1.3.2 Toggle visibility uses the following state labels:

- sidebarRatio :: Number
- previewRatio :: Number
- lastExpandedSidebarRatio :: Number
- lastExpandedPreviewRatio :: Number
- sidebarShown :: Boolean
- tocShown :: Boolean
- commentOverviewShown :: Boolean
- commentShown :: Boolean
- previewShown :: Boolean

10 FPO.Components.Editor

There are two types of editor. The editor, to which the splitview can access with id 0, is the main editor in which the user can edit and work inside of it. The other editor, which the parent can access with id 1, is used to compare older versions of the document with the main editor. It is, however, set to read-only and its content cannot be changed. Since this component has a lot of features and is quite long, many data types and functions are relocated into FPO.Component.Editor.Types.

10.1 Main editor

At the beginning, since no section has been selected, no content is loaded and the editor is set to read-only. When the TOC component selects an entry, it is then redirected to this editor and disables the read-only state. Then it requests the data from the backend with the help of ContentDto. This data contains the content, comment anchors, which are markers for the position of comments, and HTML, which is then sent to the preview component. The main editor has a lot of features, which are further explained later on.

10.2 Important Interactions

10.2.1 Toolbar

TODO: Write the toolbar section.

10.2.2 Request

This component directly loads the content for the selected entry directly from the backend and does not communicate with its parent regarding this issue. The data is a wrapper defined in ContentDto, which contains the content, comments, and rendered preview in the form of HTML. The content is stored in its state and the HTML string is sent to the preview component. The comment process is detailed at a different point. To post data, the editor wraps only the content and comments into one data type, since HTML is only created in the backend.

10.2.2.1 Request uses the following Actions:

- ChangeToSection TOCEntry (Maybe Int) (Maybe String)
- ContinueChangeToSection (Array FirstComment)
- Upload TOCEntry ContentWrapper Boolean

10.2.2.2 Request uses the following state labels:

- mContent :: Maybe Content
- html :: String
- markers :: Array AnnotatedMarker

10.3 Comment

The comment aspect is quite complex, as it has multiple features related to this. It thus has its own state called CommentState, which stores all relevant information in it. The editor loads the information about the comments in two different locations. It gets the positions of its anchors in this component and the extra information from the comment component. Therefore, when TOC requests an entry change, the corresponding Action is divided into two parts. The first part is to get data from the backend and send an information request to the comment component. After receiving it, it then processes it in the second part.

10.3.1 Comment position representation

There exist two arrays that hold different representations of the positions of the comments. The **live markers** are the temporary representation, which the editor actually uses to work with. The positions of the comments are constantly updated by mainly two interactions:

- **Anchor:** For each comment, there exists a start and end anchor, which move according to the content changes made by the user inside the editor. Each comment is then given a custom CSS class to highlight this range in the editor.

- **Handler:** At most, only one comment can be selected at a time. When selected, each end now has a handle, which the user can use to change its range through mouse drag and drop.

The other data type is called **AnnotatedMarker**. It is an intermediate representation of the live markers. It converts the live markers to JSON or the other way around.

10.3.2 Modifying the range of a comment

As previously mentioned, we have two ways of changing the positions of the ends of a comment. The anchor updates with every text change, but the corresponding CSS class does not. With every Anchor change notification, we replace the old marker with the updated one. The other method is more complicated and uses a lot of Actions and state labels.

10.3.2.1 Handles Handles are a CSS class that the user can click on and drag. These are, however, only for visualization and can be turned on and off by the two functions **showHandlesFor** and **hideHandlesFrom**. Only selected comments can have those handles. The interactions of actually moving the handles are realized by the three listeners, which react to the events **mousedown**, **mousemove**, and **mouseup**. We use four Actions for the interaction:

- **TryStartDrag:** First check if the clicked position is near the selected comment's start or end handle. If this is true, then put the corresponding **DragHandle**, **mPrevHandler** for auto-save, and **mHandleBorder** for range detection in the commentState and go into Action StartDrag.

- **StartDrag:** Only able to drag comments if we are the main editor, which can be checked if there is **no compareToElement**. We add the **fpo-dragging** CSS class to prevent the cursor from highlighting the selected text. Then refresh the handles before going to the next Action.

- **DragMove:** We use an FFI to convert the mouse position to coordinates in the editor. While dragging, it prevents the new position from being beyond the other handles. We store **dragRowAS** and **dragColAS**, with AS for auto-save, to check if the comment range has changed since the last save.

- **EndDrag:** First, set the **DragState** to Nothing to stop the dragging. Remove the **fpo-dragging** CSS class to allow selecting text highlighting again and remove the selection from the user. Then prepare for potential auto-save.

It has additional **ShowHandles** and **HideHandles** Actions, which are mainly used for the corresponding **showHandlesFor** and **hideHandlesFrom** functions. They are primarily used for easy access by other Actions in this compo-

nent.

10.3.2.2 Handles use the following Actions:

- TryStartDrag Number Number – clientX, clientY
- StartDrag DragHandle LiveMarker Number Number – mouse down: which, lm, clientX, clientY
- DragMove Number Number – mouse move: clientX, clientY
- EndDrag – mouse up

10.3.2.3 Handles use the following commentState labels:

- dragState :: Maybe {‘{’} which :: DragHandle, lm :: LiveMarker {‘}’}
- startHandleMarkerId :: Maybe Int
- endHandleMarkerId :: Maybe Int
- mPrevHandler :: Maybe Types.Position
- dragRowAS :: Int
- dragColAS :: Int
- mHandleBorder :: Maybe HandleBorder

10.3.3 Annotated Marker sequence

After loading the comment from ContentWrapper in the GET request, it is then converted to an AnnotatedMarker and stored in markers. Since we need more information from the comments, we request the first message of the comment from the comment container at the end of the first Action. In the continuation Action, we filter the acquired data to have only non-resolved comments. Then, with the help of the **addAnchor** function, the annotated marker is converted to live markers, added to the editor, and stored in the state.

10.3.4 Annotation

On the gutter on the left side of the editor, it shows the comment annotation. It shows where the comments start in the editor. When hovering over the icon, it also shows the name of the author who wrote the first message in the comment conversation. If there are multiple comments starting on the same line, all different users are listed in the annotation. Same users on the line are compressed in the format: user+int. For the latter feature, we use two hashmaps.

- **markerAnnoHS:** Takes the line of the editor as its key and returns as value another hashmap. The returned hashmap represents the occurrence of the user.

- **oldMarkerAnnoPos:** Used to find out on which line the comment lies. It uses the ID of the marker as the key, and the returned row number can then be used in markerAnnoHS.

10.3.5 Creating Comment

The user first selects a text and then clicks on the Comment button. This triggers the **Comment** Action. There it gets the data **user** to use its name later. It converts the selected text into a range, from which the start and end positions are extracted. From this data, it creates a new **AnnotatedMarker**. With the help of **addAnchor**, it converts it into a **liveMarker**. This marker is then stored as the current selected **liveMarker** and as a temporary liveMarker **tmpLiveMarker**. Temporary because we do not want to send it to the backend until the first message is sent in its conversation. When a **tmpLiveMarker** already existed prior to clicking the Comment button, it just overwrites **tmpLiveMarker** with the new one. After saving the new marker in the state, it then sends a notification **AddComment** towards the comment component. It then shows the handles of the **tmpLiveMarker**, as it is selected, and clears the marked selection. When the first message has been sent, this editor receives a query **UpdateComment**.

10.3.5.1 Creating Comment uses the following Actions/Output/Query:

- Comment
- AddComment
- UpdateComment

10.3.5.2 Creating Comment uses the following state labels:

- markerAnnoHS
- oldMarkerAnnoPos
- tmpLiveMarker
- selectedLiveMarker
- markers
- liveMarkers

10.4 Saving

There are two ways to save in the editor: **manual saving** and **auto-saving**. Both of them have the same sequence of sending the **content** and the **positions of the comments** in this section to the backend. The only difference is that manual saving is done with the save button and other interactions, while auto-save is done with a timer. It also sends an **autoSave boolean** value to the backend, which is used for compare editor. It can only be sent if the mutable value reference **mDirtyRef** is set to true. This action occurs in the added listener with the function **addChangeListenerWithRef**, which sets multiple flags to true and starts a timer for auto-save. Since saving uses a lot of state labels, it has a separate type called **SaveState**.

10.4.1 Saving

The user is able to save the changes with the help of the save button. It only works if the mutable value reference **mDirtyRef** is set to true beforehand. There we first go into the **Save** Action, where it first checks if it is the **main editor** or not. Only the main editor has the ability to save the changes. It then checks the dirty flag. Only if the previous checks have turned to true does it start to collect all the necessary data before uploading it. It first extracts the text from the editor and updates the **content** value, stored in **SaveState**. Afterwards, it then updates the positions of the **array of AnnotatedMarker called markers** from its corresponding **array of LiveMarker called liveMarkers**, with each **LiveMarker** having the same ID as its matching **AnnotatedMarker**. With both pieces of data, it then puts them into a wrapper defined in **ContentDto**. It then goes into the next Action **Upload**. (With this separation, we can call **Upload** again in case of a failed upload attempt, without extracting the data from the editor again. This is, however, deprecated.) With the data, it converts them into JSON and sends it to the backend. When an error is sent back, it displays it in a toast.
TODO: other cases
At the end, it sends the rendered HTML to the preview component and updates the section title displayed above the editor, which is requested from the TOC component.

10.4.1.1 Saving uses the following Actions/Output:

- Save
- Upload
- SavedIcon
- RequestFullTitle

10.4.1.2 Saving uses the following state labels:

- markers
- liveMarkers
- mDirtyRef
- showSavedIcon
- mSavedIconF

10.4.2 Auto Save

With every change in the text, the **dirty flag** is set to true and goes into the **AutoSaveTimer** Action. There, two timers are started with the help of two fibers: a 2-second and a 20-second timer. The shorter timer is reset every time the user makes another change. The longer one is used to save the changes regardless of whether the user has made some changes or not. When either of the timers has run out, it then goes into the Action **AutoSave**. There, it kills

both fibers to stop the timers and goes into the **Save** Action, which is already described above.

10.4.2.1 Auto Save uses the following Actions:

- AutoSaveTimer
- AutoSave
- Save

10.4.2.2 Auto Save uses the following state labels:

- mDirtyRef
- mPendingDebounceF
- mPendingMaxWaitF

10.4.3 The following actions triggers save:

- Save button
- Background timer of 20 seconds or 2 seconds after latest content change
- Change to a diffrent Leaf
- Closing tab
- Send first message in a new comment
- Change border of a comment
- TODO: Older versions of section.

11 FPO.Components.Comment

The comment module is used to showcase the **conversation of a comment**. This comment is selected in the **editor** module and showcased on the left side in the **splitview**.

11.0.1 Showing Comment

A comment is a conversation where users can send more messages to the previous ones. It follows the logic where the most recent message is put at the end of this conversation. There are two types of comments: the **first comment** and the later ones. Whenever the user creates a new comment in the **editor** module, there do not exist any messages yet. The first message thus has, beside the **send button** to send a message, a **delete button**. When clicking the delete button, a prompt appears to ask the user if they really wish to delete this comment. After sending the first message, this one is displayed bigger and in a brighter and more yellowish tone than the later ones. Then the future messages have, instead of the **delete**, a **resolve button** to "close" this conversation. After resolving, the first message now has a green background and the user cannot write more messages.

11.0.1.1 Showing Comment uses the following Actions:

- SendComment
- ResolveComment
- DeleteComment
- RequestModal Mode
- CancelModal

11.0.1.2 Showing Comment uses the following state labels:

- docID :: Int
- tocID :: Int
- markerID :: Int
- commentSections :: Array CommentSection
- mCommentSection :: Maybe CommentSection
- newComment :: Boolean
- commentDraft :: String

11.0.2 Draft and Sending Comment

When writing a new message for the conversation, its draft is then stored in its state. It only allows sending a message if its content is not empty. Depending on whether this draft belongs to a first message or not, it uses a different API.

11.0.2.1 Draft and Sending Comment uses the following Actions:

- UpdateDraft String
- SendComment

11.0.2.2 Draft and Sending Comment uses the following state labels:

- commentDraft :: String

12 FPO.Components.CommentOverview

This component shows all comments sent to this text section. This also include **resolved comments**, which are not visible in editor. Instead of showing all open conversations of all comments, it only shows the **first message** of each comments. When clicking on a comment, it then opens the comment module in splitview with the selected comment.

13 FPO.Components.Preview

This component is on the right side of the splitview module. It receives the rendered HTML text from the editor and simply displays it here. With the state label **isDragging**, it then activates or deactivates the iframe of this module to make it possible for the splitview module to resize this window.

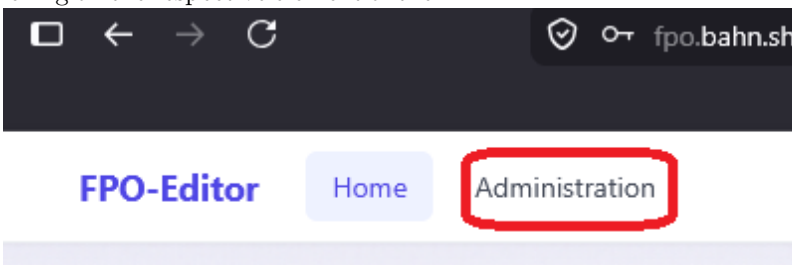
14 User Documentation

This section provides an overview of how to work with projects within the application. It explains the structure and functionality of the project interface, with a particular focus on the Table of Content (TOC), section management, and version handling. Users will learn how to navigate through a project, add or remove content sections, and access or edit previous versions of their work. The goal of this chapter is to enable efficient and structured project work while maintaining traceability and control over changes.

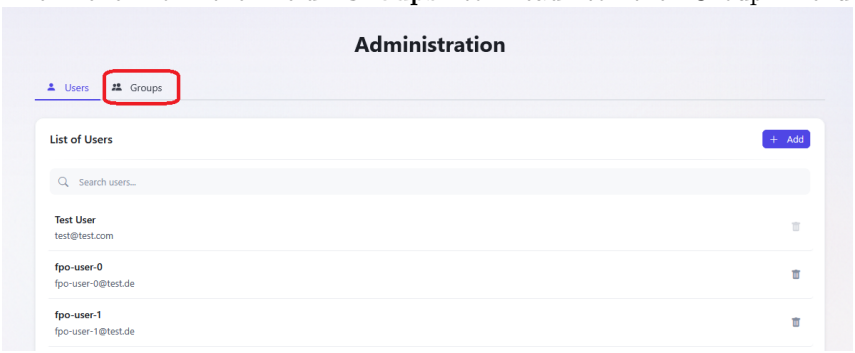
15 Group Management

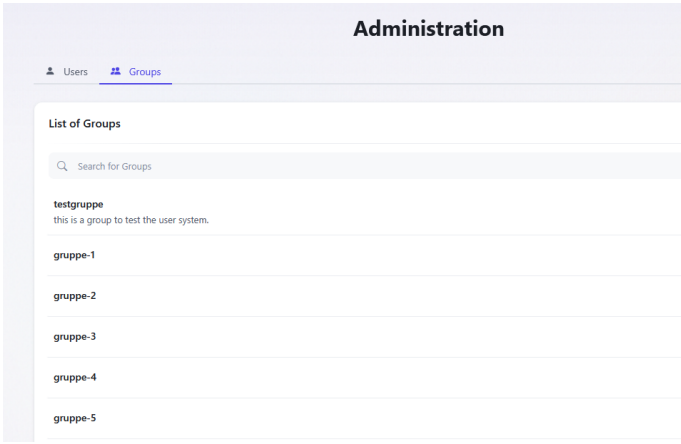
15.1 Creating a new Group

1. Go to the Administration menu by clicking on the respective element of the navigation bar in the top left corner.

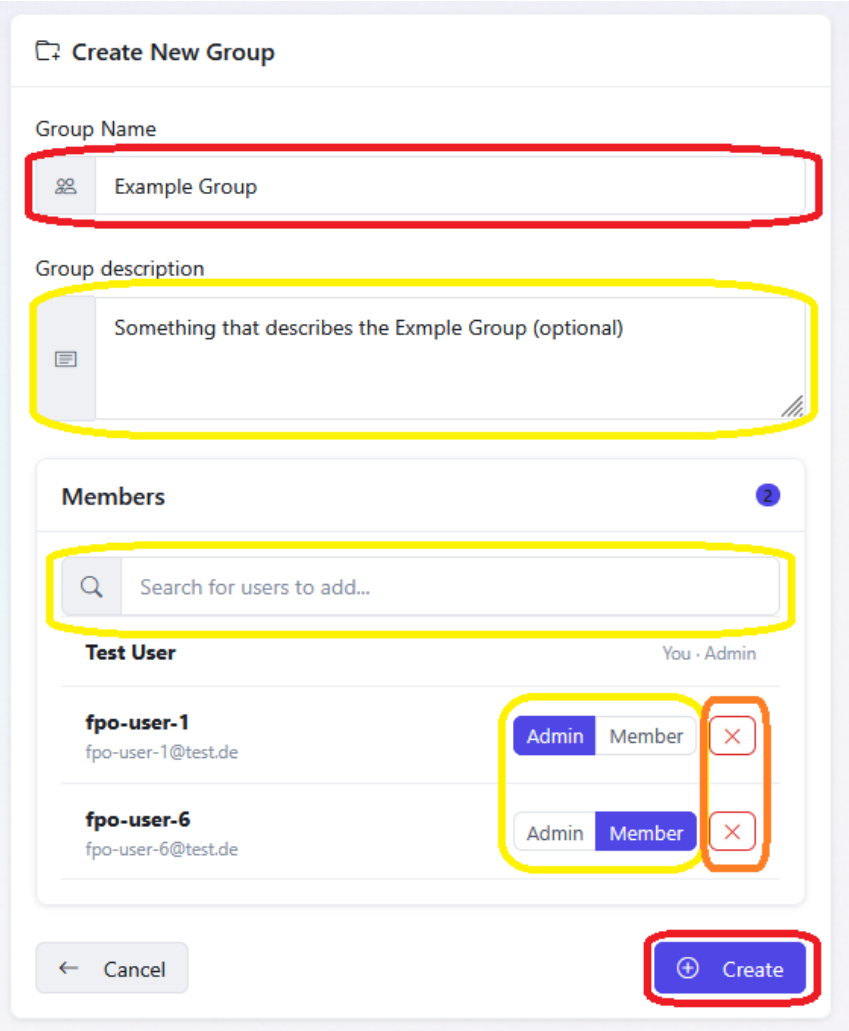


2. Then click on the field **Groups** to head to the Group menu.





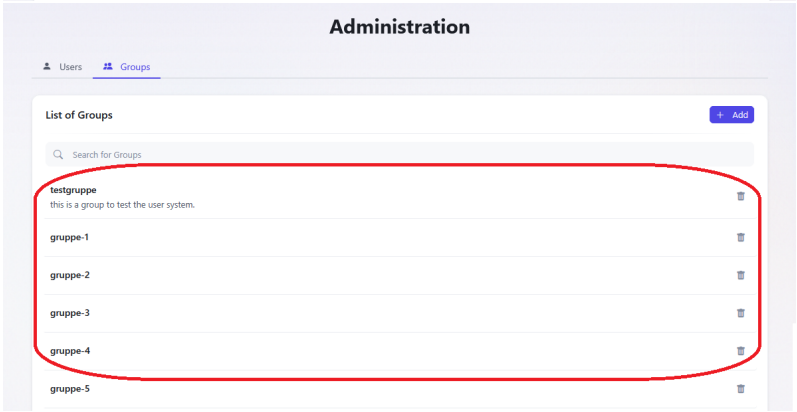
3. Click on the blue **Add** button in the top right.
4. A new window will appear where you have to name the group and confirm the creation by clicking **Create**. Before comfirming with **Create** You may also optionally:
    1. Enter a group description for your group.
    2. Add members to the new group by using the searchbar below **Members** and the drop-down menu of that bar. (You are automatically part of the group).
    3. The roles of the added members can be administrated with the **Admin**/**Member** toggle button on the right.
    4. If you added the the wrong member, you can remove them again with the cross on the very right of the menu.

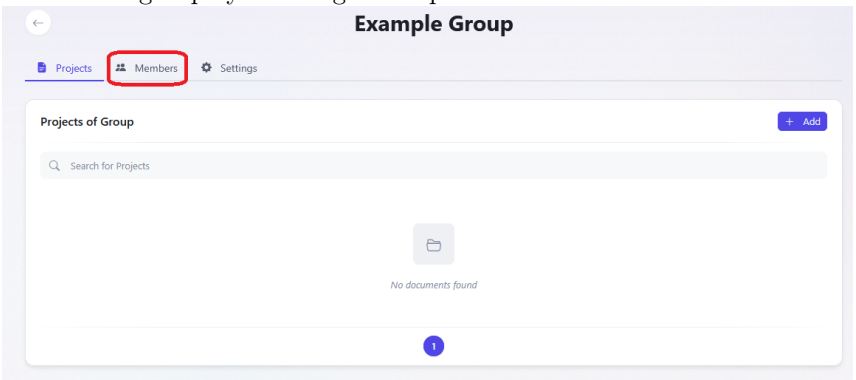


15.2 Member Management

1. To manage the members of a group, first access the respective group by heading to the **Groups** menu as described in **1.** and **2.** in Creating a new Group and choosing it from the **List of Groups**. If needed the groups can be filtered by name with the integrated search bar.

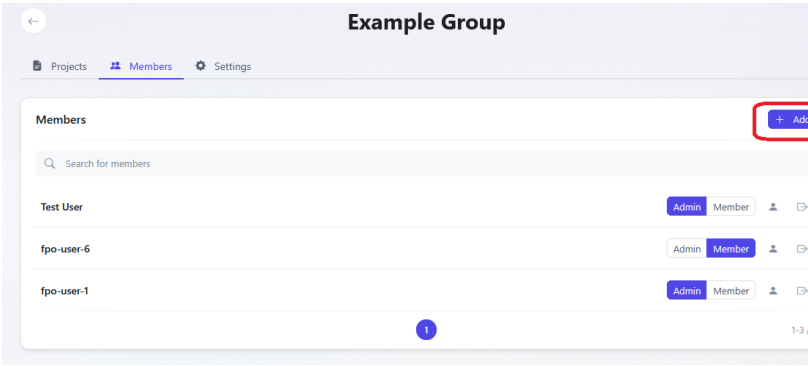
1. 

2. 

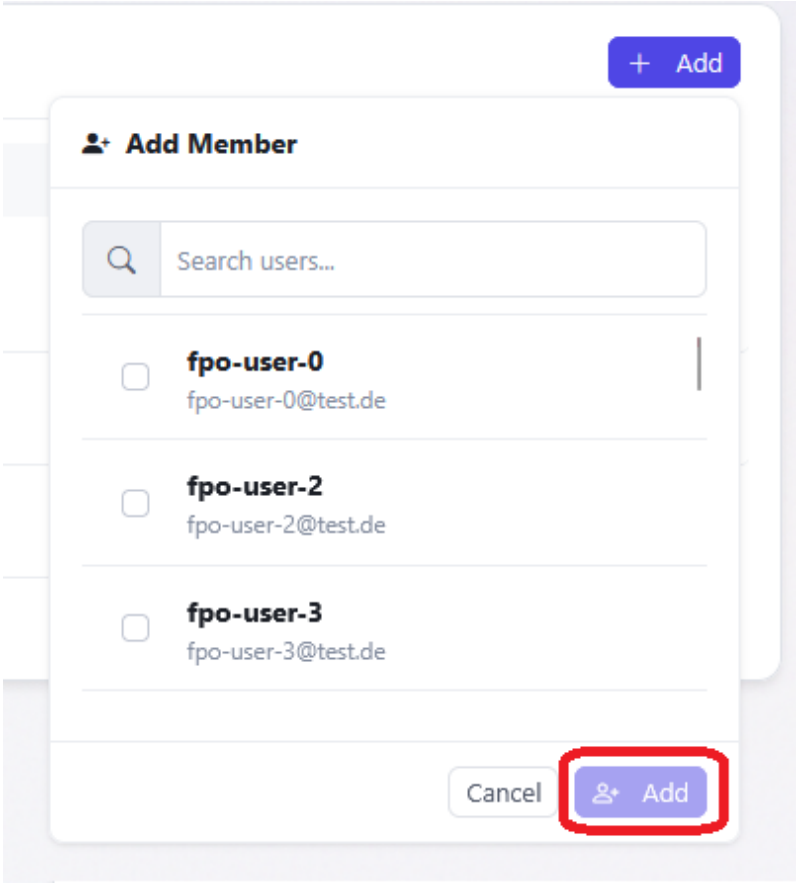
2. Go to the members section of the group by selecting the respective element on the left hand side. 

15.2.1 Add new Member

1. Go to the **Members** Page as described above.

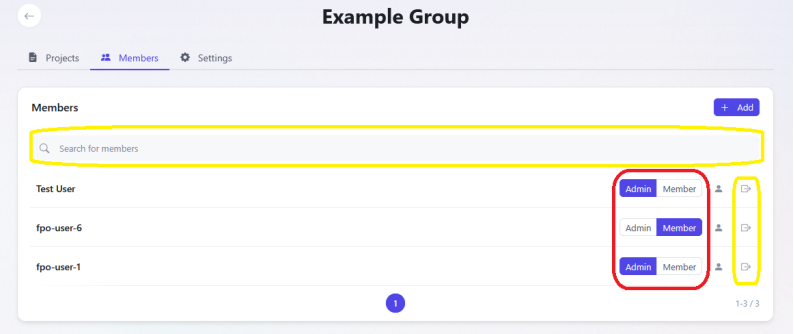


2. Select **Add** in the top right corner.
3. Any Member can now be added by clicking the checkbox on the left of the dropdown menu and confirmed by presseing **Add** in the bottom right corner.
    - Members can be searched by using the searchbar on the top. You can search by name or email.



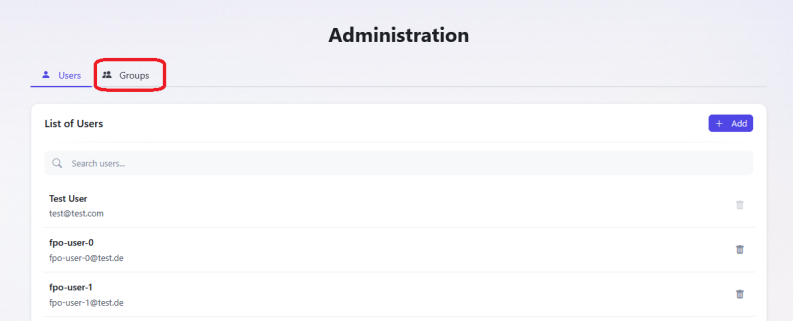
15.2.2 Manage Roles

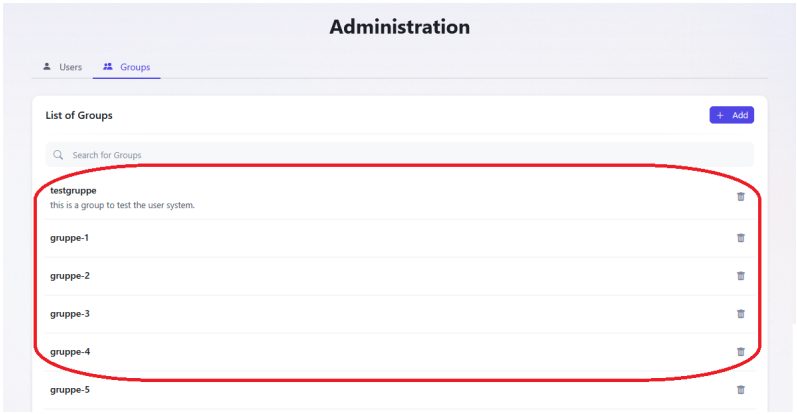
1. Go to the **Members** Page as described above.
2. To change the role of a member select the respective role of the member with the toggle button.
    1. Members can be searched with the search bar above the list.
    2. Members can also be removed in this menu with the red door button on the right of the dropdown menu.



15.3 Project Management

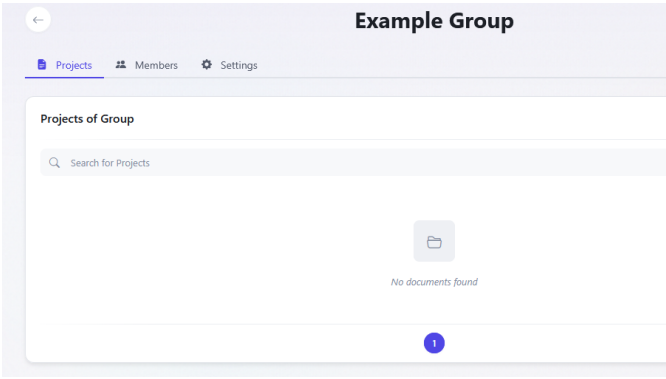
1. To manage the projects of a group, first access the respective group by clicking on **groups** and choosing it from the **List of Groups**. If needed the groups can be filtered by name with the integrated search bar.



2. 

15.3.1 Project Creation

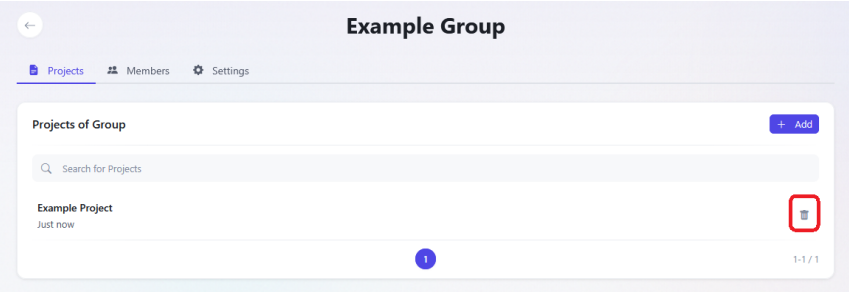
1. First go to the **project management** interface as described above.



2. Click on **Create Project** on the left hand side.
3. Enter the name of the project you want to create and confirm it with **Create**. All members of the group will have automatically access to this project and can edit it.

15.3.2 Project Deletion

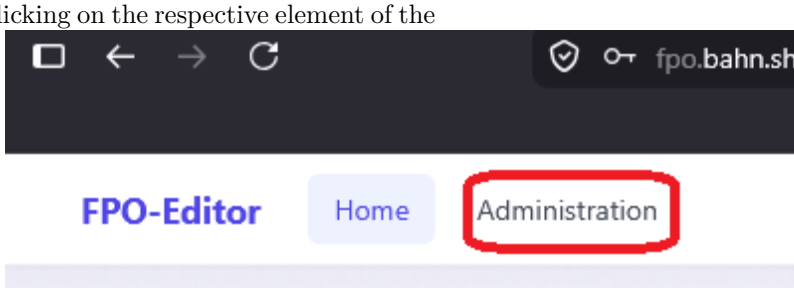
1. First go to the **project management** interface as described above.
2. You can delete project by clicking on the trash bin icon in of the projects in the project list and confirming with **Delete** in the pop-up.

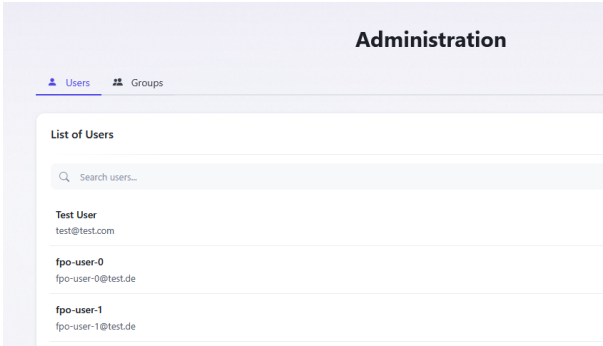


16 User Management

16.1 User Creation

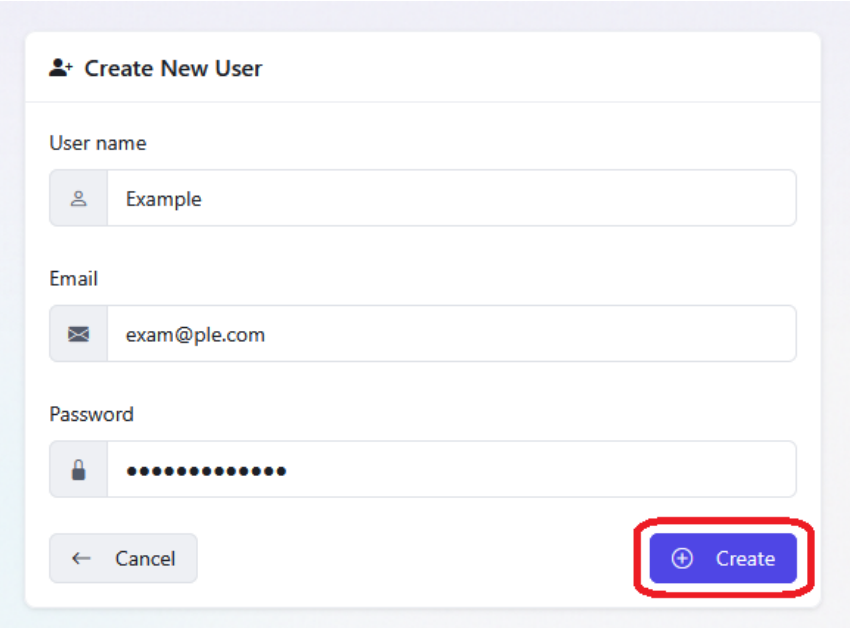
1. Go to the Administration menu by clicking on the respective element of the navigation bar in the top left corner.





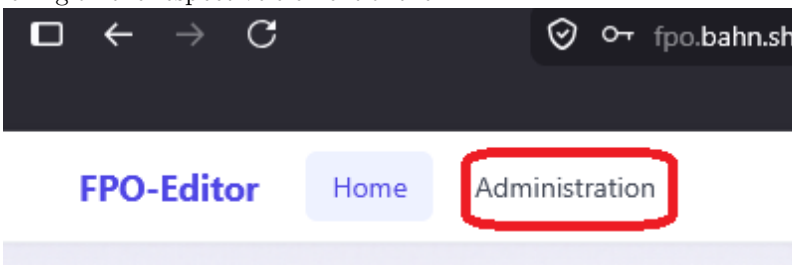
2. Click on the blue **Add** button in the top right corner.
3. Enter the user name, email, and a provisional password of the respective user into the fields in the **Create New User** menu and confirm it by click-

ing on **Create**

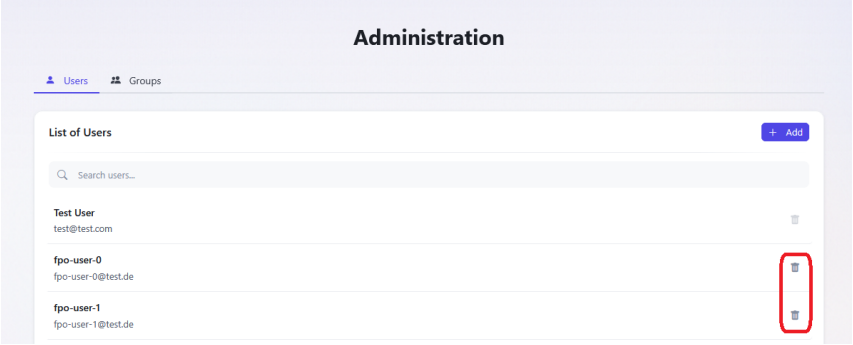


16.2 User Deletion

1. Go to the Administration menu by clicking on the respective element of the navigation bar in the top left corner.



2. Click on the red trash bin symbol on the right of the **List of Users** of the the respective user. You can search for a particular user with the **Filter by** window on the left, with the user name and/or the email.



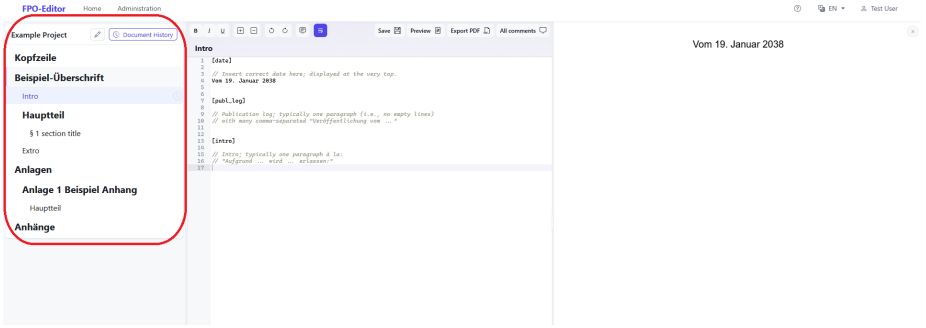
3. Confirm the deletion in the pop-up, with the red **Delete** button.

17 Working on a project

This section describes the core workflows for creating, structuring, and maintaining a project. It introduces the project interface and explains how content is organized, edited, and versioned over time. The following chapters guide users through navigating the Table of Content (TOC), managing sections, and working with past versions, ensuring a structured and transparent editing process throughout the project lifecycle.

18 Table of Content (TOC)

After creating a new project, you can access it by the home screen or the group interface. The left hand side displays the **Table of Content (TOC)** you can switch to a section by selecting the relevant point of content.

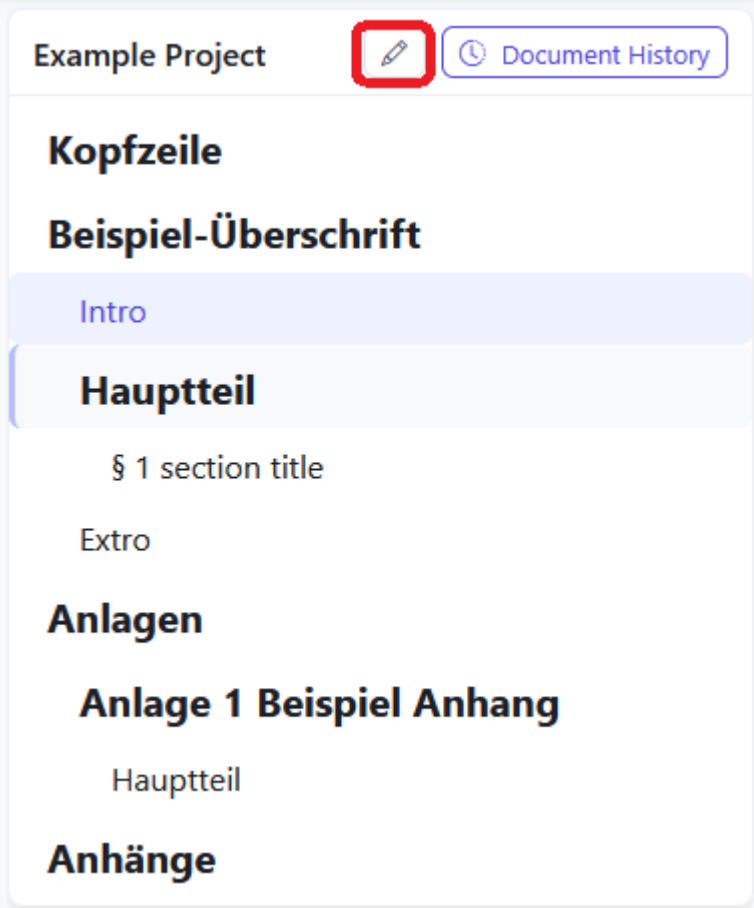


18.1 Parts of the TOC

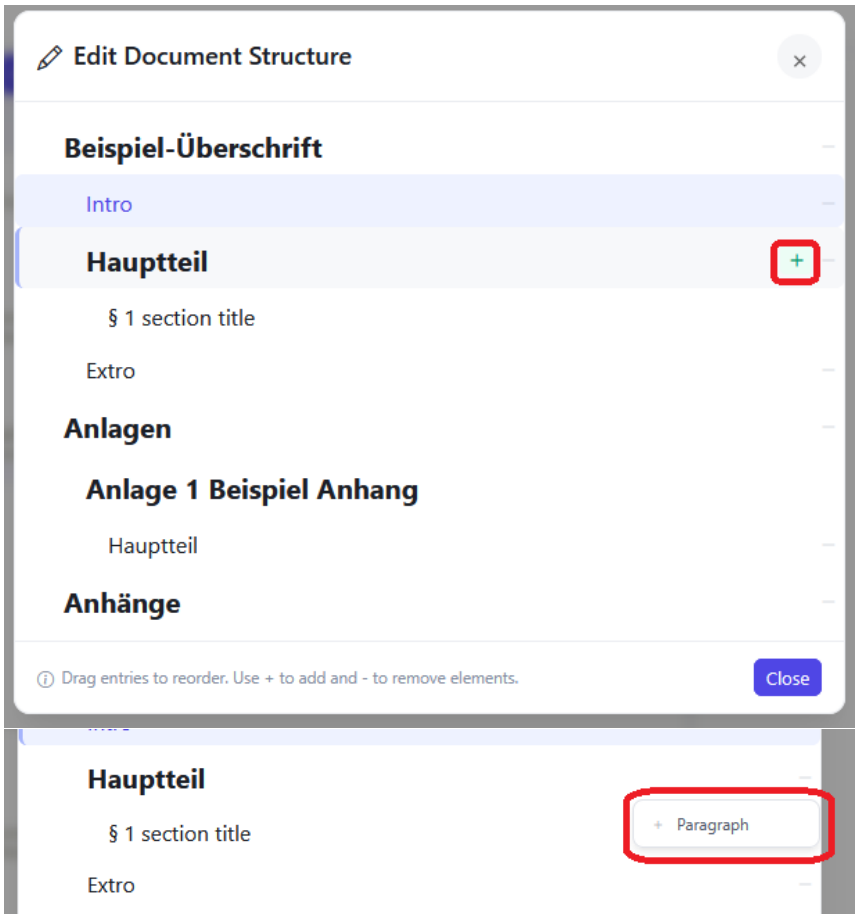
part	function
Kopfzeile	In the Kopfzeile, headers for the pdf generation can be set
Heading	The heading is set after the ! in this file
Intro	The Intro gives space for the date of publication, a publication log and a short introduction.
Hauptteil	Here the main part of the content can be written
Extro	This gives space for the date and name of the signatory, with space for the actual signature, as well as for the legal declaration.
Anlagen	Here goes all text regarding the appendix section. Tables can be inserted here.
Anhänge	This is the space for all necessary attachments

18.2 Adding a new section

1. Open the structure editor with the pen symbol in the top of the TOC

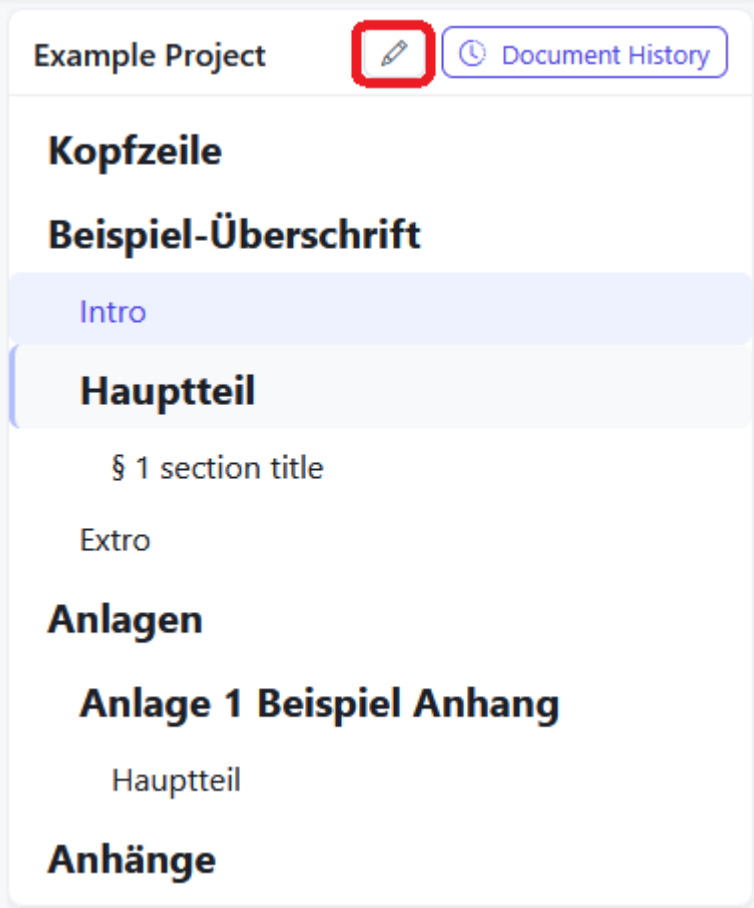


2. You can add new sections of **Hauptteil**, **Anlagen** and **Anhänge**, by hovering over the respective heading in the structure editor and clicking on the plus that appears. And confirming the popup window. It does not matter how many section are added but new sections are always created by going to the respective heading and new last section will be created.


Edit Document Structure
Beispiel-Überschrift
Intro
Hauptteil
§ 1 section title
Extro
Anlagen
Anlage 1 Beispiel Anhang
Hauptteil
Anhänge
Drag entries to reorder. Use + to add and - to remove elements.
Close
Hauptteil
§ 1 section title
Paragraph
Extro

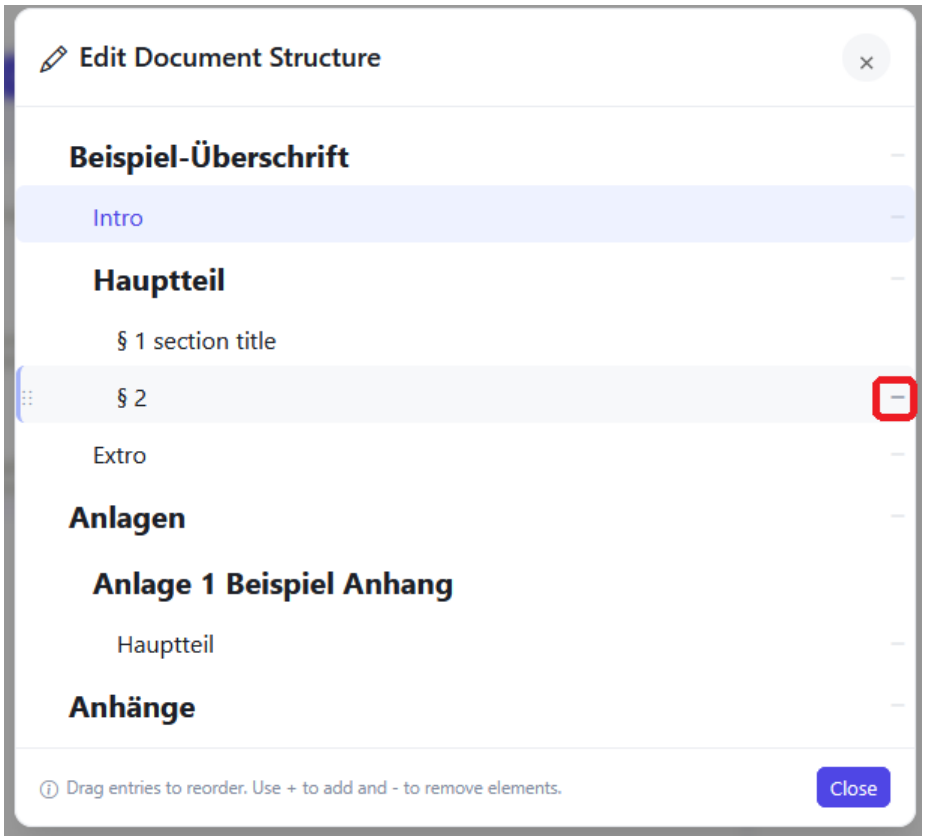
18.3 Deleting a section

Open the structure editor with the pen symbol in the top of the TOC



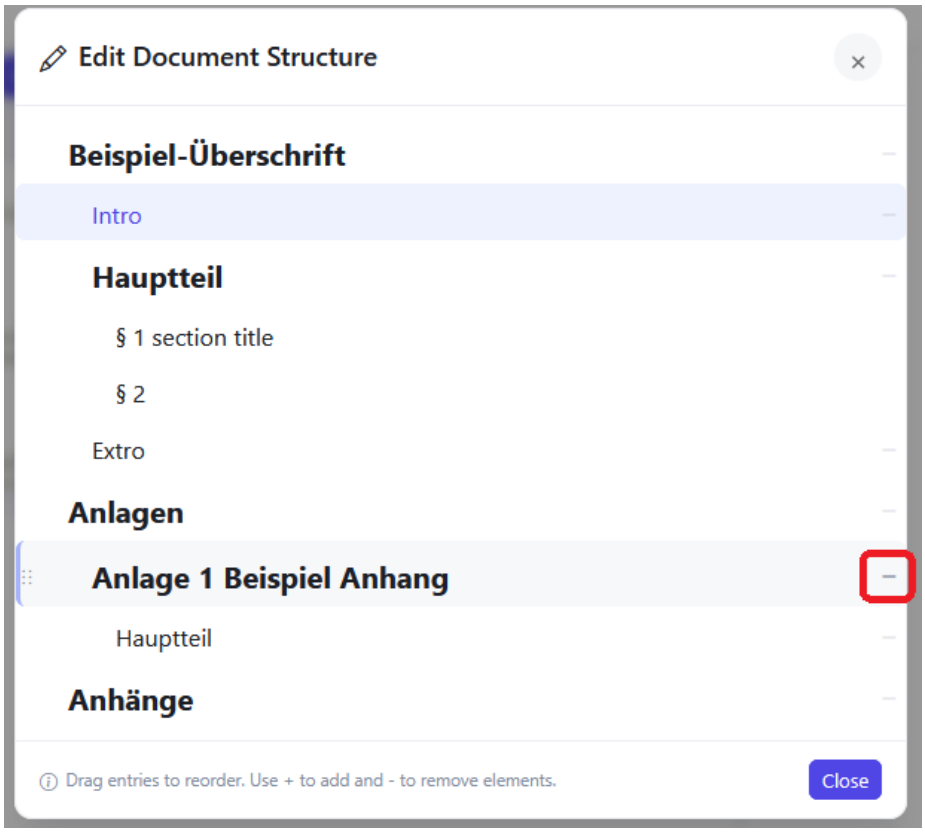
In the Hauptteil

You can delete a section by hovering over the respective section in the structure editor, clicking on the minus that appears and confirming the pop-up with

Delete. 

18.3.1 In the Anlagen/Anhänge

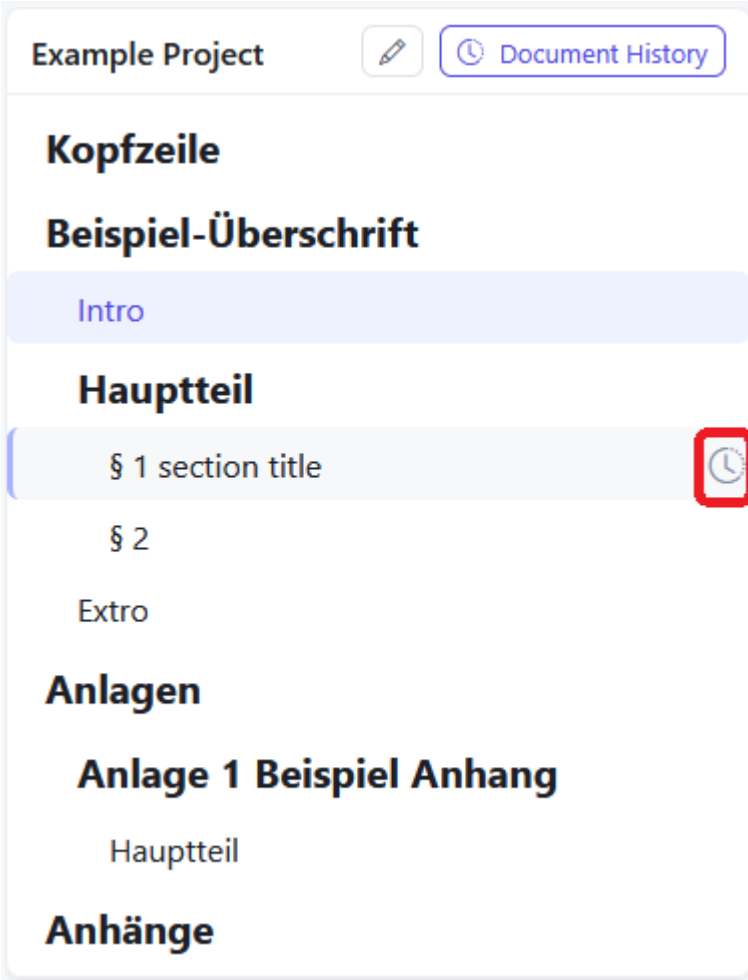
For the Anlagen and Anhänge you do not hover over the Hauptteil of the section in question but over the heading of the relevant Anlage/Anhang, again in the structure editor. You also have to confirm with the **Delete** button in the pop-up.


Edit Document Structure
Beispiel-Überschrift
Intro
Hauptteil
§ 1 section title
§ 2
Extro
Anlagen
Anlage 1 Beispiel Anhang
Hauptteil
Anhänge
Drag entries to reorder. Use + to add and - to remove elements.
Close

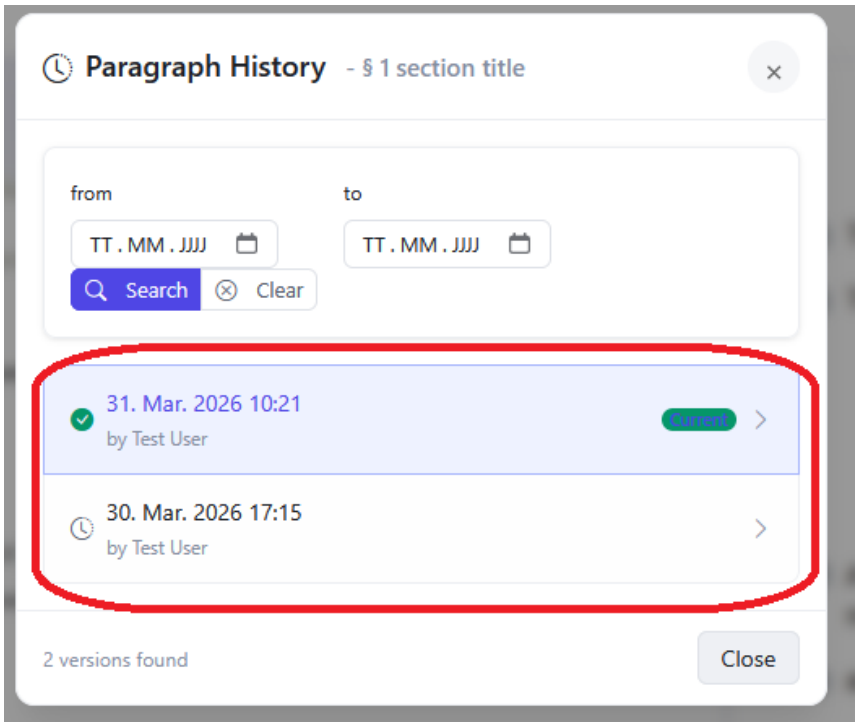
19 Working with past versions

19.1 Accessing a past version

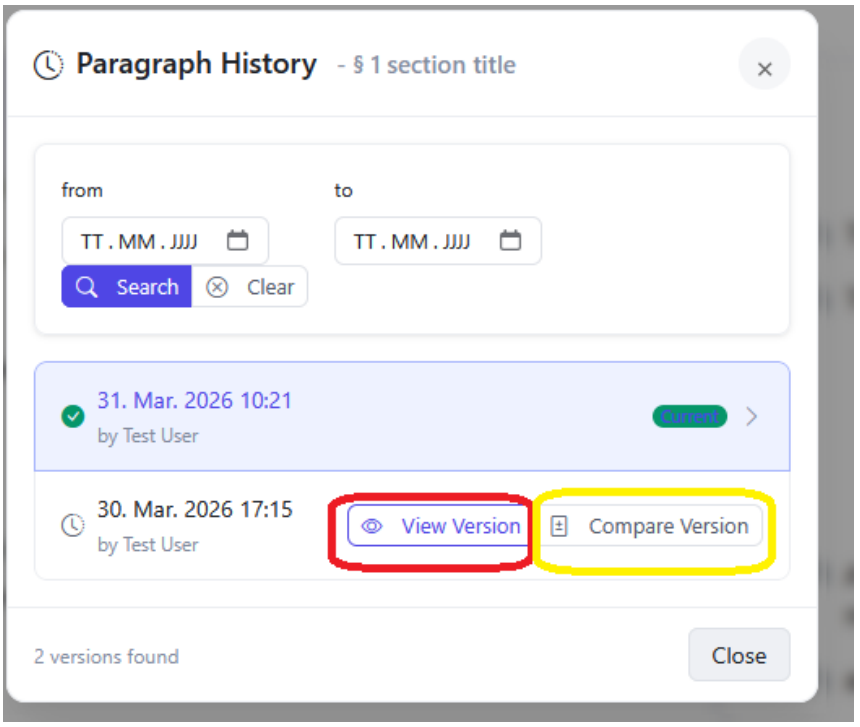
1. Hover in the TOC over the section you want to see the prior version of and



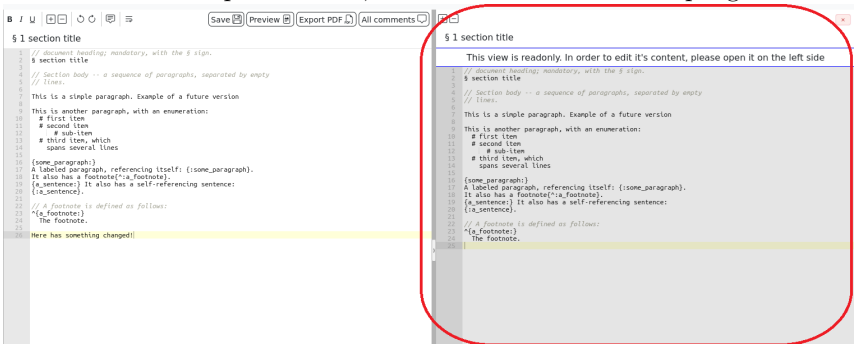
   click on the the appearing clock symbol.
2. Choose the version by clicking on one of the offered options. You can also filter the version by date with the top part of the pop-up.



3. Confirm your choice by clicking on **View Version**. Then the past version of the section will open in the editor. You can return by clicking **discard** in



the top banner of the editor. Alternatively you can choose **Compare Version**. Then the source code of the past version will open on the right and and the current version will stay in the editor. To close the past version, click on the cross in the top right corner.



19.2 Editing the past version

1. First open the past version in the editor as described above.
2. Make the desired changes and save the file.
3. Save the file by clicking on **Save** or CTRL+S
4. The Merge menu now opens. Add every change of the old new version you want to keep from the right side to the editor on the left.
5. Confirm by clicking **Merge**. Everything you have not added to the left,

will not be in the new new version.

20 Language

We define the Legal Text Markup language (LTML), which is actually a family of languages, where each language is defined by an LTML Schema Definition (LSD), specified in the eponymous language (compare XML and XSD).

20.1 Formal grammar specification

To specify formal grammars, we generally use a variant of EBNF.

20.2 Haskell code structure

- `Language/`
  - `Lsd/`
    - `AST.Type[.*]`: LTML types and node formats.
    - `Example[.Fpo]`: Example LSD; as of now the only available LSD, hardcoded as such. To be demoted to proper example once LSD can be parsed.
  - `Ltml/`
    - `AST.*`: LTML AST
    - `Parser[.*]`: LTML Parser
    - `Tree`: simple textual tree where the structure mirrors the navigation TOC in the frontend and the leafs are editable in the text editor in the frontend. This simple tree structure is strongly related to the tree structure of a document in the database.
    - `Tree/`
      - `ToMeta`: Building of metadata for use in the frontend.
      - `ToLtml`: Main translation function.
      - `Parser[.*]`: "Parsing" simple trees to LTML.
      - `Example.Fpo`: Simple example tree. Used as default for new documents.
    - `HTML[.*]`: Conversion to HTML.
    - `ToLaTex.*`: Conversion to LaTeX; to be further translated to PDF.

21 EBNF syntax

For specifying languages by formal grammar, we use a variant of EBNF (Extended Backus-Naur form).

21.1 Basic syntax

As basis, we use the W3C/XML EBNF.

21.2 Extensions

21.2.1 Indentation

We permit special `INDENT` and `DEDENT` tokens, like Python.

22 LTML Schema Definition (LSD)

22.1 Overview

- An LTML Schema Definition defines *types*.
- Each type is of a specific *kind*.
- There is a pre-defined set of type constructors.
  - Each type constructor yields types of a specific kind.
- An LSD should normally define at least one `document` type (type of kind `document`).

22.2 Syntax

- The syntax is defined by the EBNF grammar together with the set of type constructors.
- Parsing LSD is not yet implemented (TODO); relatedly, the syntax is likely to change.

23 Legal Text Markup Language (LTML)

LTML is a language designed to write legal texts, specifically German "Fachprüfungsordnungen", but to be flexible enough to use for any kind of structured text.

The flexibility is achieved via the LTML Schema Definition (LSD) language, which defines types for LTML nodes and how these may be composed.

LTML documents are written partially via a GUI, and partially via a text editor.

The syntax (of the text portions) is inspired by Markdown, among others.

LTML document containers are to be converted to an output format (specifically, PDF and HTML).

23.1 Structure

LTML defines document containers, which are made up of documents, but are sometimes themselves referred to as documents.

See also:

- Syntax (outdated)

23.2 Labels & references

- Nodes can generally be labeled, and referenced in text using those labels.

23.3 Comments

LTML documents permit C-style line comments; that is, two forward slashes (`//`) generally cause text until the end of line to be ignored. This does not work within language constructs enclosed in curly brackets.

24 Type constructors

- This is semi-formal, and to be read together with the EBNF grammar.
- A type constructor header is to be read as `type/KindName constructor TypeConstructorName`
- `format-string(A, B, ...)` matches strings that may contain `{A}`, `{B}`, ...
- `[type/Kind]` matches comma-separated lists of defined types of kind `Kind`.
- `regexp([type/Kind])` matches simple regular expressions built from defined types of kind `Kind` as atoms and combined using , `|`, `?`, `*`, `+`, `()`.

24.1 Document type constructors

There is only one type constructor for the document kind.

```
type/document constructor document:
  keyword: NodeKeyword
  header_nodes: [type/header-node]
  body_nodes: regexp([type/body-node])
```

24.2 Header node type constructors

none (TODO)

24.3 Body node type constructors

```
type/body-node constructor section:
  keyword: NodeKeyword
  has_title: Bool
  output_identifier_format: format-string("arabic", "alph", "Alph")
  output_heading_location: "top-center"|"left-top"
  output_heading_format: format-string("identifier")
  heading_line_children: regexp([type/body-node])
  children: regexp([type/body-node])
  permit_label: Bool
  identifiers_are_fixed: Bool
```

```
type/body-node constructor text:
  keyword: NodeKeyword
  output_identifier_format: format-string("arabic", "alph", "Alph")
  line_children: regexp([type/body-node])
  permit_label: Bool
  identifiers_are_fixed: Bool

type/body-node constructor text-block:
  keyword: NodeKeyword
  output_identifier_format: format-string("arabic", "alph", "Alph")
  line_children: regexp([type/body-node])
  permit_label: Bool
  identifiers_are_fixed: Bool

type/body-node constructor text-list:
  item_keyword: NodeKeyword
  item_output_identifier_format: format-string("arabic", "alph", "Alph")
  item_output_key_format: format-string("identifier")
  item_line_children: regexp([type/body-node])
  item_permit_label: Bool
  item_identifiers_are_fixed: Bool

type/body-node constructor footnote:
  keyword: NodeKeyword
  has_title: Bool
  output_identifier_format: format-string("arabic", "alph", "Alph")

type/body-node constructor document-list:
  keyword: NodeKeyword
  documents: [type/document]
```

25 Document containers

An LTML document container can be seen as a large document, which is made up of smaller actual documents.

More precisely, it is composed of

- a header, which contains some metadata,
- a main document, and
- a list of appendix sections.

25.1 Header

The header may contain the following nodes.

- Title

- Output header (displayed on the top of each page)
- Output footer (displayed on the bottom of each page)

25.2 Appendix section

An appendix section has a title and contains a list of documents (appendices/attachments).

26 Documents

An LTML document is composed of:

- a header: some metadata
- a body: a tree of *nodes*
- a set of labeled footnotes

26.1 Header

The header may contain the following nodes.

- Title

26.2 Body

The body is composed of:

- an intro: a sequence of fixed-type simple sections
- a main part: a section body
  - One may thus see a document as a fancy section.
- an outro: a sequence of fixed-type simple sections

26.3 Footnotes

- In the AST, footnotes are part of the document in the form of a map from labels to footnotes.
- In the input, these occur within leaf sections or leaf simple sections that are part of the document's body.
- In the output, footnotes are placed appropriately.

27 Enumerations

Enumerations are composed of a list of textual items, which are written as a consecutive sequence of text children, each headed by a keyword that is specific to the enumeration type.

27.1 Enumeration text

Enumeration text permits styling, footnote references, and (nested) enumerations.

27.2 Enumeration identifiers

Enumeration items get assigned a sequential enumeration identifier (e.g., a number). Optionally—depending on the enumeration type—this identifier is automatically printed next to the respective item in the output (c.f. LaTeX' `enumerate` and `itemize` environments).

The enumeration identifiers can always be produced by referencing the respective items.

27.3 Nesting

Enumerations may indirectly contain sub-enumerations: the text within an enumeration may contain an enumeration.

27.4 Example

```
Some sentence with
  # one enumeration item,
  #{itm:} a labeled item ({:itm}),
  # another item,
    which spans
    multiple lines,
  # an item
      # with
      # sub-items
    and more text,
  #
    and an item starting in a new line.
Another sentence
  # has a single item.
```

Assuming an enumeration type with keyword `#` and numbered (`N.`) output, we get:

```
Some sentence with
  1. one enumeration item,
  2. a labeled item (2),
  3. another item, which spans multiple lines,
  4. an item
       1. with
       2. sub-items
     and more text,
```

```
  5. and an item starting in a new line.
Another sentence
  1. has a single item.
```

28 Footnotes

Footnotes are written as keyword-headed text, without initial indentation, where the keyword depends on the footnote's type.

Footnotes may be referenced as footnotes within the same document, in which case they get inserted soon after their first footnote reference.

Footnotes that are not referenced as such are ignored.

28.1 Footnote text

Footnote text permits styling, but not footnote references or enumerations.

28.2 Example

```
Some text with a footnote{^:fn} and more text.

^{fn:} The footnote.
```

Example output (assuming footnote keyword `^`):

```
Some text with a footnote[1] and more text.

---
[1] The footnote.
```

29 Paragraphs

Paragraphs are composed of text.

Unlike most other nodes, paragraphs are not introduced by keywords. They are just text, and separated by (one or more) empty lines.

```
This is a simple paragraph.

This is another paragraph.  With another sentence.
And another.
```

In the output, however, paragraphs are not merely seperated by whitespace, but each each paragraph is provided with its output identifier (typically, a number).

29.1 Labeling

Paragraphs are special w.r.t. labeling, in that the labeling is to be written on its own line, directly preceding the paragraph:

```
{example:}
Some paragraph.

Another paragraph referring to paragraph {:example}.
```

See also sentence labeling.

29.2 Paragraph text

Paragraph text permits all of styling, footnote references, and enumerations.

Additionally, paragraph text extends text with sentences.

29.3 Sentences

Text within a paragraph is split into sentences, Sentences are generally terminated by a single period (`.`), exclamation mark (`!`), or question mark (`?`), each. Additionally, they are terminated by enumeration items (with the next sentence starting after the sequence of enumeration items).

Sentence termination can be undone by inserting the continuation token `{>}` where otherwise a new sentence might start. This is meant for continuing a sentence after an enumeration, where it is to be inserted at the start of the first line after the enumeration. Otherwise, one may also simply escape the line terminator, s.t., e.g., `X\. Y.` is equivalent to `X. {>} Y.`).

Sentences are not full nodes; in particular, styling may overlap with sentences.

However, sentences may be labeled at their beginning (and referenced). Paragraph labeling takes precedence over sentence labeling; to label a sentence at the start of a paragraph (and best always), write the labeling on the same line as the start of the sentence.

29.3.1 Enumerations

There should be at most one enumeration per sentence; otherwise, it is generally impossible to properly reference an enumeration item, for the enumeration item output identifiers would generally be reused within the sentence. This is not enforced; i.e., it is the responsibility of the user.

Note that adding multiple enumerations to a sentence requires either using continuation tokens (see above) or using enumerations of different types in direct succession.

Further, enumerations generally don't start sentences, but this can be enforced by prepending a sentence labeling (in a separate line). In particular, if a paragraph starts with an enumeration, that enumeration does normally not belong to any sentence.

29.4 Example

```
{paragraph_a:}
This paragraph uses all text features in a minimalistic way.
Some of this text is <*bold>, </cursive>, <_underlined>.
This is an enumeration:
  # enumeration item 1
  # enumeration item 2
    # nested enumeration item
Next, we use a footnote{^:fn}.
{sentence_a:} This sentence has a label.
This is a reference to the preceding sentence: {:sentence_a}.
This sentence spans
multiple
lines.  {sentence_b:} This sentence starts in-line.
```

This assumes an enumeration keyword `#`, and an elsewhere-defined footnote with label `fn`.

See also the examples for general text.

30 Sections

Sections have an unstyled-textual heading and a body.

The heading and body are separated by at least one linebreak.

Sections are identified by keywords, which precede the heading.

Compare Markdown's different headings (`#`, `##`), MediaWiki's sections (`==`, `===`, …), and LaTeX' sections (`\section`, `\subsection`, …).

30.1 Heading text

Heading text permits neither styling nor enumerations, but footnote references.

Further, heading text is headed by the (possibly labeled) section keyword (i.e., one level of indentation is implicitly added—noticeable with heading text that spans multiple lines).

30.2 Body

The body is a fixed-type sequence of any of:

- lower-level sections
- paragraphs
- simple blocks

These body elements may be separated by any number of empty lines from each other.

Additionally, if the section is a leaf section (that is, does not contain sub-sections, but only either paragraphs or simple blocks), it may contain footnotes in place of any paragraph or simple block. These footnotes do not become part of the section, but rather the encompassing document.

30.3 Example

```
={main:} Main

§{section_a:} Some section

This paragraph is in {:section_a} in super-section {:main}.

This is another paragraph in {:section_a}.
Paragraphs don't have keywords and are just separated by empty lines.

§{section_b:} Another section, with a title
  spanning
  several lines
  and a footnote{^:fn}

This paragraph is in {:section_b} in super-section {:main}.

^{fn:} The heading's footnote.
```

This assumes two section types, one with keyword `=` and another with keyword `§`, where the former may contain the latter, and the latter may contain paragraphs.

See also the examples for paragraphs and general text.

31 Simple blocks

A simple block is any of:

- a simple paragraph
- a table

A simple block type is a tuple of:

- a simple paragraph type
- a set of table types

32 Simple Paragraphs

Simple paragraphs are just like regular paragraphs, except:

- They do not have an output identifier.
- They cannot be labeled or referenced.
- They do do not contain formal sentences (i.e., sentences that can be labeled and referenced).

33 Simple sections

Simple sections are composed of simple paragraphs.

Unlike regular sections, they do not have a heading or identifier.

The simple section's children are separated by any number of linebreaks.

Simple sections are identified by keywords, which occur on their own line, preceding the simple section's children.

Additionally, a simple section may contain footnotes in place of any simple paragraph, which are added to the encompassing document.

33.1 Example

```
[intro]

This is a simple paragraph.
```

This assumes a simple section type with keyword `[intro]`.

34 Tables

TODO.

35 Text

Some nodes contain *text*, which may generally be styled, and contain references, footnote references, and enumerations.

35.1 Text kinds

There are several text kinds (e.g., paragraph text), which are determined by where text occurs, but do not depend on node types.

The text kind determines whether styling, footnote references, and enumerations within text are permitted, each, and may further extend text.

References are always permitted.

35.2 Line breaks & Whitespace

Text may be spread over several lines—all with the same minimum indentation. Such input lines form a single logical line, joined by whitespace. Text is terminated by a final newline character.

Any internal whitespace—either from a single linebreak as described above, or a non-empty sequence of ASCII spaces, but neither initial nor final—is treated the same: as a word separator, which is generally rendered as a single space character. Normally, lines in the output are automatically broken when full.

A hard line break can be encoded as `{nl}`, and a non-breaking space as `~`.

Empty lines are disallowed within text. Note, however, that empty lines may be used to split up paragraphs, which are otherwise basically just text.

35.3 Escaping

Any non-whitespace character that is generally valid may be escaped by prepending a backslash (\), yielding the respective literal character. This is useful for characters with special meaning. In particular, a literal backslash can be encoded as `\\`.

All Unicode characters except Unicode control characters (`Cc`; e.g., `U+007F DEL`) are generally valid.

35.4 Keyword-headed text

In some contexts (e.g., enumerations), text is headed by a keyword.

In this case, in order to belong to the headed text, any input line following the first must be indented more than that first line, but not necessarily by the same amount.

The first line may generally be written right after the keyword (in the same line, separated by at least one ASCII space); otherwise, it starts on the subsequent line, indented.

See enumerations for example input.

35.5 Styling

- Some (but not all) node kinds permit styled text.
- Specifically, text may be `<*bold>`, `</in italics>`, or `<_underlined>`.
- Different style tags may be nested (e.g., `<*bold </and italics>>`).
  - Nesting the same style tag within itself is possible but of little use.
    - E.g., `<*bold <*again>>` is visually equivalent to `<*bold again>` in the output.

- ```
  <*Styled text can
  span multiple lines,
    # contain enumeration items,
  and continue afterwards---at the same indentation level.>
  ```

35.6 References

Labeled nodes may be referenced within text, as `{:LABEL}`, where `LABEL` is the respective label. In the output, the respective output identifier is substituted.

35.6.1 Example

```
{p:}
This is a labeled paragraph.

This is a reference to paragraph {:p}.
```

Output:

```
This is a labeled paragraph.

This is a reference to paragraph 1.
```

35.7 Footnote references

- Footnotes may generally be referenced within text, as `{^:LABEL}`, where `LABEL` is the respective footnote's label.
  - Note that the `^` character is fixed, and unrelated to the footnote type's keyword (which is also commonly `^`).
- A footnote reference is substituted by a correspondingly formatted output identifier in the output.
- Footnote references have document scope:
  - Footnote references must reference footnotes defined in the same document.
  - Footnote output identifiers are unique within a document, but generally not over several documents.
- A footnote may be referenced repeatedly.
- Note that footnotes may also be referenced using ordinary references, in which case they are not formatted as footnote references.
- Example.

35.8 Child nodes

Text nodes (e.g., headings, sentences) generally permit in-line children. That is, at any point in such a text node, certain child nodes may be inserted. This requires breaking the line where the children are to be inserted.
```

Child nodes need not be indented further than the context, but this is deemed good practice. Similarly, they should not appear before any regular text, but this is not prohibited.

There is currently one kind of text children, enumeration items.

35.8.1 Example

```
Everybody loves these fruits:
  # Apples, unless
      # unripe, or
      # not tasty
  # Bananas,
    if yellow
  # Oranges
```

This example is to be read as containing a single nested enumeration.

36 Identifiers

Nodes may have input and output identifiers, which are generally independent. Both depend on the node type, defined in the respective LSD.

36.1 Input identifiers

Input identifiers (also: *keywords*) are only visible in the input.

Keywords are either a single non-bracketing symbol chararacter (e.g., `#`, `§`), or a simple string enclosed in square brackets (e.g., `[intro]`). TODO: Define simple string.

36.2 Output identifiers

Output identifiers are automatically generated, and inserted in appropriate places in the output. Typically, they are printed alongside the respective nodes. They are also printed in place of references.

Examples: `42`, `a`, `42a`

36.2.1 Fixed identifiers

TODO: Outdated. Nodes may have their output identifiers fixed upon publication, in which case such a node cannot be removed, but only marked as removed (`-` suffix), and further such nodes can only be inserted with the special `*` suffix on the section keyword.

37 Labels

Body nodes may generally be labeled, in which case one may reference them.

Labeling is only possible if the respective node has an output identifier.

37.1 Label syntax

A label must start with a lower-case ASCII letter, and is otherwise composed of lower-case ASCII letters, ASCII digits, `_`, and `-`.

37.2 Labeling syntax

A labeled node is written as `KEYWORD{LABEL:} CONTENT`, where

- `KEYWORD` is empty if the respective node does not have a keyword, and
- `CONTENT` is separated from the labeling by any amount of whitespace, which generally may contain at most one newline.

Special rules apply for paragraphs and sentences.