Karlsruhe Institute of Technology

Broadcasting Webservice for Smart TVs

Lennard Kittner, Henry Boos,
Jiangang Huang, Sicheng Dong,
Jannik Wibker

Contents

1 Packages & Project structure

The frontend and dashboard are written in Typescript which does not have the same concept of packages as Java does. Therefore the directory structure and layout of these projects is detailed instead of the packages.

1.1 Frontend

1.1.1 Directory structure

```
frontend/
 ├─ public/
 └─ src/
     ├─ index
     ├─ App
     ├─ components/
     │   ├─ Layout
     │   ├─ VerticalSplitSlot
     │   ├─ HorizontalSplitSlot
     │   ├─ CarouselSlot
     │   ├─ WidgetSlot
     │   ├─ EmptySlot
     │   └─ Intl
     ├─ css/
     ├─ icons/
     ├─ plugins/
     │   ├─ Cafeteria/
     │   ├─ Calendar/
     │   ├─ Clock/
     │   └─ Publication/
     ├─ stores/
     │   └─ UIStore
     ├─ test/
     └─ types/
```

```
│
│  ├── Language
│  ├── Plugin
│  ├── Store
│  └── config/
│      ├── General
│      ├── Layout
│      └── Plugins
└── util/
```

1.1.2 Descriptions

public

This directory contains all static content like the `index.html` file, themes, and favicons.

src/components

This directory contains common component used by plugins or more general components used by the layout system or other core functionalities.

The layout system itself is also a component which is placed in this folder.

One of the most widely used components is the Intl component which provides translations.

src/index

This file is the entrypoint to the whole application.

src/icons

This directory contains common icons that can be used by the widgets.

src/plugins

This directory contains the pre-bundled plugins. This includes the **Cafeteria**, **Calendar**, **Clock** and **Publication** plugins.

src/stores

contains global data stores such as the `UIStore`.

src/test

This directory contains unit tests.

src/types

This directory contains type definitions. This includes Types for configuration files.

src/util

This directory contains utility functions.

1.2 Backend

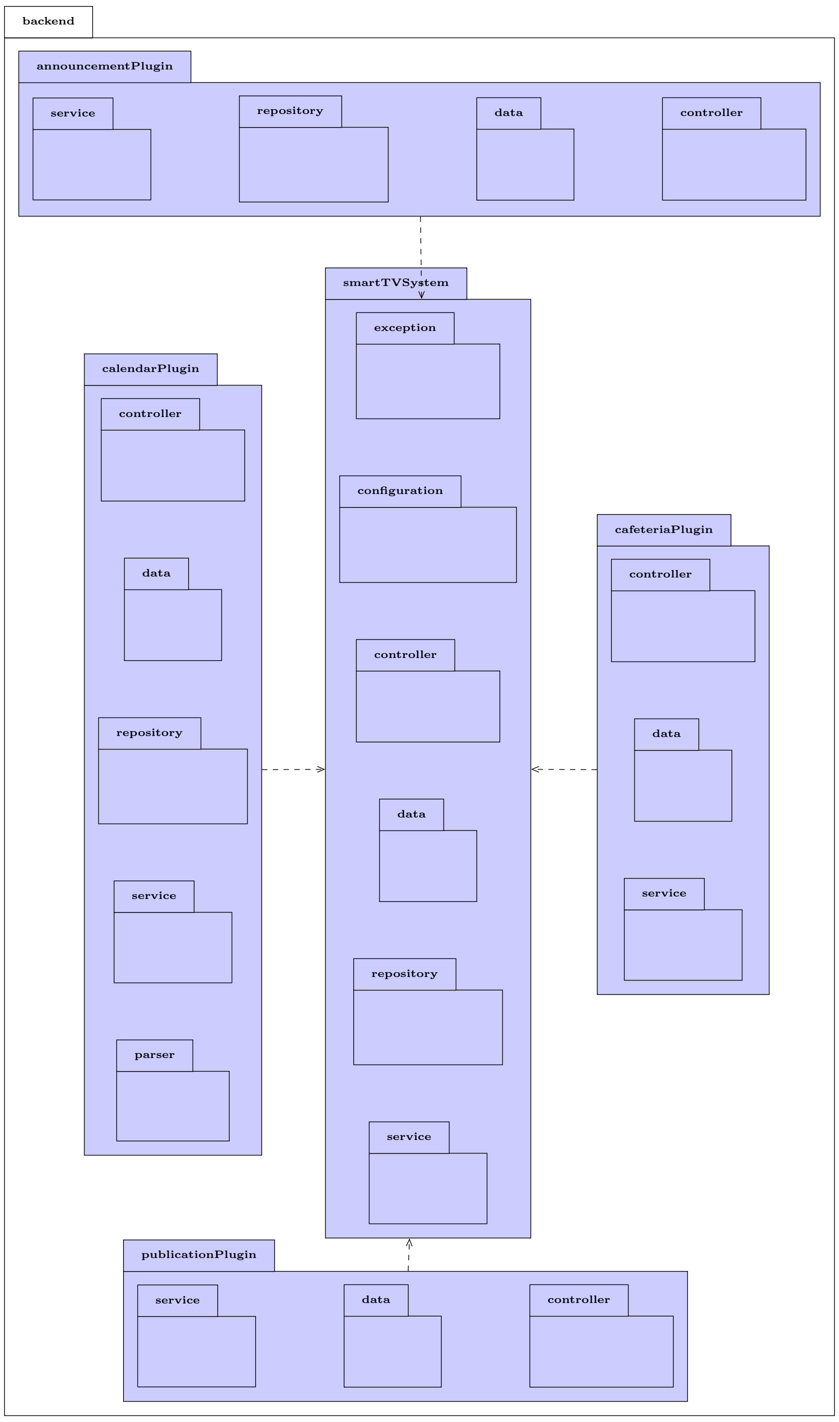


Figure 1.1: backend package structure

1.2.1 General Structure

Every plugin hast its own package. The main package (**smartTVSystem**) provides interfaces that can be used or extended by plugin classes. This enables services of the main package to discover and use plugins. Furthermore it helps to guarantee consistency across the system, e.g. the way a http response should be structured.

Every plugin has the same internal package structure.

1.2.2 Controller

This package contains *controllers*. *Controllers* are classes that provide the REST API endpoints.

1.2.3 Service

This package contains *services*. *Services* are classes providing special functionality to other classes, e.g. saving an announcement to a repository. A single instance of a service is shared between all classes.

1.2.4 Data

This package contains definitions of data types.

1.2.5 Repository

This package contains *repositories*. A *repository* is used to directly store data or interface with the database.

1.2.6 Configuration

This package contains configuration classes to configure the spring application.

1.2.7 Exception

This package contains exceptions that can be then used by other classes.

1.2.8 Parser

This package contains classes that are used to parse data obtainend from external sources.

1.3 Dashboard

1.3.1 Directory structure

```
dashboard/
├── public/
└── src/
    ├── components/
    │   ├── ConfigFileSection
    │   ├── Intl
    │   ├── Login
    │   ├── PageWrapper
    │   ├── PrivateRoute
    │   ├── Table
    │   ├── TimeInput
    │   └── Button
    ├── pages/
    │   ├── CoreConfigurationPage
    │   ├── PluginConfigurationPage
    │   ├── ErrorLogPage
    │   ├── AnnouncementsPage
    │   └── CalendarPage
    ├── css/
    ├── test/
    ├── types/
    └── util/
```

1.3.2 Descriptions

public

This directory contains all static content like the `index.html` file, themes, and favicons.

src/components

This directory contains components used by pages and by the core code of the dashboard.

An example is the Intl component which is used by pretty much everything. It is used for providing translations.

src/pages

This directory contains a react component for each page that the dashboard can show (core configuration, error logs, ...).

src/test

This directory contains unit tests.

src/types

This directory contains type definitions.

src/util

This directory contains utility functions.

2 API endpoints

In the following the API endpoints are described that are used to comunicate with the backend.

2.1 Data

2.1.1 Announcements

Get a list of all announcements

Request: `GET: /api/announcements/`

Role: No role requirement

The response will be of the following structure:

```
type AnnouncementsListResponse = {
  message: "successfully listed announcements"
         | "failed to list announcements",
  status: "success" | "failure",
  data: {
    id: number,
    message: string,
    urgency: "low" | "medium" | "high",
    startTime: number,
    endTime: number
  }[]
}
```

Retrieve an announcement by its id

Request: `GET: /api/announcements/<id>`

Role: No role requirement

The response will be of the following structure:

```
type AnnouncementsGetResponse = {
  message: "successfully retrieved announcement"
         | "failed to retrieve announcement",
  status: "success" | "failure",
  data: {
    id: number,
    message: string,
    urgency: "low" | "medium" | "high",
    startTime: number,
    endTime: number
  }
}
```

Create a new announcement

Request: `POST: /api/announcements/`

Role: Editor or Admin

The request body needs to be of the following structure:

```
type AnnouncementsCreateRequest = {
  message: string,
  urgency: "low" | "medium" | "high",
  startTime: number,
  endTime: number
}
```

The response will be of the following structure:

```
type AnnouncementsCreateResponse = {
  message: "successfully created announcement"
         | "failed to create announcement",
  status: "success" | "failure",
  data: {
    id: number,
    message: string,
    urgency: "low" | "medium" | "high",
    startTime: number,
    endTime: number
  }
}
```

Update an announcement

Request: `PATCH: /api/announcements/<id>`

Role: Editor or Admin

The request body needs to be of the following structure:

```
type AnnouncementsCreateRequest = {
  message: string,
  urgency: "low" | "medium" | "high",
  startTime: number,
  endTime: number
}
```

The response will be of the following structure:

```
type AnnouncementsUpdateResponse = {
  message: "successfully updated announcement"
         | "failed to update announcement",
  status: "success" | "failure",
  data: {
    id: number,
    message: string,
    urgency: "low" | "medium" | "high",
    startTime: number,
    endTime: number
  }
}
```

Delete an announcement

Request: `DELETE: /api/announcements/<id>`

Role: Editor or Admin

The response will be of the following structure:

```
type AnnouncementsDeleteResponse = {
  message: "successfully deleted announcement"
         | "failed to delete announcement",
  status: "success" | "failure"
}
```

2.1.2 Calendars

Get a list of all calendars

Request: `GET: /api/calendars/<internal | external>`

Role: No role requirement

The response will be of the following structure:

```
type CalendarListResponse = {
  message: "successfully listed calendars"
         | "failed to list calendars",
  status: "success" | "failure",
  data: {
    id: number,
    timeOfLastUpdate: number,
    name: string,
    color: string,
    events: {
      id: number,
      uid: string,
      name: string,
      startTime: number,
      endTime: number,
      location: string,
      isAllDay: boolean
  }[],
  internal: boolean
}[]
```

Retrieve a calendar by its id

Request: `GET: /api/calendars/<internal | external>/<calendar id>`

Role: No role requirement

The response will be of the following structure:

```
type CalendarEventsGetResponse = {
  message: "successfully retrieved calendar event"
         | "failed to retrieve calendar event",
  status: "success" | "failure",
  data: {
    id: number,
    timeOfLastUpdate: number,
    name: string,
    color: string,
    events: {
      id: number,
      uid: string,
      name: string,
      startTime: number,
      endTime: number,
      location: string,
      isAllDay: boolean
  }[],
  internal: boolean
}
```

Create a new calendar event

Request: `POST: /api/calendars/internal/<calendar id>`

Role: Editor or Admin

The request body needs to be of the following structure:

```
type CalendarEventsCreateRequest = {
  id: number,
  uid: string,
  name: string,
  startTime: number,
  endTime: number,
  location: string,
  isAllDay: boolean
}
```

The response will be of the following structure:

```
type CalendarEventsCreateResponse = {
  message: "successfully created calendar event"
         | "failed to create calendar event"
         | "could not find calendar",
  status: "success" | "failure",
  data: {
    id: number,
    uid: string,
    name: string,
    startTime: number,
    endTime: number,
    location: string,
    isAllDay: boolean
  }
}
```

Update a calendar event

Request: `PATCH: /api/calendars/internal/<event id>`

Role: Editor or Admin

The request body needs to be of the following structure:

```
type CalendarEventsUpdateRequest = {
  uid: string,
  name: string,
  startTime: number,
  endTime: number,
  location: string,
  isAllDay: boolean
}
```

The response will be of the following structure:

```
type CalendarEventsUpdateResponse = {
  message: "successfully updated calendar event"
         | "failed to update calendar event",
         | "could not find calendar event",
  status: "success" | "failure",
  data: {
    id: number,
    uid: string,
    name: string,
    startTime: number,
    endTime: number,
    location: string,
    isAllDay: boolean
  }
}
```

Delete a calendar event

Request: `DELETE: /api/calendars/internal/<event id>`

Role: Editor or Admin

The response will be of the following structure:

```
type CalendarEventsDeleteResponse = {
  message: "successfully deleted calendar event"
         | "failed to delete calendar event",
  status: "success" | "failure"
}
```

2.1.3 Publications

Get a list of all publications

Request: `GET: /api/publications/`

Role: No role requirement

The response will be of the following structure:

```
type PublicationsListResponse = {
  message: "successfully listed publications"
         | "failed to list publications",
  status: "success" | "failure",
  data: {
    id: number,
    title: string,
    authors: string[],
    description: string | null,
    publisher: string | null,
    publishLocation: string | null,
    publishDate: string,
  }[]
}
```

Retrieve a publication by its id

Request: `GET: /api/publications/<id>`

Role: No role requirement

The response will be of the following structure:

```
type PublicationsGetResponse = {
  message: "successfully retrieved publication"
         | "failed to retrieve publication",
  status: "success" | "failure",
  data: {
    id: number,
    title: string,
    authors: string[],
    description: string | null,
    publisher: string | null,
    publishLocation: string | null,
    publishDate: string
  }
}
```

2.1.4 Cafeteria

Get a list of all cafeterias

Request: `GET: /api/cafeterias/`

Role: No role requirement

The response will be of the following structure:

```
type CafeteriaGetResponse = {
  message: "successfully listed cafeterias"
         | "failed to list cafeterias",
  status: "success" | "failure",
  data: {
    id: number,
    timeOfLastUpdate: number,
    name: string,
    openingHours: {
      id: number,
      name: string,
      openingTime: number,
      closingTime: number
    }[],
    lines: {
      id: number,
      lineName: string,
      dishes: {
        id: number,
        mealName: string,
        price: number,
        servedOn: number,
        classifiers: string[],
        additives: number[]
      }[]
    }[],
    additivesLegend: { [name: string]: string }
  }[]
}
```

Get a list of all dishes served at a specific line on a specific day of the week

Request:

`GET: /api/cafeterias/<cafeteriaID>/<lineID>/<mon|tue|wed|thu|fri|saty|sun>/`

Role: No role requirement

The response will be of the following structure:

```
  type CafeteriaGetResponse = {
    message: "successfully retrieved dishes"
           | "failed to retrieve dishes",
    status: "success" | "failure",
    data: {
      id: number,
      mealName: string,
      price: number,
      servedOn: number,
      classifiers: string[],
      additives: number[]
    }[]
  }
```

Retrieve a line by its id

Request: `GET: /api/cafeterias/<cafeteriaID>/<lineID>/`

Role: No role requirement

The response will be of the following structure:

```
  type CafeteriaGetResponse = {
    message: "successfully retrieved line"
           | "failed to retrieve line",
    status: "success" | "failure",
    data: {
      id: number,
      lineName: string,
      dishes: {
        id: number,
        mealName: string,
        price: number,
        servedOn: number,
        classifiers: string[],
        additives: number[]
      }[]
    }
  }
```

Retrieve a cafeteria by its id

Request: `GET: /api/cafeterias/<cafeteriaID>/`

Role: No role requirement

The response will be of the following structure:

```
  type CafeteriaGetResponse = {
    message: "successfully retrieved cafeteria"
           | "failed to retrieve cafeteria",
    status: "success" | "failure",
    data: {
      id: number,
      timeOfLastUpdate: number,
      name: string,
      openingHours: {
        id: number,
        name: string,
        openingTime: number,
        closingTime: number
      }[],
      lines: {
        id: number,
        lineName: string,
        dishes: {
          id: number,
          mealName: string,
          price: number,
          servedOn: number,
          classifiers: string[],
          additives: number[]
        }[]
      }[],
      additivesLegend: { [name: string]: string }
    }
  }
```

2.2 Update

Get the aggregated data of all Data endpoints

Request: `GET: /update/`

Role: No role requirement

The response will be of the following structure:

```
type UpdateResponse = {
  message: "successfully aggregated services"
         | "failed to aggregate services",
  status: "success" | "failure",
  data: {
    timeOfLastRelevantUpdate: number,
    aggregatedData: {
      name: <endpoint name>,
      data: <endpoint latest data>
    }[]
  }
}
```

2.3 Configuration

Get config file by name

Request: `GET: /config/<data|widget|core>/<filename>.yml`

Role:

Admin for `/config/data/<filename>.yml`
None for `/config/<widget|core>/<filename>.yml`

Response:

The response will just contain the config file. The config file will be written in YAML.

Each core configuration file has a predefined structure, these can be found here: Types for configuration files.

Pre-bundled plugins also have exact type definitions for their plugins.

Set config file with name

Request: `POST: /config/<data|widget|core>/<filename>.yml` with the config file as the body.

The config file must be written in YAML and follow strict structure guidelines.

Each core configuration file has a predefined structure, these can be found here: Types for configuration files.

Pre-bundled plugins also have exact type definitions for their plugins.

Role: Admin

The response will be of the following structure:

```
type ConfigFileGetResponse = {
  message: "successfully set config file"
         | "failed to set config file",
  status: "success" | "failure",
}
```

2.4 Error logging

Get the error log file

Request: `GET: /log/`

Role: Admin

The response will just contain the log file. Each line is formatted in the following way:

```
[<timestamp>;<urgency>]: <message>
```

All newlines in the message are escaped, the timestamp is formatted according to ISO 8601.

Add log entry

Request: `POST: /log/`

Role: No role requirement

The request body needs to be of the following structure:

```
type ErrorLogAppendRequest = {
  message: string,
  timestamp: string,
  urgency: "low" | "medium" | "high"
}
```

The response will be of the following structure:

```
type ErrorLogAppendResponse = {
  message: "successfully added log entry"
         | "failed to add log entry",
  status: "success" | "failure"
}
```

2.5 Internationalization

Localization files (or translation files or i18n files) are stored in the backend and retrieved by both the frontend and dashboard.

Localization files follow the definition provided by the Language type definition and are provided in JSON format.

Get localization file

Request: `GET: /i18n/<language_id>.json`

Role: No role requirement

The response will be of the following structure:

```
type Language = {
  name: string,
  locale: string,
  words: { [name: string]: string }
}
```

2.6 Miscellaneous

Get uptime

Request: `POST: /uptime/`

Role: No role requirement

The response will be a timestamp of the servers uptime.

Get current version

Request: `POST: /current-version/`

Role: No role requirement

The response will be the current version.

3 Class descriptions - Frontend

3.1 Types for configuration files

3.1.1 Location

Data type describing how a location might be expressed.

```
type Location = { city_id: number }
              | { city_name: string, country_code: string }
              | { zip: string, country_code: string }
              | { lat: number, long: number }
```

All of those formats are valid ways of expressing a location. Some formats are more accurate than others, searching for a city name might yield multiple cities with the same name. Using a format that describes the location precisely is therefore recommended.

3.1.2 GeneralConfig

Datatype for the datastructure of the general.yml config file.

```
type General = {
  default_timezone: string,
  alternative_timezones: string[],
  default_location: Location,
  available_languages: string[],
  language_switch_interval: number,
  languages: string[],
  date_format: string,
  available_themes: string[],
  themes: ({ theme: string } & (TimeRestriction | {}))[],
}
```

Fields

- `default_timezone: string`
  Accepts multiple formats of specifying timezones:

  - <Continent>/<City>
  - GMT+<offset> (GMT can be used instead of GMT+0)
  - UTC+<offset> (UTC can be used instead of UTC+0)
  - timezone abbreviation (e.g. CEST or CET)

  <Continent>/<City> is the only option that automatically determines wether daylight saving time is active

  Default value: `"Europe/Berlin"`
  Example: `"CEST"`, `"Asia/Shenzhen`, `"UTC+3`, `"GMT"`

- `alternative_timezone: string[]`
  Accepts multiple formats of specifying timezones:

  - <Continent>/<City>
  - GMT+<offset> (GMT can be used instead of GMT+0)
  - UTC+<offset> (UTC can be used instead of UTC+0)
  - timezone abbreviation (e.g. CEST or CET)

  <Continent>/<City> is the only option that automatically determines wether daylight saving time is active

  Default value: `[]`
  Example: `["CEST", "Asia/Shenzhen"]`, `["UTC", "Europe/Athens"]`

- `default_location: Location`
  Specifies the location of the smart TV. This can be used by plugins which require location information (e.g. for displaying weather data).

- `available_languages: string[]`
  Accepts an array of available languages. Must be non-empty.

  The languages should be specified by their full name or a shortened code (ISO-639-1 or later) identifying them.
  Full name refers to the name of the language in english in all lower case, this can

be prefixed with the variant (dialect, writing system or similar).

Note: These are just guidelines, they are not strictly enforced; the translation files need to declare which name refers to them (e.g a german translation might declare that "german" or "de" refers to it). The files must be named accordingly.
Default value: `"english"`
Example: `"german"`, `"de"`, `"simplified_chinese"`, `"zh-hans"`, `"spanish"`, `"es"`

- `available_languages: string[]`
  Accepts an array of languages to display. Must be a non-empty subset of `available_languages`.

  The languages should be specified by their full name or a shortened code (ISO-639-1 or later) identifying them.

  Full name refers to the name of the language in english in all lower case, this can be prefixed with the variant (dialect, writing system or similar).

  Note: These are just guidelines, they are not strictly enforced; the translation files need to declare which name refers to them (e.g a german translation might declare that "german" or "de" refers to it). The files must be named accordingly.
  Default value: `"english"`
  Example: `"german"`, `"de"`, `"simplified_chinese"`, `"zh-hans"`, `"spanish"`, `"es"`

- `language_switch_interval: number`
  Interval between languages switches in seconds. If set to `0` the language stays the same all the time.

  Default value: `0`
  Example: `60`

- `languages: string[]`
  Accepts an array of languages to display. Must be a non-empty subset of `available_languages`.

  The languages should be specified by their full name or a shortened code (ISO-639-1 or later) identifying them.
  Full name refers to the name of the language in english in all lower case, this can be prefixed with the variant (dialect, writing system or similar).

  Note: These are just guidelines, they are not strictly enforced; the translation files need to declare which name refers to them (e.g a german translation might declare that "german" or "de" refers to it). The files must be named accordingly.
  Default value: `"english"`

Example: `"german"`, `"de"`, `"simplified_chinese"`, `"zh-hans"`, `"spanish"`, `"es"`

- `date_format: string`
  Date format the UI uses

`"y"`	year
`"m"`	month in numerical format
`"M"`	month as a word (Jan, ...)
`"d"`	day in a numerical format
`"D"`	day of the week as a word (Mon, ...)
`"ISO8601"`	ISO 8601 date string (not including time); alias for `"yyyy-mm-dd"`
`"LANGUAGE"`	date format should follow currently shown language

  Repeating letters indicate a longer version of the information. E.g. `"yy"` is a year with 2 digits, `"yyyy"` with 4. For `"d"` and `"m"` this means adding leading zeroes to potentially bring the total length up to 2 digits.

  Default value: `"dd.mm.yyyy"`
  Example: `"yyyy-mm-dd"`

- `({ theme: string } & (TimeRestriction | {}))[]`
  Accepts an array of themes with an optional time restriction to use.
  Themes are identified by their string name (which is all lowercase, can only contain letters, underscores (_) and dashes (-))

3.1.3 Widget

Describes a widget slot in the layout system.

```
type Widget = {
  widget: string,
  properties: { [name: string]: string }
}
```

Fields

- `widget: string`
  ID of the widget

- `properties: [name: string]: string`
  Properties to pass to the widget instance.

3.1.4 LayoutConfig

Datatype for the datastructure of the layout.yml config file.

```
type Layout = {
  root: Slot
}
```

Fields

- `root: Slot`
  Root slot of the whole layout system

  Default value: `{} as EmptySlot`

3.1.5 EmptySlot

Describes an empty slot in the layout.

```
type EmptySlot = {

}
```

An empty slot does not have any fields

3.1.6 CarouselSlot

Describes a carousel slot in the layout.

Switches through the slots it has with a set time interval.

```
type CarouselSlot = {
  time?: number,
  slots: Slot[]
}
```

Fields

- `item?: number`
  The time it takes for the displayed slot to be switched out for the next one. Time in seconds.

  Default value: `20`
  Example: `10`, `15`, `20`, `30`

- `slots: Slot[]`
  List of slots to cycle through. If the end of the list is reached restart from the front. If a slot cannot be displayed for some reason skip it. If the list is empty mirrors the behaviour of an EmptySlot.

  Default value: `[]`

3.1.7 HSplit

Describes a horizontal split in the layout.

```
type HSplit = {
  ratio?: number,
  left: Slot,
  right: Slot
}
```

Fields

- `ratio?: number`
  Number between 0 and 1 detailing how much space (percentage wise) the first slot gets.

  Note: Any space possibly required for seperators / spacing between both of the slots is excluded from the calculations.
  Default value: `0.5`
  Example: `0.3`, `0.5`, `0.7`

- `left: Slot`
  The slot on the left.

- `right: Slot`
  The slot on the right.

3.1.8 VSplit

Describes a vertical split in the layout.

```
type VSplit = {
  ratio?: number,
  top: Slot,
  bottom: Slot
}
```

Fields

- `ratio?: number`
  Number between 0 and 1 detailing how much space (percentage wise) the first slot gets.

  Note: Any space possibly required for seperators / spacing between both of the slots is excluded from the calculations.
  Default value: `0.5`
  Example: `0.3`, `0.5`, `0.7`

- `top: Slot`
  The slot on the top.

- `bottom: Slot`
  The slot on the bottom.

3.1.9 TimeRestriction

Restrict widgets or themes to only be displayed in certain time periods.

Note: If the specified time is invalid (see restrictions of fields) it is ignored.

- `from_time: [number, number]`
  (hour, minute)-tuple

Note: Uses 24h time. If the time is invalid this restriction is ignored (hour must be <24, minute must be <60).
Example: `[14, 30]` (for 14:30)

- `to_time: [number, number]`
  (hour, minute)-tuple
  Note: Uses 24h time. If the time is invalid this restriction is ignored (hour must be <24, minute must be <60).
  Example: `[14, 30]` (for 14:30)

3.1.10 Slot

Describes a slot in the layout. Slots can be of different subtypes (e.g. VSplit - vertical split).

Can have a TimeRestriction if needed.

```
type Slot = (Widget
              | EmptySlot
              | { hsplit: HSplit }
              | { vsplit: VSplit }
              | { carousel: CarouselSlot })
            & (TimeRestriction | {}})
```

3.1.11 PluginConfig

Datatype for the datastructure of the plugins.yml config file.

```
type Plugin = {
  available_data_plugins: string[],
  available_frontend_plugins: string[],
  available_dashboard_plugins: string[],
  active_data_plugins: string[],
  active_frontend_plugins: string[],
  active_dashboard_plugins: string[]
}
```

Fields

- `available_data_plugins: string[]`
  List of available backend / data plugins; elements refer to the plugin id in the data plugin code.

  Example: `["publication", "calendar", "cafeteria"]`

- `available_frontend_plugins: string[]`
  List of available frontend plugins; elements refer to the Plugin.id of a plugin.

  Example: `["clock", "calendar", "cafeteria"]`

- `available_dashboard_plugins: string[]`
  List of available dashboard plugins; elements refer to the Plugin.id of a plugin.

  Example: `["calendar", "image"]`

- `active_data_plugins: string[]`
  List of active backend / data plugins; elements refer to the plugin id in the data plugin code.

  Example: `["publication", "calendar"]`

- `active_frontend_plugins: string[]`
  List of active frontend plugins; elements refer to the Plugin.id of a plugin.

  Example: `["clock", "calendar"]`

- `active_dashboard_plugins: string[]`
  List of active dashboard plugins; elements refer to the Plugin.id of a plugin.

  Example: `["calendar"]`

3.2 Interfaces and Internationalization

3.2.1 Language

Type describing what a language configuration file (or Internationalization file) should look like.

```
type Language = {
  name: string,
  locale: string,
  words: { [name: string]: string }
}
```

- `name: string`
  Name of the language. This is also the filename of the translation file.

  Example: `"english"`

- `locale: string`
  Locale code of the language. Used for date format strings.

  Example: `"en-GB"`

- `words: { [name: string]: string }`
  Translations, mapping from an english word to its translation.

3.2.2 ErrorLogUrgency

Type (enum) representing the urgency of an error or warning.

```
enum ErrorLogUrgency {
  LOW    = "low",
  MEDIUM = "medium",
  HIGH   = "high"
}
```

3.2.3 AnnouncementUrgency

Type (enum) representing the urgency of an announcement.

```
enum AnnouncementUrgency {
  LOW    = "low",
  MEDIUM = "medium",
  HIGH   = "high"
}
```

3.2.4 Announcement

Type representing an announcement.

```
type Announcement = {
  message: string,
  urgency: AnnouncementUrgency,
  startTime: number,
  endTime: number,
}
```

Fields

- `message: string`
  The text of the announcement that will be displayed.
  Example: `"Warning internet outage at 12am!"`

- `urgency: AnnouncementUrgency`
  The urgency of the announcement. It may be either low, medium or high.
  Example: `AnnouncementUrgency.HIGH`

- `startTime: number`
  The time the announcement will start. This means it will be shown. The time is specified as a unix timestamp.

- `endTime: number`
  The time the announcement will end. This means it will no longer be shown. The time is specified as a unix timestamp.

3.2.5 Plugin

Interface for encapsulating a plugin (React Component and Store) and its metadata.

```
interface Plugin<APIData extends any[], ConfigData> {
  name: string
  id: string
  overwrite_api_endpoint?: string
  Component: Class<Component>
  store: Store<APIData, ConfigData>
}
```

`APIData` is tuple generic parameter (this is a workaround for typescript not supporting variadic generic parameters) which is used by the Store. The store will accept data retrieved from the API in this form.
`ConfigData` is a generic paramter which is used by the Store. The store will accept a config file of this format on startup.

Fields

- `name: string`
  Name of the plugin.

  Note: Can contain spaces, uppercase letters and so on. Will **not** be used to uniquely identify the plugin.
  Example: `"Clock"`

- `id: string`
  Id of a plugin.

  Will be used to uniquely identify the plugin.
  Note: Needs to be all lowercase; can contain alphanumeric characters (no leading digits), underscores (_) and dashes (-)
  Example: `"clock"`

- `overwrite_api_endpoint?: string`
  Use incase the API endpoint the plugin uses differs from the plugin id. This is useful incase multiple frontend plugins requrie data from the same endpoint.

- `component: Class<Component>`
  React component class of the widget.

- `store: Store<APIData, ConfigData>`
  mobx store for the widget which manages the widgets config file as well as any data the widget needs.

3.2.6 Store

Mobx store used by a single plugin.

```
interface Store<DS extends any[], C> {
  injectData(...data: DS): void
  injectConfig(config: Partial<C>): void
}
```

DS is a tuple of types which describe what API data can get injected.

Methods

- `injectData(...data: DS): void`
  Used for injecting data retrieved from the API into the store. Each argument of injectData is directly linked to the respective element of the DS tuple type.

- `injectConfig(config: Partial<C>): void`
  Used for injecting a config file into the store on startup.

3.3 Stores

3.3.1 UIStore

Mobx store which manages all UI but not layout related data. This mostly includes things configured in the general.yml config file but also translation files.

```
class UIStore {
  default_timezone: string
  alternative_timezones: string[]
  default_location: Location
  available_languages: string[]
  languages: string[]
  active_language: string
  language_switch_interval: number
  language_data: Language[]
  date_format: string
  available_themes: string[]
  themes: ({ theme: string } & (TimeRestriction | {}))[]
  active_theme: string
  constructor(): UIStore
  changeTheme(theme: string): void
  changeLanguage(language: string): void
  getLanguageByName(name: string): Language
}
```

Fields

- `default_timezone: string`
  Refer to GeneralConfig.default_timezone.
- `alternative_timezones: string[]`
  Refer to GeneralConfig.alternative_timezones.
- `default_location: Location`
  Refer to GeneralConfig.default_location.
- `available_languages: string[]`
  Refer to GeneralConfig.available_languages.
- `languages: string[]`
  Refer to GeneralConfig.languages.
- `active_language: string`
  The currently active language.
- `language_data: Language[]`
  Loaded languages to be used by other pieces of code to display things in the correct language.
  All languages in the `active_languages`.list are loaded on startup.
- `language_switch_interval: number`
  Refer to GeneralConfig.language_switch_interval.
- `date_format: string`
  Refer to GeneralConfig.date_format.
- `themes: ({ theme: string } & (TimeRestriction | {}))[]`
  Refer to GeneralConfig.themes.
- `active_theme: string`
  Refer to GeneralConfig.active_theme.

Methods

- `constructor(): UIStore`
  Initialize the UIStore. This loads the general config file and all needed translation files.
- `changeTheme(language: string): void`
  Changes the current theme.

  Note: This is called internally when a theme change that has been scheduled using TimeRestriction's for themes is to be executed.
- `changeLanguage(language: string): void`
  Changes the current language.
  If the language needs to be loaded it is loaded and the change is applied afterwards.

  Notes: This is called internally when a language change that has been scheduled using GeneralConfig.language_switch_interval is executed.

- `getLanguageByName(name: string): Language`
  This retrieves a language by its name. This is used by the Internationalization system.

3.4 Plugins

3.4.1 Cafeteria plugin

CafeteriaWidgetProps

```
type CafeteriaWidgetProps = {
  cafeteria_id?: string,
  day_offset?: number,
  filter?: string[]
}
```

Fields

- `cafeteria_id?: number`
  Id of the cafeteria to display, a default can be set in the cafeteria widget config file.

- `day_offset?: number`
  Offset from the current day for which to show the menu.

  Default value: `0`

- `filter?: string[]`
  A list of filters to apply to the meals being shown. A meal is only displayed if at least one of the strings in the filter array is contained in the list of classifiers of the meal.

  Default value: `[]`

CafeteriaWidget

React Component for rendering the cafeteria widget.

```
class CafeteriaWidget extends Component<
  CafeteriaWidgetProps, any
> {
  readonly props: CafeteriaWidgetProps
  state: any
  render(): JSX.Element
}
```

Fields

- `readonly props: CafeteriaWidgetProps`
  Properties passed to a React Component are called `props` and are read-only.

  Note: `props` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.
  Default value: Refer to CafeteriaWidgetProps.

- `state: any`
  Internal state of a react Component is just called `state`. Modifying it directly is forbidden as it breaks reacts update mechanisms, use `setState` (inherited from `Component`) instead.

  Note: `state` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.

Methods

- `render(): JSX.Element`
  Method called by react to render the component.

CafeteriaDish

```
type CafeteriaDish = {
  id: number
  mealName: string,
  price: number,
  classifiers: string[],
  additives: number[]
}
```

Fields

- `id: number`
  Id identifying the meal / dish.

- `mealName: string`
  A name for the meal / dish.

- `price: number`
  Price in Euros.

- `servedOn: number`
  Date (unix timestamp) the meal is served.

- `classifiers: string[]`
  List of attributes the meal / dish has like vegan or vegetarian.

  Examples: `"VEG"` (for vegan), `"GEL"` (contains gelatine), `"S"` (contains pork)

- `additives: number[]`
  List of additives, use additivesLegend to look up what each value means.

CafeteriaLine

```
type CafeteriaLine = {
  id: string,
  lineName: string,
  dishes: CafeteriaDish[]
}
```

Fields

- `id: string`
  Id of the line.

- `lineName: string`
  Name of the line.

- `dishes: CafeteriaDish[]`
  List of dishes served at the line.

Cafeteria

```
type Cafeteria = {
  id: string,
  name: string,
  timeOfLastUpdate: string,
  lines: CafeteriaLine[],
  openingHours: {
    id: string,
    name: string,
    openingTime: number,
    closingTime: number
  },
  additivesLegend: { [name: string]: string }
}
```

Fields

- `id: string`
  Id of the cafeteria; unique.

  Example: `"adenauerring"`

- `name: string`
  Name of the cafeteria, doesn't need to be unique.

  Example: `"Mensa am Adenauerring"`

- `date: string`
  Time of last update as unix timestamp.

- `lines: CafeteriaLine[]`
  Lines of the cafeteria.

- `openingHours: { ... }`

  - `id: number`
    Internal database id.
  - `name: string`
    Name describing the reason why the cafeteria is open or what line is open. This can best be parsed using a lookup table as the text doesn't follow any real rules or format otherwise.

    Example: `"Mittagessen"`, `"[kœri]werk"`
  - `openingTime: number`
    Unix timestamp of opening time.
  - `closingTime: number`
    Unix timestamp of closing time.

- `additivesLegend: { [name: string]: string }`
  Mapping from integers (encoded as strings) to explanations what each additive means.

CafeteriaAPIData

```
type CafeteriaAPIData = Cafeteria[]
```

The API data for the cafeteria consists of an array of Cafeterias.

CafeteriaConfig

```
type CafeteriaConfig = {
  cafeteria_ids: string[] | undefined,
  filter: string[] | undefined
}
```

Fields

- `cafeteria_ids: string[] | undefined`
  List of cafeteria ids to display dishes from.

  Note: The value undefined is used to represent choosing the default value; this is only important internally.
  Default value: `undefined`

- `filter: string[] | undefined`
  A list of filters to apply to the meals being shown. A meal is only shown if at least one of the strings in the filter array is contained in the list of classifiers of the meal.

  By default every meal is shown.

Note: The value undefined is used to represent choosing the default value; this is only important internally.
Default value: `undefined`

CafeteriaStore

```
class CafeteriaStore implements Store<[CafeteriaAPIData], CafeteriaConfig> {

  cafeterias: Cafeteria[]
  menu: { [name: string]: Cafeteria[] }

  config: CafeteriaConfig

  injectData(data: CafeteriaAPIData): void
  injectConfig(config: Partial<CafeteriaConfig>): void
}
```

Fields

- `cafeterias: Cafeteria[]`
  Cafeterias.

- `menu: { [name: string]: Cafeteria[] }`
  Cafeteria menu grouped by day, indexed using ISO 8601 date string.

- `config: CafeteriaConfig`
  Config file. Refer to CafeteriaConfig

Methods

- `injectData(data: CafeteriaAPIData): void`
  Used to inject retrieved API data into the CafeteriaStore.

- `injectConfig(config: Partial<CafeteriaConfig>): void`
  Used to inject a loaded and parsed config file into the CafeteriaStore.

3.4.2 Calendar plugin

CalendarWidgetProps

```
type CalendarWidgetProps = {
  calendar_ids?: string[],
  multiple_days?: boolean,
  display_allday?: boolean,
  overwrite_color?: string,
  detailed_events?: boolean
}
```

Fields

- `calendar_ids?: string[]`
  A list of calendars to display.

  Default value: All available calendars

- `multiple_days?: boolean`
  Wether to display multiple days at once if the screen realestate allows it.

  Default value: `true`

- `display_allday?: boolean`
  Wether to display all day events at all.

  Default value: `true`

- `overwrite_color?: string`
  Set to a css color to overwrite the color of the calendar events. Don't set to use the color provided by the calendar.

  Default value: `undefined`

- `detailed_events?: boolean`
  Wether to display detailed event information (location, ...).

  Default value: `true`

CalendarWidget

```
class CalendarWidget extends Component<
  CalendarWidgetProps, any
> {
  readonly props: CalendarWidgetProps
  state: any
  render(): JSX.Element
}
```

Fields

- `readonly props:  CalendarWidgetProps`
  Properties passed to a React Component are called `props` and are read-only.

  Note: `props` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.
  Default value: Refer to CalendarWidgetProps.

- `state:  any`
  Internal state of a react Component is just called `state`. Modifying it directly is forbidden as it breaks reacts update mechanisms, use `setState` (inherited from `Component`) instead.

  Note: `state` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.

Methods

- `render():  JSX.Element`
  Method called by react to render the component.

CalendarEvent

```
type CalendarEvent = {
  id: number,
  uid: string,
  name: string,
  startTime: number,
  endTime: number,
  location: string,
  isAllDay: boolean
}
```

Fields

- `id: number`
  Internal database id.

- `uid: string`
  Id from iCal, may be a UUID, may be something else entirely.

- `name: string`
  Message / name of the event.

  Example: `"Meeting with James"`

- `startTime: number`
  Starting time of the event as a unix timestamp.

- `endTime: number`
  End time of the event as a unix timestamp

- `location: string`
  String describing the location the event takes place at.

  Example: `"Office 3"`

- `isAllDay: boolean`
  Boolean flag for indicating wether an event is the whole day long or not.

  Note: If the boolean flag is set the startTime and endTime values are interpreted as being date values only. Time information is disregarded.

TransformedCalendarEvent

```
type TransformedCalendarEvent = CalendarEvent & {
  color: string,
  calendar: number,
  partOf: number[],
  splitStartTime: number,
  splitEndTime: number,
}
```

Fields

- `color: string`
  The color the calendar event should be displayed in.

- `calendar: number`
  The id of the calendar which this event is a part of.

- `partOf`
  Events which take place over multiple days are split up into multiple pieces internally; this keeps track of the parts.

- `splitStartTime`
  Start time of the split; this is used internally for showing the event at the correct place. The normal startTime value is not overwritten as this might be displayed alongside the event.

- `splitEndTime`
  End time of the split; this is used internally for showing the event at the correct place. The normal endTime value is not overwritten as this might be displayed alongside the event.

CalendarConfig

```
type CalendarConfig = {
  calendar_ids: string[] | undefined,
  multiple_days: boolean,
  display_allday: boolean,
  overwrite_color: string | undefined,
  detailed_events: boolean
}
```

Fields

- `calendar_ids: string[] | undefined`
  A list of calendars to display. By default this uses all available calendars.

Note: The value undefined is used to represent choosing the default value; this is only important internally.
Default value: `undefined`

- `multiple_days: boolean`
  Wether to display multiple days at once if the screen realestate allows it.

  Default value: `true`

- `display_allday: boolean`
  Wether to display all day events at all.

  Default value: `true`

- `overwrite_color: string | undefined`
  Set this to a css color to overwrite the color of the calendar events. Don't set to use the color provided by the calendar.

  Note: The value undefined is used to represent choosing the default value; this is only important internally.
  Default value: `undefined`

- `detailed_events: boolean`
  Wether to display detailed event information (location, ...).

  Default value: `true`

CalendarAPIData

```
type Calendar = {
  id: number,
  timeOfLastUpdate: number,
  name: string,
  color: string,
  events: CalendarEvent[]
}
```

Fields

- `id: number`
  Id of the calendar.

- `timeOfLastUpdate: number`
  Time the calendar was last updated (unix timestamp).

- `name: string`
  Name of the calendar.

- `color: string`
  A preferred color for the calendar, may be used to make events more distinguishable.

  Note: Must be a valid CSS color (named color, hex, rgb, rgba, hsl, hsla, ...).

- `events: CalendarEvent[]`
  List of events

```
type CalendarAPIData = Calendar[]
```

The API data for the calendar widget contains a list of calendars.

CalendarStore

```
class CalendarStore implements Store<[CalendarAPIData], CalendarConfig> {

  calendars: Calendar[]
  events: TransformedCalendarEvent[]

  config: CalendarConfig

  injectData(data: CalendarAPIData): void
  injectConfig(config: Partial<CalendarConfig>): void
}
```

Fields

- `calendars: Calendar[]`
  All calendars

  Default value: `[]`

- `events: TransformedCalendarEvent[]`
  All calendar events (with some extra values (compare TransformedCalendarEvent with CalendarEvent)).

  Default value: `[]`

- `config: CalendarConfig`
  Config file. Refer to CalendarConfig.

Methods

- `injectData(data: CalendarAPIData): void`
  Inject data retrieved from the API into the CalendarStore.

- `injectConfig(config: Partial<CalendarConfig>): void`
  Used to inject a loaded and parsed config file into the CalendarStore.

3.4.3 Clock plugin

ClockWidgetProps

```
type ClockWidgetProps = {
  format?: '12h' | '24h',
  timezone?: string,
  always_display_timezone_name?: boolean,
  display_seconds?: boolean
}
```

Fields

- `format?: '12h' | '24h'`
  12 hour or 24 hour format.

  Default value: '24h'

- `timezone?: string`
  Timezone to use.

  Default value: default_timezone configured in the general config

- `always_display_timezone_name?: boolean`
  Wether to always display the timezone name.

  Tri-state logic:

  - if set to `true` it always displays the timezone,
  - if set to `false` it never displays the timezone,
  - if not set at all (`undefined`) it shows the timezone name when the widget deems it necessary.

  Default value: `undefined`

- `display_seconds?: boolean`
  Wether to display seconds or not.

  Default value: `false`

ClockWidget

React Component for rendering the clock widget.

```
class ClockWidget extends Component<
  ClockWidgetProps, any
> {
  readonly props: ClockWidgetProps
  state: any
  render(): JSX.Element
}
```

Fields

- `readonly props: ClockWidgetProps`
  Properties passed to a React Component are called `props` and are read-only.

  Note: `props` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.
  Default value: Refer to ClockWidgetProps.

- `state: any`
  Internal state of a react Component is just called `state`. Modifying it directly is

forbidden as it breaks reacts update mechanisms, use `setState` (inherited from `Component`) instead.

Note: `state` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.

Methods

- `render(): JSX.Element`
  Method called by react to render the component.

ClockConfig

```
type ClockConfig = {
  format: '24h' | '12h',
  always_display_timezone_name: boolean,
  display_seconds: boolean
}
```

Fields

- `format: '24h' | '12h'`
  24 hour or 24 hour format.

  Default value: `'24h'`

- `always_display_timezone_name: boolean | undefined`
  Wether to always display the timezone name.

  Tri-state logic:

  - if set to `true` it always displays the timezone,
  - if set to `false` it never displays the timezone,
  - if not set at all (`undefined`) it shows the timezone name when the widget deems it necessary.

  Default value: `undefined`

- `display_seconds: boolean`
  Wether to display seconds or not.

Default value: false

ClockStore

Store for clock widget. Handles loading clock config file, parsing it as well as managin all the data stored in it afterwards.

```
ClockStore implements Store<[], ClockConfig> {

  format: '24h' | '12h'
  always_display_timezone_name: boolean | undefined
  display_seconds: boolean

  injectData(): void
  injectConfig(config: Partial<ClockConfig>): void
}
```

Fields

- `format: '24h' | '12h'`
  Refer to ClockConfig.format.

- `always_display_timezone_name: boolean | undefined`
  Refer to ClockConfig.always_display_timezone_name.

- `display_seconds: boolean`
  Refer to ClockConfig.display_seconds.

Methods

- `injectData(): void`
  Inject data retrieved from the API into the store. Since the clock doesn't get any data from the API no argument is given.

- `injectConfig(config: Parial<ClockConfig>): void`
  Used to inject a loaded and parsed config file into the ClockStore.

3.4.4 Publication plugin

PublicationWidget

```
class PublicationWidget extends Component<{}, any> {
  readonly props: PublicationWidgetProps
  state: any
  render(): JSX.Element
}
```

Fields

- `readonly props: PublicationWidgetProps`

- `state: any`
  Internal state of a react Component is just called `state`. Modifying it directly is forbidden as it breaks reacts update mechanisms, use `setState` (inherited from `Component`) instead.

  Note: `state` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.

Methods

- `render(): JSX.Element`
  Method called by react to render the component.

PublicationAPIData

```
PublicationAPIData = {
  id: number,
  title: string,
  authors: string[],
  publicationDate: string,
  description: string | null,
  publisher: string | null,
  publishLocation: string | null
}[]
```

Fields

- `id: number`
  Id of the publication

- `title: string`
  Title of the publication

- `authors: string[]`
  List of authors of the publication. Author name includes academic titles.

- `publicationDate: string`
  Date the publication was published, this may be unspecific (like "early 2020", "dec 2020") so it is a string which can theoretically contain arbitrary text.

- `description: string | null`
  A description of the publication if available.

- `publisher: string | null`
  A possible publisher for the publication.

- `publishLocation: string | null`
  A possible location where the publication was published like a scientific journal or magazine.

PublicationConfig

```
type PublicationConfig = {
  publication_sources: string[] | undefined
}
```

Fields

- `publication_sources: string[] | undefined` List of publication sources to show. By default all publication sources are shown.

  Note: The value `undefined` is used to represent choosing the default value; this is only important internally.
  Default value: `undefined`

PublicationStore

```
class PublicationStore implements Store<PublicationAPIData, PublicationConfig> {

  publications: PublicationAPIData
  config: PublicationConfig

  injectData(data: PublicationAPIData): void
  injectConfig(config: Partial<PublicationConfig>): void
}
```

Fields

- `publications: PublicationAPIData`
  Publications.

  Default value: `[]`

- `config: PublicationConfig`
  Config file. Refer to PublicationConfig.

Methods

- `injectData(data: PublicationAPIData): void`
  Used to inject retrieved API data into the PublicationStore.

- `injectConfig(config: Partial<PublicationConfig>): void`
  Used to inject a loaded and parsed config file into the PublicationStore.

3.5 Components

3.5.1 Layout

React component which handles rendering the layout.

```
class Layout extends Component<{}, LayoutConfig & LayoutState> {
  state: LayoutConfig & LayoutState
  componentDidMount(): void
  render(): JSX.Element
}
```

Fields

- `state: LayoutConfig & LayoutState` Internal state of a react Component is just called `state`. Modifying it directly is forbidden as it breaks reacts update mechanisms, use `setState` (inherited from `Component`) instead.

  Note: `state` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.

Methods

- `componentDidMount(): void`
  React lifecycle hook. Called when the component is mounted.

- `render(): JSX.Element`
  Method called by react to render the component.

3.5.2 CarouselSlot

React component which renders a carousel cycling through its given components. Used by the layout system.

```
const CarouselSlot: (props: {
  slots: Slot[],
  time: number
}) => JSX.Element
```

Props

- `slots: Slot[]`
  An array of slots to cycle through.

- `time: number`
  Time between switching from one slot to the next.

3.5.3 VerticalSplitSlot

React component responsible for rendering a vertical split. Used by the layout system.

```
const VerticalSplit: (props: {
    top: Slot;
    bottom: Slot;
    ratio: number;
}) => JSX.Element
```

Props

- `top: Slot`
  The slot at the top.

- `bottom: Slot`
  The slot at the bottom.

- `ratio: number`
  Amount of space the first slot takes up compared to the overall space available.

3.5.4 HorizontalSplitSlot

React component responsible for rendering a horizontal split. Used by the layout system.

```
const HorizontalSplit: (props: {
    left: Slot;
    right: Slot;
    ratio: number;
}) => JSX.Element
```

Props

- `left: Slot`
  The slot on the left.

- `right: Slot`
  The slot on the right.

- `ratio: number`
  Amount of space the first slot takes up compared to the overall space available.

3.5.5 WidgetSlot

React component responsible for rendering a single widget in a slot. Used by the layout system.

```
const WidgetSlot: ({ component, props }: {
    component: string;
    props: any;
}) => JSX.Element
```

Props

- `component: string`
  Component referenced by its id (refer to Plugin). If the component is part of a pre-bundled plugin it will just be accessed like any other code. If the component is not part of a pre-bundled plugin the plugin will have to fetched. This process is initiated here.

- `props: any`
  Properties (abbreviated to *props* in the react world) to pass to the component.

3.5.6 EmptySlot

react component which renders an empty slot. Used by the layout system.

Shows the background color selected by theme and nothing else.

```
const EmptySlot: () => JSX.Element
```

3.5.7 Announcement

React component which renders an announcement.

Used by the layout system.

```
const AnnouncementComp: (props: { announcement: Announcement }) => JSX.Element
```

- `announcement: Announcement`
  The announcement to render.

3.6 Program entry point and Internationalization

3.6.1 App

Entrypoint to the app.

Handles injecting all global mobx stores into the react context.

```
class App extends Component {
  private timer?: number
  private configs_and_plugins: Promise<[any[], Plugin<any, any>[]]>

  state: { loaded: boolean, error: Error | null, announcements: Announcement[] }

  constructor(): App

  private refresh(configs: any[], plugins: Plugin<any, any>[]): void

  componentDidMount(): void
  componentWillUnmount(): void
  render(): JSX.Element
}
```

Fields

- `private timer?: number`
  Internal timer for making update api calls.

- `private configs_and_plugins: Promise<[any[], Plugin<any, any>[]]>`
  Promise resolving to a tuple of config files and plugins.

- `state: { ... }`
  Internal state of the component.

Constructor

Loads the plugins config file and using this proceeds to load all non-bundled plugins as well as all config files for bundled and non-bundled plugins.

Methods

- `private refresh(configs: any[], plugins: Plugin<any, any>[]): void`
  Method called using the timer to refresh the data by making an update api call.

  - `configs: any[]`: array of config files for each plugins.
  - `plugins: Plugin<any, any>[]`: array of plugins.

- `componentDidMount(): void`
  React lifecycle hook. Called when the component is mounted. Initiates repeated data updates using the `timer` and the `refresh` method.

- `componentWillUnmount(): void`
  React lifecycle hook. Called when the component will be unmounted.

- `render(): JSX.Element`
  Method called by react to render the component.

3.6.2 Internationalization

IntlProps

```
type IntlProps = {
  language?: string,
  children: string
} | {
  language?: string,
  word: string
}
```

- `language?: string`
  The language to be used. If not specified the current language is retrieved from the UIStore.

- `children: string`
  React child elemnet which specifies which word to render (required if word is not specified).

- `word: string`
  parameter which specifies which word to render (required if children is not specified).

Intl

React component which translates a word to a specified language and renders it.

```
const Intl = (props: IntlProps) => ReactElement<IntlProps, string>
```

3.7 API Interactions

3.7.1 FileLoader

Loads a file from a network location, validates the file and parses it to a specified format.

```
const FileLoader: <FileType>(
  filename: string,
  base_url: string | undefined,
  parse: (text: string) => any,
  verify?: (file: any) => boolean,
  transform?: (file: any) => FileType
) => Promise<FileType>
```

`FileType` is the wanted type after parsing and transforming.

Arguments

- `filename: string`
  Relative filename

Default value:

- `base_url: string`
  Base URL to combine with the filename, defaults to

  Default value: `window.BASE_URL`

- `parse: (text: string) => any`
  Parses the file to a wanted data format (e.g. JSON, YAML, ... to JavaScript object)

- `verify: (file: any) => boolean`
  Verifies the correctness of the data (correct structure)

  Default value: `() => true`

- `transform: (file: any) => FileType`
  Does a last transformation with the data after it has been verified as correct

  Default value: `(file: any) => file`

3.7.2 YAMLFileLoader

Specialized form of FileLoader for working with YAML files.

```
const YAMLFileLoader: <FileType>(
  filename: string,
  base_url?: string,
  verify?: (file: any) => boolean,
  transform?: (file: any) => FileType
) => Promise<FileType>
```

`FileType` is the wanted type after parsing and transforming.

- `filename: string`
  Relative filename

  Default value:

- `base_url: string`
  Base URL to combine with the filename, defaults to

  Default value: `window.BASE_URL`

- `verify: (file: any) => boolean`
  Verifies the correctness of the data (correct structure)

  Default value: `() => true`

- `transform: (file: any) => FileType`
  Does a last transformation with the data after it has been verified as correct

  Default value: `(file: any) => file`

3.7.3 JSONFileLoader

Specialized form of FileLoader for working with JSON files.

```
const JSONFileLoader: <FileType>(
  filename: string,
  base_url?: string,
  verify?: (file: any) => boolean,
  transform?: (file: any) => FileType
) => Promise<FileType>
```

`FileType` is the wanted type after parsing and transforming.

- `filename: string`
  Relative filename

  Default value:

- `base_url: string`
  Base URL to combine with the filename, defaults to

  Default value: `window.BASE_URL`

- `verify: (file: any) => boolean`
  Verifies the correctness of the data (correct structure)

Default value: `() => true`

- `transform: (file: any) => FileType`
  Does a last transformation with the data after it has been verified as correct

  Default value: `(file: any) => file`

3.7.4 DependencyInjectionTarget

Wraps an instance of `T` in a function which injects dependencies into it.

Dynamically loaded plugins require access to certain dependencies.

Note: This is done through dependency injection and this type describes a target for injection.

```
type DependencyInjectionTarget<T> =
  (dependencies: { [name: string]: any }) => T
```

Arguments

- `dependencies: { [name: string]: any }`
  Mapping from strings to dependencies. Since dependencies do not have a similar signature this cannot be further specified.

Return value: instance of `T` which now has dependencies in its scope.

3.7.5 PluginLoader

Specialized form of FileLoader for loading plugins.

```
const PluginLoader: (filename: string)
  => Promise<DependencyInjectionTarget<Plugin<any>>>
```

Arguments

- `filename: string`
  Id of the plugin, does not include anything else; no `.js`, no path.

Return value: Promise resolving to `DependencyInjectionTarget<Plugin<any»>` if the plugin could be retrieved and parsed.

3.7.6 API

```
class API {
  get_updates(): Promise<{
    data: { [k: string]: any; }
    last_update: null
  }>
  log_error(msg: string, urgency: ErrorLogUrgency, timestamp: number): Promise<void>
}
```

Methods

- `get_updates(): Promise<...>`
  Retrieves updates from the api using the Update.

  Returns a Promise resolving to the data and the timeOfLastRelevantUpdate (as `last_update`).

- `log_error(msg: string, urgency: ErrorLogUrgency, timestamp: number): Promise<void>`
  Logs an error.

  - `msg: string`: The message.
  - `urgency: ErrorLogUrgency`: The urgency.
  - `timestamp: number`: Date the error / warning occured.

4 Class descriptions - Backend

4.1 Smart TV System

4.1.1 Program entry point

Backend

The main entrypoint to the Backend server.

```
class Backend {
  public long final STARTUP
  public String final VERSION
  private ConfigurableApplicationContext context

  public static void main(String[] args)
  public static void shutDown(boolean restart)
}
```

Fields

- `long STARTUP`
  The startup time of this server.

- `String VERSION`
  The current version of the backend.

- `ConfigurableApplicationContext context`
  The context to manage the application.

Methods

- `static void main(String[] args)`
  Starts the server.

  Parameter
  `String[] args`: The commandline arguments.

- `static void shutDown(boolean restart)`
  Gracefully stops the server and optionally restarts it.

  Parameter
  `boolean restart`: If true restarts the server.
  IF false just shuts the server down.

4.1.2 Configuration

KeycloakSecurityConfig

Configures the security policies.

```
class KeycloakSecurityConfig extends KeycloakWebSecurityConfigurerAdapter {
  protected void configure(HttpSecurity http) throws Exception
  public void configureGlobal(AuthenticationManagerBuilder auth)
  protected SessionAuthenticationStrategy sessionAuthenticationStrategy()
  public KeycloakConfigResolver KeycloakConfigResolver()
}
```

Methods

- `protected void configure(HttpSecurity http) throws Exception`
  Configures how http requests are handheld.

  Parameter
  `HttpSecurity http`: The instance of the current HttpSecurity.

  Exception: Exception: Thrown if the configuration fails.

- `public void configureGlobal(AuthenticationManagerBuilder auth)`
  Registers the KeycloakAuthenticationProvider with the authentication manager.

Parameter
`AuthenticationManagerBuilder auth`: The Authentication manager builder where the keycloak authentication provider is set.

- `protected SessionAuthenticationStrategy sessionAuthenticationStrategy()`
  Defines the session authentication strategy.

  Return value: The new authentication strategy.

- `public KeycloakConfigResolver KeycloakConfigResolver()`
  Configures the configuration resolver to uses this configuration class.

  Return value: A configuration resolver that uses this class for configuration.

AppConfiguration

Configures various aspects of the application.

```
class AppConfiguration implements WebMvcConfigurer {
  private final String BASE_PATH

  public void addResourceHandlers(ResourceHandlerRegistry registry)
}
```

Fields

- `String BASE_PATH`
  The base path where static resources are located.

Methods

- `void addResourceHandlers(ResourceHandlerRegistry registry)`
  Add handlers to serve static resources such as config and language files.

  Parameter
  `ResourceHandlerRegistry registry`: The active resource registry.

4.1.3 Controller

AggregationController

Provides the api endpoints to communicate with the aggregation service.

```
class AggregationController extends RequestController {
  private AggregationService aggregationService
  private ErrorLoggingService logger

  public ResponseEntity<RequestWrapper> aggregateServices()
}
```

Fields

- `AggregationService aggregationService`
  Used to interface with the aggregation system.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `ResponseEntity<RequestWrapper> aggregateServices()`
  Gets a timestamp of the last time a core or a widget file got updated and a list of elements, of which each element represents a service and its most recent data.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints. See Update.

ConfigController

Provides the api endpoints to write to config files.

```
class ConfigController extends RequestController {
  private ConfigurationFileService configurationFileService
  private ErrorLoggingService logger

  public ResponseEntity<RequestWrapper> writeDataConfig
    (String name, String newContent)
  public ResponseEntity<RequestWrapper> writeCoreConfig
    (String name, String newContent)
  public ResponseEntity<RequestWrapper> writeWidgetConfig
  (String name, String newContent)
  private ResponseEntity<RequestWrapper> writeFile
    (String name, String newContent)
}
```

Fields

- `ConfigurationFileService configurationFileService`
  Used to access the config files.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `ResponseEntity<RequestWrapper> writeDataConfig(String name, String newContent)`
  Sets the data config file.

  Parameter
  `String name`: The name of the file that should be set.
  `String newContent`: The data to which the content of the file should be set.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Configuration

- `ResponseEntity<RequestWrapper> writeCoreConfig(String name, String newContent)`
  Sets the core config file.

  Parameter
  `String name`: The name of the file that should be set.
  `String newContent`: The data to which the content of the file should be set.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints See Configuration

- `ResponseEntity<RequestWrapper> writeWidgetConfig(String name, String newContent)`
  Sets the widget config file.

  Parameter
  `String name`: The name of the file that should be set.
  `String newContent`: The data to which the content of the file should be set.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints See Configuration

- `ResponseEntity<RequestWrapper> writeFile(String name, String newContent)`
  Writs data to a file.
  Overwrites the data in the file.

  Parameter
  `String name`: The name of the file that should written to.
  `String newContent`: The data to which the content of the file should be set.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints See Configuration

LoggingController

Provides the api endpoints to log errors and retrieve logs.

```
class LoggingController extends RequestController {
  private ErrorLoggingService errorLoggingService

  public ResponseEntity<RequestWrapper> appendToLog(LogEntry logEntry)
  public ResponseEntity<?> readLog()
}
```

Fields

- `ErrorLoggingService errorLoggingService`
  Used to interface with the logging system.

Methods

- `ResponseEntity<RequestWrapper> appendToLog(LogEntry logEntry)`
  Appends a log entry to the log file.

  Parameter
  `LogEntry logEntry`: The log entry that should be appended to the log file.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints. See Error logging

- `ResponseEntity<?> readLog()`
  Returns the content of the log file.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints. See Error logging

MiscellaneousController

Provides the api endpoints to retrieve information like uptime or server version.

```
class MiscellaneousController {
  private ErrorLoggingService logger

  public long getUptime()
  public String getVersion()
}
```

Fields

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `long getUptime()`
  Gets the uptime of the server.

  Return value: The uptime of the server. See Miscellaneous

- `String getVersion()`
  Gets the current version number of the server.

  Return value: The version number of the server. See Miscellaneous

RequestController

Defines a basic Controller and offers the functionality to easily create a new ResponseEntity.

```
abstract class RequestController {
  protected ResponseEntity<RequestWrapper> createResponse
  (String message, boolean success, Object data, HttpStatus status)
}
```

Methods

- `ResponseEntity<RequestWrapper> createResponse(String message, boolean success,`
  `Object data, HttpStatus status)`
  Creates a new ResponseEntity that conforms to the api endpoint specification.

  Parameter
  `String message`: The status message of the response.
  `boolean success`: True if the Request was successful, if not false.
  `Object data`: The data of the response.
  `HttpStatus status`: The HTTP status code.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints

4.1.4 Data

AggregationEntityWrapper

Wraps aggregated data so it can be referenced by a name.

```
class AggregationEntityWrapper {
  private String name
  private List<Object> data

  public AggregationEntityWrapper(String name, List<Object> data)

  public String getName()
  public List<Object> getData()
}
```

Fields

- `String name`
  The name of the service that generated the data.

- `List<Object> data`
  The data that is aggregated.

Methods

- `AggregationEntityWrapper(String name, List<Object> data)`
  Initializes a new instance of an AggregationEntityWrapper.

  Parameter
  `String name`: The name of the service that generated the data.
  `Object data`: The data that is aggregated.

- `String getName()`
  Gets tne name of the corresponding service.

  Return Value: The name of the service that generated the data.

- `List<Object> getData()`
  Gets the data that is aggregated.

  Return Value: The data that is aggregated.

AggregationWrapper

Wraps all the aggregated data and adds at timestamp for the last time a frontend relevant config file got updated.

```
class AggregationWrapper {
  private Date timeOfLastRelevantUpdate
  private List<AggregationEntityWrapper> aggregatedData

  public AggregationWrapper
    (Date timeOfLastRelevantUpdate, List<AggregationEntityWrapper> aggregatedData)

  public Date getTimeOfLastRelevantUpdate()
  public List<AggregationEntityWrapper> getAggregatedData()
}
```

Fields

- `Date timeOfLastRelevantUpdate`
  A timestamp describing the last time a frontend relevant config file got updated.

- `List<AggregationEntityWrapper> aggregatedData`
  Contains all aggregated data.

Methods

- `AggregationWrapper(Date timeOfLastRelevantUpdate, List<AggregationEntityWrapper> aggregatedData)`
  Initializes a new instance of an AggregationEntityWrapper.

  Parameter
  `Date timeOfLastRelevantUpdate`: A timestamp describing the last time a frontend relevant config file got updated.
  `List<AggregationEntityWrapper> aggregatedData`: Contains all aggregated data.

- `Date getTimeOfLastRelevantUpdate()`
  Gets a timestamp describing the last time a frontend relevant config file got updated.

  Return Value: A timestamp describing the last time a frontend relevant config file got updated.

- `List<AggregationEntityWrapper> getAggregatedData()`
  Gets all aggregated data.

  Return Value: All aggregated data.

LogEntry

A datatype describing a log entry.

```
class LogEntry {
  private String message
  private Date timestamp
  private Urgency urgency

  public LogEntry()
  public LogEntry(String message, Date timestamp, Urgency urgency)

  public String getMessage()
  public Date getTimestamp()
  public Urgency getUrgency()
}
```

Fields

- `String message`
  An error message that describes the error.

- `Datetimestamp`
  The time when the error occurred.

- `Urgency urgency`
  The urgency of the error.

Methods

- `LogEntry()`
  Initializes a new empty instance of an LogEntry.

- `LogEntry(String message, Date timestamp, Urgency urgency)`
  Initializes a new instance of an LogEntry.

Parameter
`String message`: A message that describes the error.
`Date timestamp`: The time when the error occurred.
`Urgency urgency`: The urgency of the error.

- `String getMessage()`
  Gets a message that describes this error.

  Return value: The message that describes this error.

- `Date getTimestamp()`
  Gets the time when this error occurred.

  Return value: The time when this error occurred.

- `Urgency getUrgency()`
  Gets the urgency of this error.

  Return value: The urgency of this error.

RequestWrapper

Wraps response data so it conforms to the api endpoint specification.

```
class RequestWrapper {
  private String message
  private String status
  private Object data

  public RequestWrapper(String message, boolean success, Object response)

  public String getMessage()
  public boolean isSuccess()
  public Object getData()
}
```

Fields

- `String message`
  The status message of the response.

- `String status`
  The status of the request.
  Example: `"success"` or `"failure"`

- `Object data`
  The data of the response.

Methods

- `RequestWrapper(String message, boolean success, Object response)`
  Initializes a new instance of a ResponseWrapper.

  Parameter
  `String message`: The status message of the response.
  `boolean success`: True if the Request was successful, if not false.
  `Object response`: The data of the response.

- `String getMessage()`
  Gets the status message of this response.

  Return value: The status message of this response.

- `boolean isSuccess()`
  Gets the status of the request.
  Example: `"success"` or `"failure"`

  Return value: The status of the request.

- `Object getData()`
  Gets the data of this response.

  Return value: The data of this response.

Urgency

Defines different urgencies for errors.

```
enum Urgency {
  LOW("low")
  MEDIUM("medium")
  HIGH("high")
  String final label

  public Urgency(String label)
  public String toString()
  public Urgency fromLabel(String label)
}
```

Values

- `LOW("low")`
  Low urgency.

- `MEDIUM("medium")`
  Medium urgency.

- `HIGH("high")`
  High urgency.

Fields

- `String label`
  A string representation of the urgency.

Methods

- `Urgency(String label)`
  Initializes a new instance of an Urgency.

  Parameter
  `String label`: A string representation of the urgency.

- `String toString()`
  Returns the string representation of the urgency.

  Return value: The string representation of the urgency.

- `Urgency fromLabel(String label)`
  Gets the urgency enum type with the marching label.

  Parameter
  `String label`: The label of the enum type.

Return value: The urgency with matching label.

4.1.5 Exception

EntityNotFoundException

Thrown if an entity could not be found.

```
class EntityNotFoundException extends Exception {

}
```

4.1.6 Repository

LocalRepository

Defines a repository stored in memory.

```
class LocalRepository<Key, Entity> implements CrudRepository<Entry, Key> {
  private final HashMap<Key, Entry> repository

  public LocalRepository()

  public <S extends Entry> S save(S entity)
  public <S extends Entry> Iterable<S> saveAll(Iterable<S> entities)
  public Optional<Entry> findById(Key key)
  public boolean existsById(Key key)
  public Iterable<Entry> findAll()
  public Iterable<Entry> findAllById(Iterable<Key> keys)
  public long count()
  public void deleteById(Key key)
  public void delete(Entry entity)
  public void deleteAll(Iterable<? extends Entry> entities)
  public void deleteAll()
}
```

Generics

- `<Key>`
  Defines the key of the repository.

- `<Entry>`
  Defines the entries in the repository.

Fields

- `HashMap<Key, Entry> repository`
  Contains the data of the local repository.

Methods

- `LocalRepository()`
  Initializes a new empty LocalRepository.

- `<S extends Entry> S save(S entity)`
  Saves a given entity. Use the returned instance for further operations as the save operation might have changed the entity instance completely.

  Parameter
  `S entity`: Must not be `null`.

  Return value
  The saved entity; will never be `null`.

  Exception: IllegalArgumentException: In case the given `entity` is `null`.

- `<S extends Entry> Iterable<S> saveAll(Iterable<S> entities)`
  Saves all given entities.

  Parameter
  `Iterable<S> entities`: Must not be `null` nor must it contain `null`.

  Return value
  The saved entities; will never be `null`. The returned `Iterable` will have the same size as the `Iterable` passed as an argument.

  Exception: IllegalArgumentException: In case the given @link Iterable entities or one of its entities is `null`.

- `Optional<Entry> findById(Key key)`
  Retrieves an entity by its id.

Parameter
`Key key`: Must not be `null`.

Return value
The entity with the given id or `Optionalempty()` if none found.

Exception: IllegalArgumentException: If `id` is `null`.

- `boolean existsById(Key key)`
  Returns whether an entity with the given id exists.

  Parameter
  `Key key`: Must not be `null`.

  Return value
  `true` if an entity with the given id exists, `false` otherwise.

  Exception: IllegalArgumentException: If `id` is `null`.

- `Iterable<Entry> findAll()`
  Returns all instances of the type.

  Return value
  All entities.

- `Iterable<Entry> findAllById(Iterable<Key> keys)`
  Returns all instances of the type @code T with the given IDs.
  If some or all ids are not found, no entities are returned for these IDs.
  Note that the order of elements in the result is not guaranteed.

  Parameter
  `Iterable<Key> keys`: Must not be `null` nor contain any `null` values.

  Return value
  Guaranteed to be not `null`. The size can be equal or less than the number of given `ids`.

  Exception: IllegalArgumentException: In case the given @link Iterable ids or one of its items is `null`.

- `long count()`
  Returns the number of entities available.

  Return value

The number of entities.

- `void deleteById(Key key)`
  Deletes the entity with the given id.

  Parameter
  `Key key`: Must not be `null` nor contain any `null` values.

  Exception: IllegalArgumentException: In case the given `id` is `null`.

- `void delete(Entry entity)`
  Deletes a given entity.

  Parameter
  `Entry entity`: Must not be `null`.

  Exception: IllegalArgumentException: In case the given entity is `null`.

- `void deleteAll(Iterable<? extends Entry> entities)`
  Deletes the given entities.

  Parameter
  `Iterable<? extends Entry> entities`: Must not be `null`. Must not contain `null` elements.

  Exception: IllegalArgumentException: In case the given `entities` or one of its entities is `null`.

- `void deleteAll()`
  Deletes all entities managed by the repository.

4.1.7 Service

Aggregable

If a class implements this interface it will be aggregated by the AggregationService.

```
interface Aggregable {
  public List<?> aggregate()
  public String getPluginID()
}
```

Methods

- `List<?> aggregate()`
  Gets the data that should be aggregated.

  Return value: The data that should be aggregated.

- `String getPluginID()`
  Get the ID of this plugin.

  Return value: The ID of this plugin.

AggregationService

Aggregates the data of all classes implementing Aggregable.

```
class AggregationService {
  private final List<Aggregable> services
  private final ErrorLoggingService logger
  private ConfigurationFileService configService

  public AggregationService(List<Aggregable> services, ErrorLoggingService logger)

  public AggregationService(List<Aggregable> services)
  public ArrayList<AggregationWrapper> aggregateServices()
}
```

Fields

- `List<Aggregable> services`
  The list of services which will be aggregated.

- `ErrorLoggingService logger`
  Used to log errors.

- `ErrorLoggingService logger`
  Used to obtain the last time a frontend relevant config file got updated.

Methods

- `AggregationService(List<Aggregable> services, ErrorLoggingService logger)`
  Initializes a new instance of an AggregationService by providing a list of services to aggregate.

  Parameter
  `List<Aggregable> services`: The list of services that should be aggregated.
  `ErrorLoggingService logger`: Used to log errors.

- `ArrayList<AggregationWrapper> aggregateServices()`
  Aggregates all specified services.

  Return value
  The list contains one entry for every service.
  The entry contains a data and a name to refer to its origen.

ConfigurationFileService

Provides the functionality to get the content of a config file by its name.

```
class ConfigurationFileService {
  private final File CONFIG_DIRECTORY
  private final ArrayList<UpdateEntry> subscriber
  private final HashMap<String, Map<String, Object>> configFiles
  private Date lastUpdateToCoreOrWidget
  private final ErrorLoggingService logger

  public ConfigurationFileService(ErrorLogginService)

  private void cacheFile(String fileName) throws IOException
  public void subscribe(ArrayList<String> monitoredFiles, Updatable updatable)
  public void unsubscribe(Updatable updatable)
  private void updateSubscriber(String monitoring)
  public boolean writeConfigFile(String name, String content) throws IOException
  public Map<String, Object> readConfigFile(String name)
  public Date getLastUpdateToCoreOrWidget()
  private String readFileAsString(File file) throws IOException

  private class UpdateEntry {
    private ArrayList<String> monitoredFiles
    private final Updatable toUpdate
    private UpdateEntry(ArrayList<String> monitoredFiles, Updatable toUpdate)
  }
}
```

Fields

- `File CONFIG_DIRECTORY`
  The directory containing all config files.

- `ArrayList<UpdateEntry> subscriber`
  Contains all subscriber that should be updated if a config file changes.

- `HashMap<String, Map<String, Object» configFiles`
  Already accessed config files are cached.
  Maps a file name to a map containing the yml property name and corresponding value.

- `Date lastUpdateToCoreOrWidget`
  A timestamp of the last time a core or a widget config file got updated.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `CongigurationFileService()`
  Initializes a new instance of a ConfigurationFileService and reads all config files in CONFIG_DIRECTORY.

  Parameter
  `ErrorLoggingService logger` Used to log errors.

- `void cacheFile(String fileName) throws IOException`
  Caches the parsed version of config file.

  Parameter
  `String fileName`: The name of the config file.

  Exception: IOException: Throws if the file could not be accessed.

- `void subscribe(ArrayList<String> monitoredFiles, Updatable updatable)`
  Adds a new subscriber that will be updated when a monitored file changes.

  Parameter

`ArrayList<String> monitoredFiles`: The files that will be monitored.
`Updatable updatable`: The subscriber on which update will be called.

- `void unsubscribe(Updatable updatable)`
  Removes a subscriber so it won't be notified anymore.

  Parameter
  `Updatable updatable`: The subscriber that should not get notified anymore.

- `void updateSubscriber(String monitoring)`
  Updates all subscribers monitoring the specified file.

  Parameter
  `String monitoring`: The name of the file that has changed.

- `boolean writeConfigFile(String name, String content) throws IOException`
  Updates a config file on disk and nortify all subscribers.

  Parameter
  `String name`: The name of the file that should be updated.
  `String content`: The new content of the file.

  Return value: True if the file was changed successfully, false if the file could not be changed.

  Exception: IOException: Thrown if the write operation generates an exception.

- `Map<String, Object> readConfigFile(String name)`
  Gets a config file by it's name.

  Parameter
  `String name`: The name of the config file.

  Return value: A map, mapping YAML properties to the corresponding values. The key `"__config_file"` contains the whole file.

- `Date getLastUpdateToCoreOrWidget()`
  Gets a timestamp of the last time a core or widget config file got updated.

  Return value: A timestamp of the last time a core or widget config file got updated.

- `String readFileAsString(File file) throws IOException`
  Utility method to read files from disk.

Parameter
`File file`: The file that should be read.

Return value: The content of the file.

Exception: IOException: Throws if the read operation generates an exception.

UpdateEntry
A datatype describing an update entry.

Fields

- `ArrayList<String> monitoredFiles`
  If one of these files changes the subscriber should be updated.

- `toUpUpdatable date`
  The subscriber that should get updated.

Methods

- `UpdateEntry(ArrayList<String> monitoredFiles, Updatable toUpdate)`
  Initializes a new instance of an UpdateEntry.

  Parameter
  `ArrayList<String> monitoredFiles`: If one of these files changes the subscriber should be updated.
  `Updatable toUpdate`: The subscriber that should get updated.

ErrorLoggingService

Provides the functionality to log errors to the log file or read out the log file.

```
class ErrorLoggingService {
  private final File LOG_FILE_LOCATION
  public String getLog()
  public void addToLog(LogEntry logEntry) throws IOException
}
```

Fields

- `File LOG_FILE_LOCATION`
  The location of the log file.

Methods

- `String getLog()`
  Gets all error logs.

  Return value: A string containing all error logs.

- `void addToLog(LogEntry logEntry) throws IOException`
  Appends an error log to the log file.

  Parameter
  `LogEntry logEntry`: The log that should be appended.

  Exception: IOException: Thrown if the log file could not be updated.

JobSchedulerService

Provides the functionality to schedule tasks for periodically execution.

```
class JobSchedulerService {
   private ScheduledExecutorService scheduler
   private final ErrorLoggingService logger

   public JobSchedulerService(ErrorLoggingService logger)

   public void submitJob(Runnable job, long delay, long period, TimeUnit timeUnit)
   public void stopAllTasks()
   public void startAllTasks()
}
```

Fields

- `ScheduledExecutorService scheduler`
  Used to execute tasks periodically.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `JobSchedulerService(ErrorLoggingService logger)`
  Initializes a new instance of a JobSchedulerService.

  Parameter
  `ErrorLoggingService logger` Used to log errors.

- `void submitJob(Runnable job, long delay, long period, TimeUnit timeUnit)`

  Parameter
  `Runnable job`: The job that should be executed.
  `long delay`: The delay until the first execution.
  `long period`: The time between executions.
  `TimeUnit timeUnit`: The time unit used for "delay" and "period".

- `void stopAllTasks()`
  Stops all task after finishing the currently active task.

- `void stopAllTasks()`
  Starts the execution of all tasks.

Updatable

If a class implements this interface, it can be notified to update its properties when a configFile changes.

```
interface Updatable {
  public boolean update()
}
```

Methods

- `boolean update()`
  Updates all properties related to the config file.

  Return value
  True if all properties could be updated successfully without a restart.
  False if a restart of the server is required for at least one property.

4.2 Announcement plugin

4.2.1 Controller

AnnouncementController

Provides the api endpoints to communicate with the announcementDataPlugin.

```
class AnnouncementController extends RequestController {
  private AnnouncementService announcementService
  private ErrorLoggingService logger

  public ResponseEntity<RequestWrapper> findAll()
  public ResponseEntity<RequestWrapper> findById(long id)
  public ResponseEntity<RequestWrapper> create(Announcement announcement)
  public ResponseEntity<RequestWrapper> update(long id, Announcement announcement)
  public ResponseEntity<RequestWrapper>deleteById(long id)
}
```

Fields

- `AnnouncementService announcementService`
  Used to interface with the stored announcements.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `ResponseEntity<RequestWrapper> findAll()`
  Gets a list of all Announcements.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Announcements.

- `ResponseEntity<RequestWrapper> findById(long id)`
  Gets an announcement by its unique id.

Parameter
`long id`: The unique id of the announcement that should be fetched.

Return value: A HTTP response is returned containing

- status code,
- the JSON response object as shown in the API endpoints. See Announcements.

- `ResponseEntity<RequestWrapper> create(Announcement announcement)`
  Creates a new announcement.

  Parameter
  `Announcement announcement`: the announcement that should be added.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Announcements.

- `ResponseEntity<RequestWrapper> update(long id, Announcement announcement)`
  Updates an announcement specified by its id.

  Parameter
  `long id`: The id of the announcement that should be updated.
  `Announcement announcement`: The announcement with which the specified announcement should be replaced.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Announcements.

- `ResponseEntity<RequestWrapper> deleteById(long id)`
  Deletes an announcement specified by its id.

  Parameter
  `long id`: The id of the announcement that should be deleted.

  Return value: A HTTP response is returned containing

  - status code,

  – the JSON response object as shown in the API endpoints. See Announcements.

4.2.2 Data

Announcement

A datatype describing an announcement.

```
class Announcement {
  private Long id
  private String message
  private AnnouncementUrgency urgency
  private Date startTime
  private Date expireTime

  public Announcement()
  public Announcement
    (String message, AnnouncementUrgency urgency, Date startTime, Date endTime)

  public Long getId()
  public void setId(Long id)
  public String getMessage()
  public AnnouncementUrgency getUrgency()
  public Date getStartTime()
  public Date getEndTime()
}
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the announcement can be referenced.

- `String message`
  The message of the announcement.

- `int urgency`
  The urgency of the announcement.

- `Date startTime`
  The time at which the announcement should be displayed.

- `Date expireTime`
  The time at which the announcement should stop being displayed.

Methods

- `Announcement()`
  Initializes a new empty instance of an Announcement.

- `Announcement(String message, int urgency, Date startTime, Date endTime)`
  Initializes a new instance of an Announcement.

  Parameter
  `String message`: The message of the new announcement.
  `int urgency`: The urgency of the new announcement.
  `Date startTime`: The time at which the announcement should be displayed.
  `Date endTime`: The time at which the announcement should stop being displayed.

- `Long getId()`
  Gets the unique id of this announcement.

  Return value: The id of this announcement.

- `void setId(Long id)`
  Set the unique id of this announcement.

  Parameter
  `Long id`: The id to which the id of this announcement should be set to.

- `String getMessage()`
  Gets the message of this announcement.

  Return value: The id of this announcement.

- `int getUrgency()`
  Gets the urgency of this announcement.

  Return value: The id of this announcement.

- `Date getStartTime()`
  Gets the time at which this announcement should be displayed.

Return value: The time at which this announcement should be displayed.

- `Date getEndTime()`
  Gets the time at which this announcement should stop being displayed.

  Return value: The time at which this announcement should stop being displayed.

AnnouncementUrgency

Defines different urgencies for announcements.

```
enum Urgency {
  LOW("low")
  MEDIUM("medium")
  HIGH("high")

  String final label

  public Urgency(String label)
  public String toString()
  public Urgency fromLabel(String label)
}
```

Values

- `LOW("low")`
  Low urgency.

- `MEDIUM("medium")`
  Medium urgency.

- `HIGH("high")`
  High urgency.

Fields

- `String label`
  A string representation of the urgency.

Methods

- `Urgency(String label)`
  Initializes a new instance of an Urgency.

  Parameter
  `String label`: A string representation of the urgency.

- `String toString()`
  Returns the string representation of the urgency.

  Return value: The string representation of the urgency.

- `Urgency fromLabel(String label)`
  Gets the urgency enum type with the marching label.

  Parameter
  `String label`: The label of the enum type.

  Return value: The urgency with matching label.

4.2.3 Repository

AnnouncementRepository

Defines a repository containing announcements.

```
interface AnnouncementRepository extends CrudRepository<Announcement, Long> {

}
```

4.2.4 Service

AnnouncementService

Offers functionalities to interface with stored announcements.

```
class AnnouncementService implements Aggregable {
  private final String PLUGIN_ID
  private AnnouncementRepository repository
  private final JobSchedulerService jobScheduler
  private final ErrorLoggingService logger

  public AnnouncementService
    (ErrorLoggingService logger, JobSchedulerService jobScheduler)

  public List<Announcement> findAll()
  public Announcement findByID(long id) throws EntityNotFoundException
  public Announcement save(Announcement announcement)
  public void deleteById(long id)
  public String getPluginID()
}
```

Fields

- `String PLUGIN_ID`
  The ID of this plugin / service.

- `AnnouncementRepository repository`
  The repository containing all announcements.

- `JobSchedulerService jobScheduler`
  Used to schedule periodic tasks.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `AnnouncementService(...)` Initializes a new instance of an AnnouncementService.

  Parameter

  - `JobSchedulerService jobScheduler` Used to schedule periodic tasks.
  - `ErrorLogginService logger` Used to log errors.

- `ExternalCalendarService()`
  Initializes a new instance of an AnnouncementService.

- `List<Announcement> findAll()`
  Gets a list of all Announcements.

  Return value: A list containing all announcements of the repository.

- `Announcement findByID(long id) throws EntityNotFoundException`
  Gets an announcement by its unique id.

  Parameter
  `Long id`: The unique id of the announcement that should be fetched.
  Return value: The announcement matching the given id.
  Exception: EntityNotFoundException: Thrown if there is no announcement with the specified id.

- `Announcement save(Announcement announcement)`
  Adds an announcement to the repository.

  Parameter
  `Announcement announcement`: The announcement that should be added.

  Return value: The announcement that was added.

- `void deleteById(long id)`
  Deletes an announcement specified by its id from the repository.

  Parameter
  `Long id`: The id of the announcement that should be deleted.

- `String getPluginID()`
  Get the ID of this plugin.

  Return value: The ID of this plugin.

- `List<Announcement> aggregate()`
  Gets a list of all announcements.

  Return value: A list of all announcements.

4.3 Cafeteria plugin

4.3.1 Controller

CafeteriaController

Provides the api endpoints to communicate with the CafeteriaDataPlugin.

```
class CafeteriaController extends RequestController {
  private CafeteriaService cafeteriaService
  private ErrorLoggingService logger

  public ResponseEntity<RequestWrapper> findAll()
  public ResponseEntity<RequestWrapper> findByID
    (long cafeteria, String line, String day)
  public ResponseEntity<RequestWrapper> findByID(long cafeteria, String line)
  public ResponseEntity<RequestWrapper> findByID(long cafeteria)
}
```

Fields

- `CafeteriaService cafeteriaService`
  Used to retrieve the cafeteria menu.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `ResponseEntity<RequestWrapper> findAll()`
  Gets a list of all cafeterias and dishes.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints. See Cafeteria.

- `ResponseEntity<RequestWrapper> findByID(long cafeteria, String line, String day)`

Gets all dishes that will be served in the specified cafeteria at the specified line on the specified day.

Parameter
`long cafeteria`: The unique id of the cafeteria that should be fetched.
`String line`: The name of the line.
`String day`: The day at which the dishes will be served.
The value has to be between 0 and 6 (including)
Monday = 0, Tuesday = 1, ..., Sunday = 6 (the week starts on monday as ISO 8601 specifies)

Return value: A HTTP response is returned containing

- status code,
- the JSON response object as shown in the API endpoints. See Cafeteria.

- `ResponseEntity<RequestWrapper> findByID(long cafeteria, String line)`
  Gets a line by the cafeteria it belongs to and its name.

  Parameter
  `long cafeteria`: The unique id of the cafeteria that should be fetched.
  `String line`: The name of the line.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Cafeteria.

- `ResponseEntity<RequestWrapper> findByID(long cafeteria, String line)`
  Gets a cafeteria by its unique id.

  Parameter
  `long cafeteria`: The unique id of the cafeteria that should be fetched.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Cafeteria

4.3.2 Data

CafeteriaDish

A datatype describing a cafeteria dish.

```
class CafeteriaDish {
    private Long id
    private String mealName
    private float price
    private Date servedOn
    private Set<String> classifiers
    private Set<Integer> additives
    private Line line

    public CafeteriaDish()
    public CafeteriaDish
      (String mealName, float price, Set<String> classifiers, Set<Integer> additives)

    public Long getId()
    public String getMealName()
    public float getPrice()
    public Date getServedOn()
    public Set<String> getClassifiers()
    public Set<Integer> getAdditives()
    public Line getLine()
}
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the cafeteria dish can be referenced.

- `String mealName`
  The name of the meal.

- `float price`
  The price of the meal in euro.

- `Date servedOn`
  The day at which the dish can be purchased.

- `Set<String> classifiers`
  A list of classifiers, describing the meal.
  Example: `"vegan"`

- `Set<Integer> additives`
  A List of ids referring to special additives contained in the meal,
  Example: `"2"` stands for `"preservatives"`. This mapping is used by CafeteriaDish.

- `Line line`
  The line at which the dish is served at.

Methods

- `public CafeteriaDish()`
  Initializes a new empty instance of an CafeteriaDish.

- `public CafeteriaDish(String mealName, float price, Set<String> classifiers, Set<Integer> additives)`
  Initializes a new instance of a CafeteriaDish.

  Parameter
  `String mealName`: The name of the meal.
  `float price`: The price of the meal in euro.
  `Set<String> classifiers`: A list of classifiers, describing the meal.
  Example: `"vegan"`
  `Set<Integer> additives`: A List of ids referring to special additives contained in the meal.
  Example: `"2"` stands for "preservatives".

- `Long getId()`
  Gets the unique id of this cafeteria dish.

  Return value: The unique id of this cafeteria dish.

- `String getMealName()`
  Gets the name of this meal.

  Return value: The name of this meal.

- `float getPrice()`
  Gets the price of this meal.

Return value: The price of this meal in euro.

- `Date getServedOn()`
  Gets the date at which this dish can be purchased.

  Return value: The date at which this dish can be purchased.

- `Set<String> getClassifiers()`
  Gets classifiers describing this meal.

  Return value: A list of classifiers describing this meal.

- `Set<Integer> getAdditives()`
  Gets ids of special additives contained in the meal.

  Return value: A List of ids referring to special additives contained in the meal.

- `Line getLine()`
  Gets the line at which this dish is served at.

  Return value: The line at which this dish is served at.

OpeningHour

A datatype describing an opening hours of a cafeteria service.

```
class OpeningHour {
   private Long id
   private String name
   private Date openingTime
   private Date closingTime
   private Cafeteria cafeteria

   public OpeningHour()
   public OpeningHour
    (String name, Date openingTime, Date closingTime, Cafeteria cafeteria)

   public Long getId()
   public String getName()
   public Date getOpeningTime()
   public Date getClosingTime()
```

```
    public Cafeteria getCafeteria()
  }
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the opening hour can be referenced.

- `String name`
  The name of the service.
  Example: `"Mittagessen"`.

- `Date openingTime`
  The time at which this service opens.

- `Date closingTime`
  The time at which this service closes.

- `Cafeteria cafeteria`
  The cafeteria this service belongs to.

Methods

- `OpeningHour()`
  Initializes a new empty instance of an OpeningHour.

- `OpeningHour(String name, Date openingTime, Date closingTime)`
  Initializes a new instance of an OpeningHour.

  Parameter
  `String name`: The name of the service.
  Example: `"Mittagessen"`
  `Date openingTime`: The time at which this service opens.
  `Date closingTime`: The time at which this service closes.
  `Cafeteria cafeteria`: The cafeteria this service belongs to.

- `Long getId()`
  Gets the unique id of this opening hour.

  Return value: The unique id of this opening hour.

- `getName(): String`

Gets the name of this service.

Return value: The name of this service.

- `Date getOpeningTime()`
  Gets the time at which the service opens.

  Return value: The time at which this service opens.

- `Date getClosingTime()`
  Gets the time at which the service closes.

  Return value: The time at which this service closes.

- `Cafeteria getCafeteria()`
  Gets the cafeteria this service belongs to.

  Return value: The cafeteria this service belongs to.

Line

A datatype describing a cafeteria line.

```
class Line {
  private Long id
  private String linName
  private Set<CafeteriaDish> dishes
  private Cafeteria cafeteria

  public Line()
  public Line(String lineName, Set<CafeteriaDish> dishes, Cafeteria cafeteria)

  public Long getId()
  public String getLineName()
  public Set<CafeteriaDish> getDishes()
}
```

Fields

- `Long id`

A unique id if not set, automatically generated, with which the line can be referenced.

- `String linName`
  The name of the line.

- `Set<CafeteriaDish> dishes`
  Maps the day the dish will be served to the dish.

- `Cafeteria cafeteria`
  The cafeteria this line is located in.

Methods

- `Line()`
  Initializes a new empty instance of a Line.

- `Line(String lineName, Set<CafeteriaDish> dishes, Cafeteria cafeteria)`
  Initializes a new instance of a Line.

  Parameter
  `String lineName`: The name of the line.
  `Set<CafeteriaDish> dishes`: Maps the day the dish will be served to the dish.
  `Cafeteria cafeteria`: The cafeteria this line is located in.

- `Long getId()`
  Gets the unique id of this Line.

  Return value: The unique id of this Line.

- `String getLineName()`
  Gets the name of this line.

  Return value: The name of this line.

- `Set<CafeteriaDish> getDishes()`
  Gets a list of dishes available for purchase in the near future at this line.

  Return value: A list of dishes available for purchase at this line.

Cafeteria

A datatype describing a cafeteria.

```
class Cafeteria {
  private Long id
  private Date timeOfLastUpdate
  private String name
  private Set<OpeningHour> openingHours
  private Set<Line> lines
  private Map<Integer,String> additivesLegend

  public Cafeteria()
  public Cafeteria(Date timeOfLastUpdate, String name,Set<OpeningHour> openingHours,
    Set<Line> lines, HashMap<Integer, String> additivesLegend)

  public Long getId()
  public String getName()
  public Date getTimeOfLastUpdate()
  public Set<OpeningHour> getOpeningHours()
  public Set<Line> getLines()
  public Map<Integer,String> getAdditivesLegend()
}
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the cafeteria can be referenced.

- `Date timeOfLastUpdate`
  The last time the cafeteria data got updated.

- `String name`
  The name of the cafeteria.

- `Set<OpeningHour> openingHours`
  The opening hours of all services available.
  Example: `"Mittagessen"`

- `Set<Line> lines`
  The lines located in the cafeteria. A line contains meals for different days.

- `Map<Integer,String> additivesLegend`
  Maps an id to a corresponding additive.
  Example: `"2"` which stands for "preservatives". This mapping is used by CafeteriaDish.

Methods

- `Cafeteria()`
  Initializes a new empty instance of a Cafeteria.

- `Cafeteria(Date timeOfLastUpdate, String name, Set<OpeningHour> openingHours, Set<Line> lines, HashMap<Integer, String> additivesLegend)`
  Initializes a new instance of a Cafeteria.

  Parameter
  `Date timeOfLastUpdate`: The last time the cafeteria data got updated.
  `String name`: The name of the cafeteria.
  `Set<OpeningHour> openingHours`: The opening hours of all services available.
  Example: `"Mittagessen"`
  `Set<Line> lines`: The lines located in the cafeteria.
  `HashMap<Integer, String> additivesLegend`: Maps an id to a corresponding additive.
  Example: `"2"` which stands for "preservatives".

- `Long getId()`
  Gets the unique id of this Cafeteria.

  Return value: The id of this Cafeteria.

- `String getName()`
  Gets the name of this cafeteria.

  Return value: The name of this cafeteria.

- `Date getTimeOfLastUpdate()`
  Gets the last time this cafeteria got updated.

  Return value: The last time this cafeteria got updated.

- `Set<OpeningHour> getOpeningHours()`
  Gets the opening hours of all services available.
  Example: `"Mittagessen"`

Return value: The opening hours of all services available at this cafeteria.

- `Set<Line> getLines()`
  Gets the lines located in the cafeteria.
  A line contains meals for different days.

  Return value: The lines available at this cafeteria.

- `Map<Integer,String> getAdditivesLegend()`
  Gets a mapping between id and additive name.

  Return value
  A Mapping between id and additive name.

4.3.3 Service

CafeteriaDataConfig

Holds the configuration for the cafeteria data plugin.

```
class CafeteriaDataConfig implements Updatable {
  private final ConfigurationFileService configurationFileService
  private final ArrayList<String> configFileNames
  private String apiUserName
  private String apiKey
  private String[] selectedLocations
  private String apiGeneralUrl
  private String apiMenuUrl
  private int refreshTime

  public CafeteriaDataConfig(ConfigurationFileService configurationFileService)

  public boolean update()
  public String getApiUserName()
  public int getApiKey()
  public String[] getSelectedLocations()
  public String getApiGeneralUrl()
  public String getApiMenuUrl()
  public int getRefreshTime()
}
```

Fields

- `ConfigurationFileService configurationFileService`
  The service is used to interface with the config file.

- `ArrayList<String> configFileNames`
  Contains all file names that are used to set the configuration.

- `String apiUserName`
  The unser name used for accessing the data.

- `String apiKey`
  The api key used for accessing the data.

- `String[] selectedLocations`
  The id of the locations like lines or cafeterias that should be displayed.

- `String apiGeneralUrl`
  The url for the api that sends general data like the names of the lines or additives.

- `String apiMenuUrl`
  The url for the api that sends the menu data.

- `int refreshTime`
  The time in seconds after which the data should be refreshed.

Methods

- `CafeteriaDataConfig(ConfigurationFileService configurationFileService)`
  Initializes a new instance of a CafeteriaDataConfig.
  Loads properties from config files.

  Parameter
  `ConfigurationFileService configurationFileService`: The service is used to interface with the config files.

- `update(): boolean`
  Updates all properties related to the config file.
  Example: apiUserName, apiKey and the selectedLocations are updated.

  Return value True if all properties could be updated successfully without a restart.
  False if a restart of the server is required for at least one property.

- `String getApiUserName()`
  Gets the api username.

  Return value The api user name.

- `String getApiKey`
  Gets the api key.

  Return value The api key.

- `String[] getSelectedLocations()`
  Gets the locations that will be displayed.

  Return value The locations selected.

- `String getApiGeneralUrl()`
  Gets the url of the general api.

  Return value The general api's url.

- `String getApiMenuUrl()`
  Gets the url of the menu api.

  Return value The menu api's url.

- `getRefreshTime()`
  Gets the refresh time in seconds.

  Return value The refresh time in seconds.

CafeteriaService

Offers functionalities to interface to retrieve the current cafeteria menu.

```
class CafeteriaService implements Aggregable {
  private final String PLUGIN_ID
  private LocalRepository<Long, Cafeteria> repository
  private CafeteriaDataConfig config
  private final JobSchedulerService jobScheduler
  private final ErrorLoggingService logger
  public CafeteriaService(CafeteriaDataConfig config, JobSchedulerService jobSchedule

  public List<Cafeteria> findAll()
  public Cafeteria findCafeteriaByID(long id) throws EntityNotFoundException
  public Line findLineByID(long cafeteriaID, String lineName)
    throws EntityNotFoundException
  public List<CafeteriaDish> findDishByDay
    (long cafeteriaID, String lineName, int day) throws EntityNotFoundException
  public String getPluginID()
  public List<Cafeteria> aggregate()
  private void fetchCafeteria(){
}
```

Fields

- `String PLUGIN_ID`
  The ID of this plugin / service.

- `LocalRepository<Long, Cafeteria> repository`
  The repository containing cafeterias.

- `CafeteriaDataConfig config`
  Holds the current configuration.

- `JobSchedulerService jobScheduler`
  Used to schedule periodic tasks.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `CafeteriaService(...)`
  Initializes a new instance of a CafeteriaService.

  Parameter

  - `CafeteriaDataConfig config`: Holds the current configuration.
  - `JobSchedulerService jobScheduler` Used to schedule periodic tasks.
  - `ErrorLogginService logger` Used to log errors.

- `List<Cafeteria> findAll()`
  Gets a list of all cafeterias and their dishes.

Return value: A list containing all stored cafeterias.

- `Cafeteria findCafeteriaByID(long id) throws EntityNotFoundException`
  Gets a cafeteria by its unique id.

  Parameter
  `long id`: The unique id of the cafeteria that should be fetched.

  Return Value: The cafeteria matching the given id.

  Exception: EntityNotFoundException Thrown if there is no cafeteria with the specified id.

- `Line findLineByID(long cafeteriaID, String lineName)`
  `throws EntityNotFoundException`
  Gets a line by the cafeteria it belongs to and its name.

  Parameter
  `long cafeteriaID`: The unique id of the cafeteria.
  `lineName`: The name of the line.

  Return Value: The line with the matching cafeteria and name.

  Exception: EntityNotFoundException Thrown if there is no cafeteria with the specified id or line with the specified name.

- `List<CafeteriaDish> findDishByDay(long cafeteriaID, String lineName, int day) throws EntityNotFoundException`
  Gets all dishes that will be served in the specified cafeteria at the specified line on the specified day.

  Parameter
  `long cafeteriaID`: The unique id of the cafeteria.
  `String lineName`:The name of the line.
  `int day`: The day at which the dishes will be served. The value has to be between 0 and 6 (including)
  Monday = 0, Tuesday = 1, ..., Sunday = 6 (the week starts on monday as ISO 8601 specifies).

  Return Value: All dishes that will be served in the specified cafeteria at the specified line on the specified day.

  Exception: EntityNotFoundException: Thrown if there is no cafeteria with the specified id.

- `String getPluginID()`
  Get the ID of this plugin.

  Return value: The ID of this plugin.

- `List<Cafeteria> aggregate()`
  Gets a list of all cafeterias and their dishes.

  Return value: A list of all cafeterias and their dishes.

- `void fetchCafeteria()`
  Fetches the cafeteria Data from the external source.

4.4 Calendar plugin

4.4.1 Controller

CalendarController

Defines a basic calendar controller to retrieve calendars.

```
class CalendarController extends RequestController {
  private CalendarService calendarService
  private ErrorLoggingService logger
  public CalendarController(CalendarService calendarService)
  public ResponseEntity<RequestWrapper> findAll()
  public ResponseEntity<RequestWrapper> findById(long id)
}
```

Fields

- `CalendarService calendarService`
  Used to interface with the stored external calendars.

- `ErrorLoggingService logger`
  Used to log errors.

`Methods`

- `CalendarController(CalendarService calendarService)`
  Initializes a new instance of a CalendarController.

  Parameter
  `CalendarService calendarService`: The service used to retrieve calendars.

- `ResponseEntity<RequestWrapper> findAll()>`
  Gets a list of all calendars.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Calendars

- `ResponseEntity<RequestWrapper> findByID(long id)`
  Gets a calendar by its unique id.

  Parameter
  `long id`: The unique id of the calendar event that should be fetched.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Calendars

InternalCalendarController

Provides the api endpoints to communicate with the calendarDataPlugin (internal calendars).

```
class InternalCalendarController extends RequestController {
  private InternalCalendarService calendarService
  private ErrorLoggingService logger

  public InternalCalendarController(InternalCalendarService calendarService)

  public ResponseEntity<RequestWrapper> create(long id, CalendarEvent calendarEvent)
  public ResponseEntity<RequestWrapper> update(long id, CalendarEvent calendarEvent)
  public ResponseEntity<RequestWrapper> deleteById(long id)
}
```

Fields

- `InternalCalendarService calendarService`
  Used to interface with the stored internal calendars.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `InternalCalendarController(InternalCalendarService calendarService)`
  Initializes a new instance of an InternalCalendarController.

  Parameter
  `InternalCalendarService calendarService`: The service used to retrieve calendars.

- `ResponseEntity<RequestWrapper> create(long id, CalendarEvent calendarEvent)`
  Creates a new calendar event.

  Parameter
  `long id`: The calendar id.
  `CalendarEvent calendarEvent`: The calendar event that should be added.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Calendars.

- `ResponseEntity<RequestWrapper> update(long id, CalendarEvent calendarEvent)`
  Updates a calendar event specified by its id.

  Parameter
  `long id`: The id of the calendar event that should be updated.
  `CalendarEvent calendarEvent`: The calendar event with which the specified calendar event should be replaced.

  Return value: A HTTP response is returned containing

  - status code,
  - the JSON response object as shown in the API endpoints. See Calendars.

- `ResponseEntity<RequestWrapper> deleteById(long id)`

Deletes a calendar event specified by its id.

Parameter
`long id`: The id of the calendar event that should be deleted.

Return value: A HTTP response is returned containing

- status code,
- the JSON response object as shown in the API endpoints. See Calendars.

ExternalCalendarController

Provides the api endpoints to communicate with the calendarDataPlugin (external calendars).

```
class ExternalCalendarController extends CalendarController {
  private ExternalCalendarServcie calendarService
  private ErrorLoggingService logger
  public ExternalCalendarController(ExternalCalendarService calendarService)
}
```

Fields

- `ExternalCalendarServcie calendarService`
  Used to interface with the stored external calendars.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `ExternalCalendarController(ExternalCalendarService calendarService)`
  Initializes a new instance of an ExternalCalendarController.

  Parameter
  `ExternalCalendarService calendarService`: The service used to retrieve calendars.

4.4.2 Data

Calendar

A datatype describing a calendar.

```
class Calendar {
  private Long id
  private Date timeOfLastUpdate
  private String name
  private String color
  private boolean isInternal
  priavte Set<CalendarEvent> events

  public Calendar()
  public Calendar(Date timeOfLastUpdate, String name,
    String color, boolean isInternal, Set<CalendarEvent> events)

  public String getId()
  public Date getTimeOfLastUpdate()
  public String getName()
  public String getColor()
  public boolean isInternal()
  public Set<CalendarEvent> getEvents()
}
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the calendar can be referenced.

- `Date timeOfLastUpdate`
  The last time the calendar data got updated.

- `String name`
  The name of the calendar.

- `String color`
  A color which is used to color events of this calendar in hex.
  Example: `"#ffffff"` for white

- `boolean isInternal`
  True if the calendar can be modified. False if the calendar is from an external source and can't be modified.

- `Set<CalendarEvent> events`
  Contains all events of the calendar.

Methods

- `Calendar()`
  Initializes a new empty instance of a Calendar.

- `Calendar(Date timeOfLastUpdate, String name, String color, boolean isInternal, Set<CalendarEvent> events)`
  Initializes a new instance of a Calendar.

  Parameter
  `Date timeOfLastUpdate`: The last time the calendar data got updated.
  `String name`: The name of the calendar.
  `String color`: A color which is used to color events of this calendar in hex.
  `boolean isInternal`: True if the calendar can be modified. False if the calendar is from an external source and can't be modified.
  `Set<CalendarEvent> events`: Contains all events of the calendar.

- `Long getId()`
  Gets the unique id of this calendar.

  Return value: The id of this calendar.

- `Date getTimeOfLastUpdate()`
  Gets the last time this calendar got updated.

  Return value: The last time this calendar got updated.

- `String getName()`
  Gets the name of this calendar.

  Return value: The name of this calendar.

- `String getColor()`
  Gets the color which is used to present events of this calendar.

  Return value: The color which is used to present events of this calendar in hex format.

- `boolean isInternal()`
  Determines whether the calendar is internal and can be modified or external and thus can't be modified.

  Return value: True if the calendar can be modified. False if the calendar is from an external source and can't be modified.

- `Set<CalendarEvent> getEvents()`
  Gets all events of the calendar, events are ordered according to their starting time.

  Return value: The set contains all events of the calendar, events are ordered according to their starting time.

CalendarEvent

A datatype describing a calendar event.

```
class CalendarEvent {
  private Long id
  private String uid
  private String name
  private Date startTime
  private Date endTime
  private String location
  private boolean isAllDay
  private Calendar calendar

  public CalendarEvent()
  public CalendarEvent(String uid, String name, Date starTime, Date endTime,
    String location, boolean isAllDay, Calendar calendar)

  public Long getId()
  public void setId(Long id)
  public String getUid()
  public String getName()
  public Date getStartTime()
  public Date getEndTime()
  public String getLocation()
  public boolean getIsAllDay()
  public Calendar getCalendar()
  public void setCalendar(Calendar calendar)
  public int compareTo(CalendarEvent calendarEvent)
}
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the calendar event can be referenced.

- `String uid`
  The UID parameter of the iCalendar format.

- `String name`
  The name of the event, which is the same as the SUMMARY parameter in the iCalendar format.

- `Date startTime`
  The time the event starts, which is the same as the DTSTART parameter in iCalendar.

- `Date endTime`
  The time the event ends, this is either determined by the DTEND parameter or calculated with the DURRATION parameter of the iCalendar event.
  This may be optional.

- `String location`
  The location of the event.

- `boolean isAllDay`
  Determines whether the event is all day or not. This is the case if the DTEND parameter is not present or the DTSTART and DTEND are both just dates without time.

- `Calendar calendar`
  The calendar this event belongs to.

Methods

- `CalendarEvent()`
  Initializes a new empty instance of a CalendarEvent.

- `CalendarEvent(String uid, String name, Date starTime, Date endTime, String location, boolean isAllDay, Calendar calendar)`
  Parameter
  `String uid`: The UID parameter of the iCalendar format.
  `String name`: The name of the event.
  `Long startTime`: The time the event starts.
  `Long endTime`: The time the event ends.
  `String location`: The location of the event.
  `boolean isAllDay`: Determines whether the event is all day or not.
  `Calendar calendar`: The calendar this event belongs to.

- `Long getId()`
  Gets the unique id of this event.

  Return value: The id of this event.

- `void setId(Long id)`
  Set the unique id of this event.

  Parameter
  `Long id`: The id to which the id of this event should be set to.

- `String getUid()`
  Gets the UID of this calendar.

  Return value: The UID of this calendar.

- `String getName()`
  Gets the name of this event.

  Return value: The name of this event.

- `Date getStartTime()`
  Gets the time this event starts.

  Return value: The time this event starts.

- `Date getEndTime()`
  Gets the time this event ends.

  Return value: THe time this event ends.

- `String getLocation()`
  Gets the location of the event.

Return value: The location of the event.

- `boolean getIsAllDay()`
  Determines whether the event is all day or not.

  Return value: True if the event is an all day event, false if not.

- `Calendar getCalendar()`
  Gets the calendar this event belongs to.

  Return value: The calendar this event belongs to.

- `void setCalendar(Calendar calendar)`
  Sets the calendar this event belongs to.

  Parameter
  `Calendar calendar`: The calendar to which this event should belong to.

- `int compareTo(CalendarEvent calendarEvent)`
  Compares two calender events depending on their starting time.

  Parameter
  `CalendarEvent calendarEvent`: The event that should be compared.

  Return value

  - The value is 0 if the two events start at the same time.
  - The value is less 0 if this event start before the other event.
  - The value is greater 0 if this event start after the other event.

CalendarConfig

Class representing the data of an external calendar saved in the config file.

```
class CalendarConfig {
  private String name;
  private String url;
  private String color;

  public CalendarConfig(String name, String url, String color)
  public String getName()
  public String getUrl()
  public String getColor()
}
```

Fields

- `String name`
  The name of the calendar.

- `String url`
  The url of the calendar.

- `String color`
  The color of the calendar.

Methods

- `CalendarConfig(String name, String url, String color)`
  Initializes the calendarConfig object.

  Parameter
  `String name`: The name of the calendar.
  `String url`: The url of the calendar.
  `String color`: The color of the calendar.

- `String getName()`
  Gets name the of the calendar.

  Return value: The name of the calendar.

- `String getUrl()`
  Gets url the of the calendar.

  Return value: The url of the calendar.

- `String getColor()`
  Gets color the of the calendar.

  Return value: The color of the calendar.

4.4.3 Parser

The files in the package are autogenerated by the antlr plugin for gradle using the given gramar at src/main/antlr. The classes ICALexer and ICALParser will be used by the external calendar service to parse iCalendar data.

4.4.4 Repository

InternalCalendarRepository

Defines a repository containing calendars that are stored and managed internally.

```
interface InternalCalendarRepository extends CrudRepository<Calendar, Long> {

}
```

jository

Defines a repository containing calendar events that are stored and managed internally.

```
interface InternalCalendarEventRepository
  extends CrudRepository<CalendarEvent, Long> {

}
```

4.4.5 Service

CalendarDataConfig

Holds the configuration for the calendar data plugin.

```
class CalendarDataConfig implements Updatable {
  private final ConfigurationFileService configurationFileService
  private final ArrayList<String> configFileNames
  private CalendarConfig[] externalCalendarConfig
  private int refreshTime

  public CalendarDataConfig(ConfigurationFileService configurationFileService)

  public boolean update()
  public CalendarConfig[] getExternalCalendarConfig()
  public int getRefreshTime()
}
```

Fields

- `ConfigurationFileService configurationFileService`
  The service is used to interface with the config file.

- `ArrayList<String> configFileNames`
  Contains all file names that are used to set the configuration.

- `CalendarConfig[] externalCalendarConfig`
  The data for the configurations of the individual calendars.

- `int refreshTime`
  The time in seconds after which the data should be refreshed.

Methods

- `CalendarDataConfig(ConfigurationFileService configurationFileService)`
  Initializes a new instance of a CalendarDataConfig.
  Loads properties from config files.

  Parameter
  `ConfigurationFileService configurationFileService`: The service is used to interface with the config files.

- `boolean update()`
  Updates all properties related to the config file.

  Return value
  True if all properties could be updated successfully without a restart.
  False if a restart of the server is required for at least one property.

- `CalendarConfig[] getExternalCalendarConfig()`
  Gets the configurations of the calendars.

Return value The configurations of the calendars.

- `getRefreshTime()`
  Gets the refresh time in seconds.

  Return value The refresh time in seconds.

CalendarService

Defines a basic calendar service to retrieve calendars.

```
interface CalendarService {
  public List<Calendar> findAll()
  public Calendar findByID(long id) throws EntityNotFoundException
}
```

Methods

- `List<Calendar> findAll()`
  Gets a list of all Calendars.

  Return value: A list containing all stored Calendars.

- `Calendar findByID(long id) throws EntityNotFoundException`
  Gets an Calendar by its unique id.

  Parameter
  `long id`: The unique id of the calendar that should be fetched.

  Return Value: The calendar matching the given id.

  Exception: EntityNotFoundException: Thrown if there is no calendar with the specified id.

ExternalCalendarService

Offers functionalities to interface with calendar events fetched from external sources.

```
class ExternalCalendarService implements CalendarService, Aggregable, Updatable {
  private final String PLUGIN_ID
  private ExternalCalendarReposLocalRepository<Long, Calendar> calendarRepository
  private CalendarDataConfig config
  private final JobSchedulerService jobScheduler
  private final ErrorLoggingService logger

  public ExternalCalendarService()
  public ExternalCalendarService
    (CalendarDataConfig config, JobSchedulerService jobScheduler, ErrorLoggingService

  public List<Calendar> findAll()
  public Calendar findByID(long id) throws EntityNotFoundException
  public String getPluginID()
  public List<Calendar> aggregate()
  private void fetchCalendars()
}
```

Fields

- `String PLUGIN_ID`
  The ID of this plugin / service.

- `LocalRepository<Long, Calendar> calendarRepository`
  The repository containing all external calendars.

- `CalendarDataConfig config`
  Holds the current configuration.

- `CalendarDataConfig config`
  Used to schedule periodic tasks.

- `CalendarDataConfig config`
  Used to log errors.

Methods

- `ExternalCalendarService()`
  Initializes a new instance of an ExternalCalendarService.

- `ExternalCalendarService(...)`
  Initializes a new instance of an ExternalCalendarService.

Parameter

  - `CalendarDataConfig config` Holds the current configuration
  - `JobSchedulerService jobScheduler` Used to schedule periodic tasks.
  - `ErrorLogginService logger` Used to log errors.

- `List<Calendar> findAll()`
  Gets a list of all Calendars.

  Return value: A list containing all stored Calendars.

- `Calendar findByID(long id) throws EntityNotFoundException`
  Gets an Calendar by its unique id.

  Parameter
  `long id`: The unique id of the calendar that should be fetched.

  Return value: The calendar matching the given id.

  Exception: EntityNotFoundException: Thrown if there is no calendar with the specified id.

- `String getPluginID()`
  Get the ID of this plugin.

  Return value: The ID of this plugin.

- `List<Calendar> aggregate()`
  Gets a list of all publications.

  Return value: A list of all publications.

- `void fetchCalendars()`
  Fetches the calendar data from external sources.

InternalCalendarService

Offers functionalities to interface with calendar events that are stored and managed locally.

```
class InternalCalendarService implements CalendarService, Aggregable {
  private final String PLUGIN_ID
  private InternalCalendarRepository calendarRepository
  private InternalCalendarEventRepository calendarEventRepository
  private final JobSchedulerService jobScheduler
  private final ErrorLoggingService logger

  public InternalCalendarService()
  public InternalCalendarService(JobSchedulerService jobScheduler, ErrorLoggingServic

  public List<Calendar> findAll()
  public Calendar findByID(long id) throws EntityNotFoundException
  public CalendarEvent saveEvent(long id, CalendarEvent calendarEvent)
    throws EntityNotFoundException
  public CalendarEvent saveEvent(CalendarEvent calendarEvent)
  public CalendarEvent updateEvent(CalendarEvent calendarEvent)
    throws EntityNotFoundException
  public void deleteEventById(long id)
  public String getPluginID()
  public List<Calendar> aggregate()
}
```

Fields

- `String PLUGIN_ID`
  The ID of this plugin / service.

- `InternalCalendarRepository calendarRepository`
  The repository containing all internal calendars.

- `InternalCalendarEventRepository calendarEventRepository`
  The repository containing all internal calendar events.

- `JobSchedulerService jobScheduler`
  Used to schedule periodic tasks.

- `InternalCalendarEventRepository calendarEventRepository`
  Used to log errors.

Methods

- `InternalCalendarService()`
  Initializes a new instance of an InternalCalendarService.

- `InternalCalendarService(...)`
  Initializes a new instance of an InternalCalendarService.

  - `JobSchedulerService jobScheduler` Used to schedule periodic tasks.
  - `ErrorLogginService logger` Used to log errors.

- `List<Calendar> findAll()`
  Gets a list of all Calendars.

  Return value: A list containing all stored Calendars.

- `Calendar findByID(long id) throws EntityNotFoundException`
  Gets a list of all Calendars.

  Parameter
  `long id`: The unique id of the calendar that should be fetched.

  Return value: A list containing all stored Calendars.

  Exception: EntityNotFoundException: Thrown if there is no calendar with the specified id.

- `saveEvent(long id, CalendarEvent calendarEvent): CalendarEvent`
  Adds a calendar event to a calendar.

  Parameter
  `long id`: The id of the calendar, the event should be added to.
  `CalendarEvent calendarEvent`: The event that should be added to the calendar.

  Return value: The newly added calendar event.

  Exception: EntityNotFoundException: Thrown if there is no calendar with the specified id.

- `CalendarEvent saveEvent(CalendarEvent calendarEvent) throws EntityNotFoundEx.`
  Adds a calendar event to the repository.

  Parameter
  `CalendarEvent calendarEvent`: The calendar event that should be saved.

  Return value: The newly added calendar event.

- `CalendarEvent updateEvent(CalendarEvent calendarEvent)`

`throws EntityNotFoundException`
Updates the calendar event with the same id as the given calendar event.

Parameter
`CalendarEvent calendarEvent`: The calendar event containing the updated properties.

Return value: The updated calendar event.

Exception
EntityNotFoundException Thrown if there is no calendar event to update.

- `deleteEventById(long id): void`
  Deletes an event specified by its id from the repository.

  Parameter
  `long id`: The id of the event that should be deleted.

- `String getPluginID()`
  Get the ID of this plugin.

  Return value: The ID of this plugin.

- `List<Calendar> aggregate()`
  Gets a list of all internal calendars.

  Return value: A list of all internal calendars.

4.5 Publication plugin

4.5.1 Controller

PublicationController

Provides the api endpoints to communicate with the publicationDataPlugin.

```
class PublicationController extends RequestController {
  private PublicationService publicationService
  private ErrorLoggingService logger
  public ResponseEntity<RequestWrapper> findAll()
  public ResponseEntity<RequestWrapper> findByID(long id)
}
```

Fields

- `PublicationService publicationService`
  Used to retrieve publications.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `ResponseEntity<RequestWrapper> findAll()`
  Gets a list of all publications.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints. See Publications.

- `ResponseEntity<RequestWrapper> findByID(long id)`
  Gets a publication by its unique id.

  Parameter
  `long id`: The unique id of the publication that should be fetched.

  Return value: A HTTP response is returned containing

  – status code,
  – the JSON response object as shown in the API endpoints. See Publications.

4.5.2 Data

Publication

A datatype describing a publication.

```
class Publication {
  private Long id
  private String title
  private Set<String> authors
  private String publicationDate
  private String description
  private String publisher
  private String publishLocation

  public Publication()
  public Publication(String title, String authors, String publicationDate,
    String description, String publisher, String publishLocation)

  public Long getID()
  public String getTitle()
  public Set<String> getAuthors()
  public String getPublicationDate()
  public String getDescription()
  public String getPublisher()
  public Stirng getPublishLocation()
}
```

Fields

- `Long id`
  A unique id if not set, automatically generated, with which the publications can be referenced.

- `String title`
  The title of the publication.

- `String authors`
  All authors of the publication.

- `String publicationDate`
  The date when the publication was made public, this may come in different forms.
  Example: `"early access 2020"` or `"Jul-May 2018"`

- `String description`
  A short description of the publication.

- `String publisher`
  The source the publication got retrieved from.

Example: `"CES"` for the data from the CES website.

- `String publishLocation`
  The journal or the event where the publication was published.

Methods

- `Publication()`
  Initializes a new empty instance of an Publication.

- `Publication(String title, String authors, String publicationDate, String description,`
  `String publisher, String publishLocation)`
  Initializes a new instance of a Publication.

  Parameter
  `String title`: The title of the publication.
  `Set<String> authors`: All authors of the publication.
  `String PublicationDate`: The date when the publication was made public.
  `String description`: A short description of the publication.
  `String publisher`: The source the publication got retrieved from.
  `String publishLocation`: The the journal or the event where the publication was published.

- `Long getID()`
  Gets the unique id of this publication.

  Return value: The id of this publication.

- `String getTitle()`
  Gets the title of this publication.

  Return value: The title of this publication.

- `Set<String> getAuthors()`
  Gets all authors of the publication.

  Return value: The authors of the publication.

- `String getPublicationDate()`
  Gets the date when this publication was published.
  Example: `"early access 2020"` or `"Jul-May 2018"`

Return value: A String describing the date this publication got published.

- `String getDescription()`
  Gets A short description of this publication.

  Return value: A short description of this publication.

- `String getPublisher()`
  Gets the source this publication got retrieved from.
  Example: `"CES"` for the data from the CES website.

  Return value: The source this publication got retrieved from.

- `String getPublishLocation()`
  Gets the journal or the event where this publication was published.

  Return value: The journal or the event where this publication was published.

4.5.3 Service

PublicationDataConfig

Holds the configuration for the cafeteria data plugin.

```
class PublicationsDataConfig implements Updatable {
  private final ConfigurationFileService configurationFileService
  private final ArrayList<String> configFileNames
  private HashMap<String, String> dataSources
  private int refreshTime

  public PublicationsDataConfig(ConfigurationFileService configurationFileService)

  public boolean update()
  public Map<String, String> getDataSources()
  public int getRefreshTime()
}
```

Fields

- `ConfigurationFileService configurationFileService`
  The service is used to interface with the config file.

- `ArrayList<String> configFileNames`
  Contains all file names that are used to set the configuration.

- `HashMap<String, String> dataSources`
  Maps a data source name to an URL.

- `int refreshTime`
  The time in seconds after which the data should be refreshed.

Methods

- `PublicationsDataConfig(ConfigurationFileService configurationFileService)`
  Initializes a new instance of a PublicationsDataConfig.
  Loads properties from config files.

  Parameter
  `ConfigurationFileService configurationFileService`: The service is used to interface with the config files.

- `boolean update()`
  Updates all properties related to the config file.

  Return value
  True if all properties could be updated successfully without a restart.
  False if a restart of the server is required for at least one property.

- `Map<String, String> getDataSources()`
  Gets a mapping between a data source name and an URL.

  Return value A mapping between a data source name and an URL.

- `getRefreshTime()`
  Gets the refresh time in seconds.

  Return value The refresh time in seconds.

PublicationService

Offers functionalities to interface with stored publications fetched from external sources.

```
class PublicationService implements Aggregable, Updatable {
  private LocalRepository<Long, Publication> repository
  private final String PLUGIN_ID
  private PublicationsDataConfig config
  private final JobSchedulerService jobScheduler
  private final ErrorLoggingService logger

  public PublicationService()
  public PublicationService
    (PublicationsDataConfig config, JobSchedulerService jobScheduler, ErrorLoggingSer

  public List<Publication> findAll()
  public Publication findByID(long id) throws EntityNotFoundException
  public String getPluginID()
  public List<Publication> aggregate()
  private void fetchPublications()
}
```

Fields

- `String regex`
  Regex to extract information from the website.

- `LocalRepository<Long, Publication> repository`
  The repository containing all publications.

- `String PLUGIN_ID`
  The ID of this plugin / service.

- `PublicationsDataConfig config`
  Holds the current configuration.

- `JobSchedulerService jobScheduler`
  Used to schedule periodic tasks.

- `ErrorLoggingService logger`
  Used to log errors.

Methods

- `PublicationService()`
  Initializes a new instance of an PublicationService.

- `PublicationService(...)`
  Initializes a new instance of a PublicationService.

  - `PublicationsDataConfig config` Holds the current configuration.
  - `JobSchedulerService jobScheduler` Used to schedule periodic tasks.
  - `ErrorLogginService logger` Used to log errors.

- `List<Publication> findAll()`
  Gets a list of all publications.

  Return value: A list containing all publications of the repository.

- `Publication findByID(long id) throws EntityNotFoundException`
  Gets a publication by its unique id.

  Parameter
  `long id`: The unique id of the publication that should be fetched.

  Return value: The publication matching the given id.

  Exception: EntityNotFoundException: Thrown if there is no publication with the specified id.

- `String getPluginID()`
  Get the ID of this plugin.

  Return value: The ID of this plugin.

- `List<Publication> aggregate()`
  Gets a list of all publications.

  Return value: A list of all publications.

- `void fetchPublications()`
  Fetches and parses the Publications form external sources.

5 Class descriptions - Dashboard

5.1 Components

5.1.1 ConfigFileSection

React component for showing name, description, upload, download and edit button for a single config file.

```
const ConfigFileSection = (props:
    ConfigFileSectionProps
) => JSX.Element
```

Props: ConfigFileSectionProps

- `name: string`
  Name of the config file (most likely starts with an uppercase letter).
- `description: string`
  Description of what the config file does.
- `link: string`
  A link to further documentation about the specific config file.
- `id: string`
  A unique id for this config file.
- `download: () => Promise<string>`
  Function to retrieve the config file, this is used for the download button and editor.
- `upload: (file: string) => any`
  Function to upload a new config file, this is used for the upload button and editor.

5.1.2 Intl

React component which translated a word to a specified language and renders it.

```
const Intl = (props:
    IntlProps
) => JSX.Element
```

It is the exact same as the frontend Intl component.

Props: IntlProps

- `language`
  Language to be used. If not specified the current language is retrieved from the UIStore.

- `children`
  React child element which specifies which word to render (required if `word` is not specified).

- `word`
  Parameter which specifies which word to render (requried if `children` is not specified).

Example usage: `<Intl word="username" />`

5.1.3 Login

React component which renders the UI for a login prompt (username & password fields, login button, ...).

```
const Login = (props:
    LoginProps
) => JSX.Element
```

Props: LoginProps

- `cb: (credentials:  username: string, password: string ) => Promise<any>`
  Callback function to call with the entered credentials after the login button was clicked.

- `error: any`
  If any kind of error happened (incorrect credentials, ...) set this value to the error or something else not-falsy and an error message is displayed.

5.1.4 PageWrapper

React component which wraps the content of other pages inside of itself. Pages can use this component to render a sidebar and other default UI.

Each page needs to do this itself because some pages like the login page don't want the sidebar to render.

```
const PageWrapper = (props: {
    children: ReactNode,
    pages: Page[]
}) => JSX.Element
```

Props

- `children: ReactNode`
  React child element to render. This is the content of the page to render.

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Example

```
class SomePage extends Component {
  render() {
    return (
      <PageWrapper pages={this.props.pages}>
        Actual page content goes here
      </PageWrapper>
    )
  }
}
```

5.1.5 PrivateRoute

React component which adds authentication to a react-router route. This component handles acquisition of access tokens. This is done by first checking if one already exists and is valid and if not redirecting to the login page. Waiting time between requesting a new JWT access token and receiving it are also handled.

```
class PrivateRoute extends Component<PrivateRouteProps, any> {
  readonly props: PrivateRouteProps

  constructor(props: PrivateRouteProps): PrivateRoute
  componentDidMount(): void
  render(): JSX.Element
}
```

Fields

- `readonly props: PrivateRouteProps`
  Properties passed to a React Component are called `props` and are read-only.

  Note: `props` is not private but only ever used as such. It is defined by Component so this cannot be changed. This has the reason that JavaScript (excluding a TC39 stage 3 proposal for adding such a feature) does not have private class fields.

Methods

- `componentDidMount(): void`
  React lifecycle hook. Called when the component is mounted.

- `render(): JSX.Element`
  Method called by react to render the component.

5.1.6 Table

A highly customizable abstraction above html tables which allows to make simple but also complex tables.

All customizations are opt-in and sensible defaults were chosen.

```
const Table: (props: TableProps) => JSX.Element
```

Props: TableProps

- `headings: string[]`
  Table headings.

- `data: any[][]`
  Cell content, can be of any type renderable by react.

- `headingAlignment?: ('left' | 'center' | 'right')[] | ('left' | 'center' | 'right')`
  Alignment of headings.

- `dataAlignment?: ('left' | 'center' | 'right')[] | ('left' | 'center' | 'right')`
  Alignment of cells by column.

- `dataRowStyle?: React.CSSProperties[]`
  Set styling for each row independently. Does not affect headings.

- `dataColStyle?: React.CSSProperties[]`
  Set styling for each column independently. Does not affect headings.

- `dataRowClassName?: string[]`
  Set className for each row independently. Does not affect headings.

- `dataColClassName?: string[]`
  Set className for each column independently. Does not affect headings.

5.1.7 Button

A button react component with multiple styles and colors to choose from.

```
const Button = (props: ButtonProps) => JSX.Element
```

Props: ButtonProps

- `color?: 'red' | 'green' | 'blue' | 'yellow' | 'white'`
  The color of the button.

- `size?: 'small' | 'normal' | 'big'`
  The size of the button.

- `kind?: 'outline-grey' | 'outline-shade' | 'bottom-border'`
  The style of the button. Either outline-grey which adds a grey outline or outline-shade which adds an outline depending on color.

- `children: React.ReactNode`
  The text of the button.

- `[name: string]: any`
  Passes properties through to the button.

5.1.8 ButtonLookalike

A wrapper that makes things look like a button (comes with the same styling and color choises as a real Button).

Useful for styling clickable things without having an extra <button>...</button> around it.

```
const ButtonLookalike: (props: ButtonProps) => JSX.Element
```

Props: ButtonProps, same as Button.

5.1.9 TimeInput

A highly configurable time input react component which allows for relative and absolute time input, selecting from a list of preset times and having lots of different precision levels.

```
const TimeInput: (props: TimeInputProps) => JSX.Element
```

Props: TimeInputProps

- `format: 'relative' | 'absolute'`
  Wether to allow relative time inputs ("in 5 minutes") or absolute time inputs ("12:34").

- `value: number`
  The value of the input.

- `presets?: { [name: string]: number }`
  A list of preconfigured options to choose from in a dropdown menu.

- `precision?: 'year' | 'month' | 'week' | 'day' | 'hour' | 'minute' | 'second'`
  How precise the time inputs need to be. The smaller the unit the more precise an input needs to be. If e.g. `"month"` is chosen you cannot select anything more specific than that (unless a preset is used).

- `id: string`
  HTML id of the input component, useful for use with a label (`htmlFor`).

- `onChange: (time: number) => void`
  Callback function to indicate an update to the input has occurred. Time is always an absolute value, not relative, regardless of the format setting.

5.2 Pages

5.2.1 Page

A description of what a page is, similar to a Plugin.

```
type Page = {
  link: string,
  name: string,
  is_private: boolean,
  section: 'login' | 'admin' | 'editor',
  Component: Class<Component>
}
```

Fields

- `link: string`
  The link used for the page. If the page is provided by a plugin it must equal `"/" + plugin.id`

- `name: string`
  Name of the page to be used in the sidebar and as a headline. If the page is provided by a plugin it must equal the plugin name.

- `is_private: boolean`
  Wether the page requires authentication or not.

- `section: 'login' | 'admin' | 'editor'`
  Which section the page belongs to.

- `Component: Class<Component>`
  The react component which renders the page.

Pages and Plugins are equivalent to each other, one can be transformed into the other without data loss.

5.2.2 PageHandler

```
const load_plugin: (id: string) => Promise<Page>
const load_active_plugins: () => Promise<Page[]>
```

Functions

- `load_plugin`
  Load a plugin dynamically and turn it into a Page.

  Arguments

  – `id` of the plugin to load

  Return value: Promise resolving to the Page.

- `load_active_plugins`
  Load the plugins.yml config file and load all active (frontend) plugins as Pages.

  Return value: Promise resolving to an array of Pages.

5.2.3 LoginPage

Login page.

Note: It is assumed that this page is always accessible via `"/login"`.

```
class LoginPage extends Component<LoginPageProps, any> {
  readonly props: LoginPageProps

  render() : JSX.Element
}
```

Props: LoginPageProps

- `location?: Location<{ from: string }>`
  React-router Location from which a link to go back to after logging in successfully can be extracted (this is called a Callback or Redirect URL when talking about IDPs).

- `cb: (credentials: { username: string, password: string }) => Promise<any>`
  Callback function to call when a login attempt is made.

- `history: History`
  React-router History using which a redirect can be made.

Methods

- `render(): JSX.Element`
  Method called by react to render the component.

5.2.4 StartPage

Start Page.

```
class StartPage extends Component<{ pages: Page[] }, any> {
  props: { pages: Page[] }

  render(): JSX.Element
}
```

Props

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Methods

- `render(): JSX.Element`
  Method called by react to render the component.

5.2.5 CoreConfigurationPage

Core configuration page which handles uploading, downloading and editing of the core configuration files (general, layout and plugins).

```
class CoreConfigurationPage extends Component<{ pages: Page[] }, any> {
  props: { pages: Page[] }

  private download: (name: string, type: ConfigFileType) =>
    () => Promise<string>

  private upload: (name: string, type: ConfigFileType) =>
    (file: string) => Promise<string>

  render(): JSX.Element
}
```

Props

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Methods

- `download: (name: string, type: ConfigFileType) => () => Promise<string>`
  Used internally as a callback function/method for when the ConfigFileSection needs to download a config file.

- `upload: (name: string, type: ConfigFileType) => (file: string) => Promise<string>`

Used internally as a callback function/method for when the ConfigFileSection needs to upload a config file.

- `render(): JSX.Element`
  Method called by react to render the component.

5.2.6 PluginConfigurationPage

Plugin Configuration Page.

Shows upload-, download and edit-buttons for all data and widget config files.

```
class PluginConfigurationPage extends Component<{ pages: Page[] }, any> {
  props: { pages: Page[] }

  private download: (name: string, type: ConfigFileType) =>
    () => Promise<string>

  private upload: (name: string, type: ConfigFileType) =>
    (file: string) => Promise<string>

  render(): JSX.Element
}
```

Props

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Methods

- `download: (name: string, type: ConfigFileType) => () => Promise<string>`
  Used internally as a callback function/method for when the ConfigFileSection needs to download a config file.

- `upload: (name: string, type: ConfigFileType) => (file: string) => Promise<string>`
  Used internally as a callback function/method for when the ConfigFileSection needs to upload a config file.

- `render(): JSX.Element`
  Method called by react to render the component.

5.2.7 ErrorLogPage

Error log page which shows the error log and allows downloading it.

```
class ErrorLogPage extends Component<{ pages: Page[] }, any> {
  props: { pages: Page[] }

  private download(): void

  componentDidMount(): void
  render(): JSX.Element
}
```

Props

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Methods

- `download(): void`
  Downloads teh log file to the computer.

- `componentDidMount(): void`
  React lifecycle hook. Called when the component is mounted.

- `render(): JSX.Element`
  Method called by react to render the component.

5.2.8 AnnouncementPage

Announcements Page.

Allows creating and deleting announcements.

```
class AnnouncementPage extends Component<{ pages: Page[] }, AnnouncementPageState> {
  props: { pages: Page[] }
  private get_jwt: Promise<string>

  private changeUrgency(urgency: Urgency): void
  private changeMessage(e: React.ChangeEvent<HTMLTextAreaElement>): void
  private changeStartingTime(time: Date): void
  private changeEndTime(time: Date): void
  private deleteAnnouncement(id: number): void
  private createAnnouncement(): void

  componentDidMount(): void
  render(): JSX.Element
}
```

Props

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Methods

- `private changeUrgency(urgency: Urgency): void`
  Used internally as a callback function/method for when the urgency dropdown menu detects a change.

- `private changeMessage(e: React.ChangeEvent<HTMLTextAreaElement>): void`
  Used internally as a callback function/method for when the message input field detects a change.

- `private changeStartingTime(time: Date): void`
  Used internally as a callback function/method for when the starting time TimeInput detects a change.

- `private changeEndTime(time: Date): void`
  Used internally as a callback function/method for when the ending time TimeInput detects a change.

- `private deleteAnnouncement(id: number): void`
  Used internally as a callback function/method for when the delete action is clicked in the table of announcements.

- `private createAnnouncement(): void`

Used internally as a callback function/method for when the create announcement button is clicked.

- `componentDidMount(): JSX.Element`
  React lifecycle hook. Called when the component is mounted.

- `render(): JSX.Element`
  Method called by react to render the component.

State: AnnouncementPageState

Internal state of the announcement page tracking the inputs for creating announcements and the announcements to show in the table.

- `selected_message: string`
  Announcement message entered in the text input field.

- `selected_urgency: Urgency`
  Announcmenet urgency currently selected.

- `selected_start_time: number`
  Announcement starting time currently selected.

- `selected_end_time: number`
  Announcement end time currently selected.

- `announcements: withID<Announcement>[]`
  Announcements to show in the table of announcements (with IDs).

Announcement

Type representing an announcement.

```
type Announcement = {
  message: string,
  urgency: AnnouncementUrgency,
  startTime: number,
  endTime: number
}
```

Fields

- `message: string`
  The text of the announcement that will be displayed.
  Example: `"Warning internet outage at 12am!"`

- `urgency: AnnouncementUrgency`
  The urgency of the announcement. It may be either low, medium or high.
  Example: `AnnouncementUrgency.HIGH`

- `startTime: number`
  The time the announcement will start. This means it will be shown. The time is specified as a unix timestamp.

- `endTime: number`
  The time the announcement will end. This means it will no longer be shown. The time is specified as a unix timestamp.

5.2.9 CalendarPage

Calendar Page.

Allows managing the internal calendar(s).

```
class CalendarPage extends Component<{ pages: Page[] }, any> {
  props: { pages: Page[] }
  private get_jwt: Promise<string>

  private changeName(e: React.ChangeEvent<HTMLTextAreaElement>): void
  private changeLocation(e: React.ChangeEvent<HTMLTextAreaElement>): void
  private changeStartingTime(time: Date): void
  private changeEndTime(time: Date): void
  private changeAllDay(isAllDay: boolean): void
  private createEvent(): void

  componentDidMount(): void
  render(): JSX.Element
}
```

Props

- `pages: Page[]`
  Available pages throughout the whole application. Used for rendering the sidebar.

Methods

- `private changeName(e: React.ChangeEvent<HTMLTextAreaElement>): void`
  Used internally as a callback function/method for when the name text input changes.

- `private changeLocation(e: React.ChangeEvent<HTMLTextAreaElement>): void`
  Used internally as a callback function/method for when the location text input changes.

- `private changeStartingTime(time: Date): void`
  Used internally as a callback function/method for when the starting time TimeInput changes.

- `private changeEndTime(time: Date): void`
  Used internally as a callback function/method for when the ending time TimeInput changes.

- `private changeAllDay(isAllDay: boolean): void`
  Used internally as a callback function/method for when the all day checkbox changes its state.

- `private createEvent(): void`
  Used internally as a callback function/method for when the create event button is clicked.

- `componentDidMount(): void`
  React lifecycle hook. Called when the component is mounted.

- `render(): JSX.Element`
  Method called by react to render the component.

State

Internal state of the calendar page tracking the inputs for creating calendar events and the events to display in the table.

- `selected_name: string`
  Calendar name entered in the text input field.

- `selected_location: string`
  Calendar location entered in the text input field.

- `selected_start_time: number`
  Event starting time currently selected.

- `selected_end_time: number`
  Event ending time currently selected.

- `selected_all_day: boolean`
  Wether the 'all day' checkbox is currently checked for not.

- `events: CalendarEvent[]`
  Calendar events to show in the table of calendar events.

CalendarEvent

An Event inside a calendar.

```
type CalendarEvent = {
  id: number,
  uid: string,
  name: string,
  location: string,
  startTime: number,
  endTime: number,
  isAllDay: boolean
  }
```

Fields

- `id: number`
  Internal database id.

- `uid: string`
  UID from iCal, a unique id as a string.

- `name: string`
  Message / name of the event. Equal to the summary property in the iCal format.

- `location: string`
  String describing the location the event takes place at.

- `startTime: number`
  Starting time of the event as a unix timestamp.

- `endTime: number`
  End time of the event as a unix timestamp.

- `isAllDay: boolean`
  Boolean flag for indicating wether an event is the whole day long or not.

  Note: If the boolean flag is set the startTime and endTime values are interpreted as being date values only. Time information is disregarded.

Calendar

A type representing a calendar.

```
type Calendar = {
  id: number,
  name: string,
  color: string,
  timeOfLastUpdate: number,
  events: CalendarEvent[]
}
```

Fields

- `id: number`
  Id of the calendar.

- `name: string`
  Name of the calendar.

- `color: string`
  A preferred color for the calendar, may be used to make events more distinguishable.

  Note: Must be a valid CSS color (named color, hex, rgb, rgba, hsl, hsla, ...).

- `timeOfLastUpdate: number`
  Time the calendar was last updated (unix timestamp).

- `events: CalendarEvent[]`
  List of events that are part of the calendar.

5.3 Stores

5.3.1 UIStore

Mobx store which manages all UI related data. This is a shrunk down version of UIStore.

```
class UIStore {
  default_timezone: string
  date_format: string
  languages: string[]
  active_language: string
  themes: string[]
  active_theme: string
  language_data: Language[]

  constructor(): UIStore
  getLanguageByName(name: string): Language
}
```

Fields

- `default_timezone: string`
  Refer to GeneralConfig.default_timezone.

- `date_format: string`
  Refer to GeneralConfig.date_format.

- `languages: string[]`
  Refer to GeneralConfig.languages.

- `active_language: string`
  The currently active language.

- `themes: string[]`
  Refer to GeneralConfig.themes.

- `active_theme: string`
  The currently active theme.

- `language_data: Language[]`
  Loaded languages to be used by other pieces of code to display things in the correct language.

Methods

- `getLanguageByName(name: string): Language` This retrieves the current language by its name. This is used by the internationalization system.

5.4 Types for configuration files

5.4.1 GeneralConfig

Datatype for the datastructure of the general.yml config file.

```
type General = {
  default_timezone: string,
  alternative_timezones: string[],
  default_location: Location,
  available_languages: string[],
  language_switch_interval: number,
  languages: string[],
  date_format: string,
  available_themes: string[],
  themes: ({ theme: string } & (TimeRestriction | {}))[],
}
```

The same as in the frontend, see GeneralConfig.

5.4.2 PluginConfig

Datatype for the datastructure of the plugins.yml config file.

```
type Plugin = {
  available_data_plugins: string[],
  available_frontend_plugins: string[],
  available_dashboard_plugins: string[],
  active_data_plugins: string[],
  active_frontend_plugins: string[],
  active_dashboard_plugins: string[]
}
```

The same as in the frontend, see PluginConfig.

5.5 Interfaces and Internationalization

5.5.1 Language

Type describing what a language configuration file (or Localization file) should look like.

```
type Language = {
  name: string,
  locale: string,
  words: { [name: string]: string }
}
```

The same as in the frontend, see Language.

Fields

- `name: string`
  Name of the language. This is also the filename of the translation file.
- `locale: string`
  Locale code of the language. Used for date format strings.
- `words`
  Translations, mapping from an english word to its translation.

5.5.2 Storage

Interface describing what a storage method needs to support.

```
interface Storage<T> {
  getItem(key: string): any
  setItem(key: string, item: T): T
  removeItem(key: string): any

  clear(): void
}
```

`T` describes the type of data the store can safe, this could for example be `string` for a simple key-value store.

5.5.3 Plugin

Interface for encapsulating a plugin (react component) and its metadata.

```
interface Plugin {
  name: string
  id: string
  component: Class<Component<any, any>>
  role_requirement: 'editor' | 'admin'
}
```

Fields

- `name: string` Name of the plugin.

  Note: Can contain spaces, uppercase/lowercase letters and so on. Will **not** be used to uniquely identify a plugin.
  Example: `"Clock"`

- `id: string` Id of a plugin.

  Will be used to uniquely identify a plugin.
  Note: Needs to be all lowercase; can contain alphanumeric characters (no leading

digits), underscores (_) and dashes (-).
Example: `"image"`

- `component: Class<Component<any, any»,` React component class of the plugin page.
- `role_requirement: 'editor' | 'admin'` Which role is required for accessing the plugin page in the dashboard.

5.6 Utilities, Entrypoint and API Interactions

5.6.1 JWT

JWTBearerPayload

Type describing the structure of a JWT access tokens payload (also called bearer token as it is used with the Bearer authentication scheme). Follows the Open-ID Connect specifications but has a few keycloak specific values.

```
type JWTBasePayload = {
  exp: number,
  iat: number,
  jti: string,
  iss: string,
  azp: string,
  session_state: string,
  typ: 'Bearer'
  acr: string,
  realm_access: {
    roles: string[]
  },
  resource_access: {
    [name: string]: { roles: string[] }
  },
  scope: string,
  email_verified: boolean,
  preferred_username: string
}
```

Fields

- `exp: number`
  Expire time as unix timestamp.

- `iat: number`
  Issue time as unix timestamp.

- `jti: string`
  Unique identifier for the token.

- `iss: string`
  Issuer.

- `session_state: string`
  Keycloak value for the state of the session.

- `typ: 'Refresh'`
  Keycloak value to indicate the type of JWT, `"Bearer"` in this case.

- `acr: string`
  Authentication Context Class Reference.

- `realm_access: { roles: string[] }`
  Keycloak value which contains roles for the realm.

- `resource_access: { [name: string]: { roles: string[] } }`
  Keycloak value which contains a mapping from application to roles the user has. This value is used to determine the users permissions in the dashboard.

- `scope: string`
  Keycloak value for the scope.
  " Scopes usually represent the actions that can be performed on a resource, but they are not limited to that. You can also use scopes to represent one or more attributes within a resource " (source).

- `email_verified: boolean`
  Keycloak value indicating wether the email address used to log in is verified or not.

- `preferred_username: string`
  Keycloak value for the username of the user that this JWT belongs to.

JWTRefreshPayload

Type describing the structure of a JWT refresh tokens payload. Follows the Open-ID Connect specifications but has a few keycloak specific values.

```
type JWTRefreshPayload = {
  exp: number,
  iat: number,
  jti: string,
  iss: string,
  azp: string,
  session_state: string,
  typ: 'Refresh'
}
```

Fields

- `exp: number`
  Expire time as unix timestamp.

- `iat: number`
  Issue time as unix timestamp.

- `jti: string`
  Unique identifier for the token.

- `iss: string`
  Issuer.

- `session_state: string`
  Keycloak value for the state of the session.

- `typ: 'Refresh'`
  Keycloak value to indicate the type of JWT, `"Refresh"` in this case.

SimplifiedJWT

Simplified version of a JWT payload. Works with both Refresh and Bearer type.

```
type SimplifiedJWT = {
  exp: number,
  iat: number,
  iss: string,
  typ: 'Bearer' | 'Refresh',
  roles?: string[]
}
```

Fields

- `exp: number`
  Expire time as a unix timestamp.
- `iat: number`
  . Issue time as unix timestamp.
- `jti: string`
  Unique identifier for the token.
- `iss: string`
  Issuer.
- `session_state: string`
  Keycloak value for the state of the session.
- `typ: 'Bearer' | 'Refresh'`
  Keycloak value to indicate the type of JWT.
- `roles?: string[]`
  Keycloak value indicating the users roles. This only exists for Bearer tokens and uses `window.APP_NAME` to retrieve the property from `resource_access`.

5.6.2 JWTStrategy

```
class JWTStrategy {
  login(creds: { username: string, password: string }): Promise<{
    access_token: string,
    expires_in: number,
    refresh_token: string,
    refresh_token_expires_in: number
  }>
  logout(): void

  get_jwt(): Promise<string>
}
```

Methods

- `login(creds: { username: string, password: string }): Promise<...>`
  Retrieve access_token and refresh_token using credentials (username and password).

  Return value: Returns a promise which will resolve to an object containing refresh and access_token or reject if the credentials are incorrect or something else went wrong.

- `logout(): void`
  Log out.

  This does **not** tell the server to invalidate the access_token or similar. It just nukes all client side information about the access_token and refresh_token.

- `get_jwt(): Promise<string>`
  Get JWT access_token if possible.

  If it is possible to get a valid JWT the promise resolves to that, if not the promise rejects. Promise rejection means that relogging is required.

  Use this function to make API calls by waiting for the promise to resolve and sending the received token along with the request.

  Example:

  ```
  get_jwt()
    .then(jwt => fetch(url, {
      ...
      headers: { Authentication: 'Bearer ' + jwt }
    }))
    .catch(err => { ... })
  ```

5.6.3 localStorageStorage

Storage implementation using localStorage.

```
class LocalStorageStorage implements Storage<string> {
  getItem(key: string): string | null
  setItem(key: string, item: string): string
  removeItem(key: string): string | null
  clear(): void
}
```

localStorage only supports storing strings as values. More complex storage methods such as JSON objects can be built untop of that but this has not been done here.

5.6.4 API

Group of functions for making API calls. This group does not include calls for authentication, this is done by JWTStrategy.

```
const current_server_version: () => Promise<string>

const current_server_uptime: () => Promise<number>

const upload_config_file: (
  filename: string,
  type: ConfigFileType,
  file: string,
  jwt: string
) => Promise<string>

const download_config_file: (
  filename: string,
  type: ConfigFileType,
  jwt: string
) => Promise<string>

const download_error_log: (jwt: string) => Promise<string>

const get_announcements: (jwt: string) => Promise<withID<Announcement, number>[]>
const create_announcement: (jwt: string, data: Announcement) =>
  Promise<withID<Announcement, number>>

const delete_announcement: (jwt: string, id: number) => Promise<{
  status: 'success' | 'failure',
  message: string
}>

const update_announcement: (jwt: string, id: number, data: Announcement) =>
  Promise<withID<Announcement, number>>

const list_calendars: (jwt: string) => Promise<Calendar[]>

const get_calendar: (jwt: string, id: string) => Promise<Calendar>

const create_calendar_event: (
  jwt: string,
  cal_id: string,
  data: CalendarEvent
) => Promise<CalendarEvent>

const update_calendar_event: (
  jwt: string,
  event_id: number,
  data: CalendarEvent
) => Promise<CalendarEvent>

const delete_calendar_event: (jwt: string, event_id: number) => Promise<{
  status: 'success' | 'failure',
  message: string
}>
```

Functions

- `const current_server_version: () => Promise<string>`
  Get the current version of the backend server.

  Note: This might be used as a ping to check if the server is still running.

- `const current_server_uptime: () => Promise<number>` Get the current uptime of the backend server.

Note: This might be used as a ping to check if the server is still running.

- `const upload_config_file: (...) => Promise<string>`
  Upload a config file to the backend server.

  Arguments

  - `filename: string` The name of the config file to replace / upload.
  - `type: ConfigFileType` The type of config file (CORE, DATA, WIDGET).
  - `file: string` The content of the file.
  - `jwt: string` for authentication.

- `const download_config_file: (...) => Promise<string>`
  Download a config file from the backend server.

  Arguments

  - `filename: string` The name of the config file to load.
  - `type: ConfigFileType` The type of config file (CORE, DATA, WIDGET).
  - `jwt: string` for authentication.

- `const download_error_log: (jwt: string) => Promise<string>`
  Load the error log file from the backend server.

  Arguments

  - `jwt: string`: JWT for authentication.

  Return value: Promise resolving to the error log file.

- `const get_announcements: (jwt: string) => Promise<withID<Announcement, number>[]>`
  Get a list of announcements.

  Arguments

  - `jwt: string`: JWT for authentication (not required for this endpoint but used for consistency with other endpoints).

  Return value: Promise with list of announcements.

- `const create_announcement: (jwt: string, data: Announcement) =>`

`Promise<withID<Announcement, number> >`
Create an announcement.

Arguments

  - `jwt: string`: JWT for authentication.
  - `data: Announcement`: Announcement.

Return value: Promise with announcement (with ID).

- `const delete_announcement: (jwt: string, id: number) => Promise<...>`
  Delete an announcement.

  Arguments

  - `jwt: string`: JWT for authentication.
  - `id: number`: Id of the announcement.

  Return value: Promise

- `const update_announcement: (jwt: string, id: number, data: Announcement) =>`
  `Promise<withID<Announcement, number> >`
  Update an announcement.

  Arguments

  - `jwt: string`: JWT for authentication.
  - `id: number`: Id of the announcement.
  - `data: Announcement`: Announcement.

  Return value: Promise with announcement.

- `const list_calendars: (jwt: string) => Promise<Calendar[]>`
  Get all calendars and their events.

  Arguments

  - `jwt: string`: JWT for authentication.

  Return value: Promise resolving to a list of calendars.

- `const get_calendar: (jwt: string, id: string) => Promise<Calendar>`
  Get a calendar with all its events.

  Arguments

  - `jwt: string`: JWT for authentication.
  - `id: number`: Id of the calendar.

  Return value: Promise resolving to the calendar.

- `const create_calendar_event: (jwt: ..., cal_id: ..., data: ...) => Promise<CalendarEvent>`
  Create a calendar event.

  Arguments

  - `jwt: string`: JWT for authentication.
  - `cal_id: string`: Calendar ID.
  - `data: CalendarEvent`: Calendar event.

  Return value: Promise resolving to the calendar event.

- `const update_calendar_event: (jwt: ..., event_id: ..., data: ...) => Promise<CalendarEvent>`
  Update a calendar event.

  Arguments

  - `jwt: string`: JWT for authentication.
  - `event_id: number`: Id of the event.
  - `data: CalendarEvent`: Calendar event.

  Return value: Promise resolving to the calendar event.

- `const delete_calendar_event: (jwt: string, event_id: number) => Promise<...>`
  Delete a calendar event.

  Arguments

  - `jwt: string`: JWT for authentication.
  - `event_id: number`: Id of the event.

  Return value: Promise

5.6.5 App

Entrypoint to the app.

Handles adding all react-router Routes and PrivateRoutes with the help of PageHandler.

```
const App = () => JSX.Element
```

6 Class diagrams - Frontend

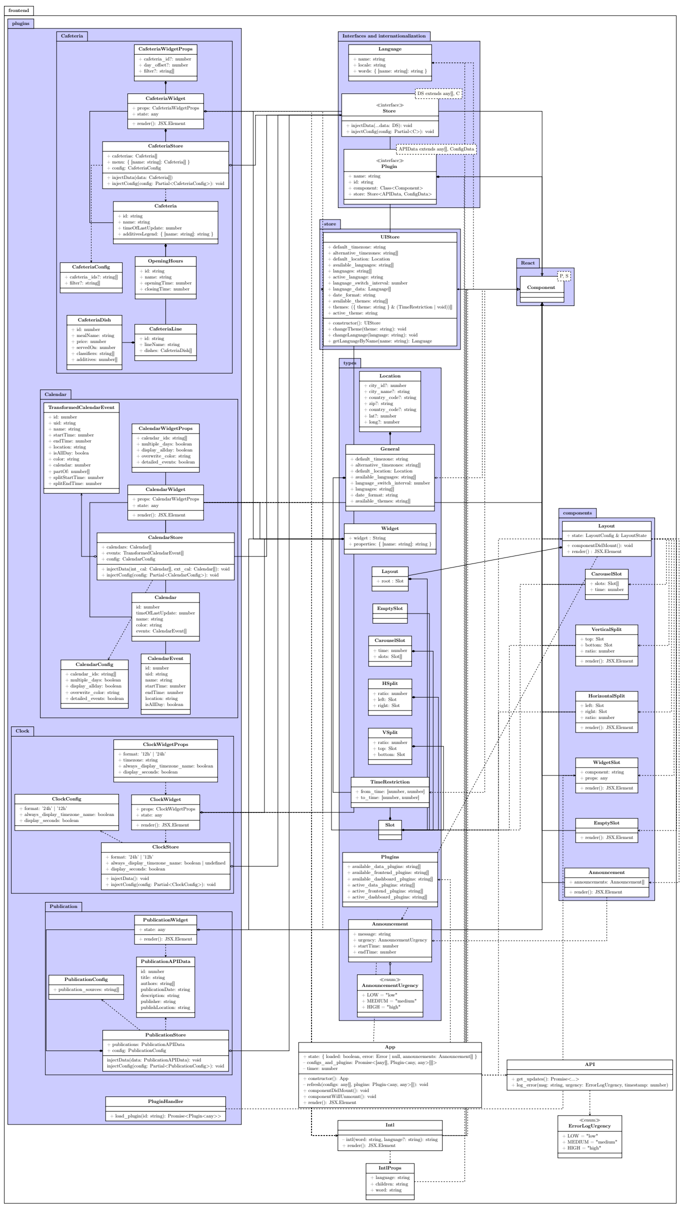


Figure 6.1: frontend class diagram

7 Class diagrams - Backend

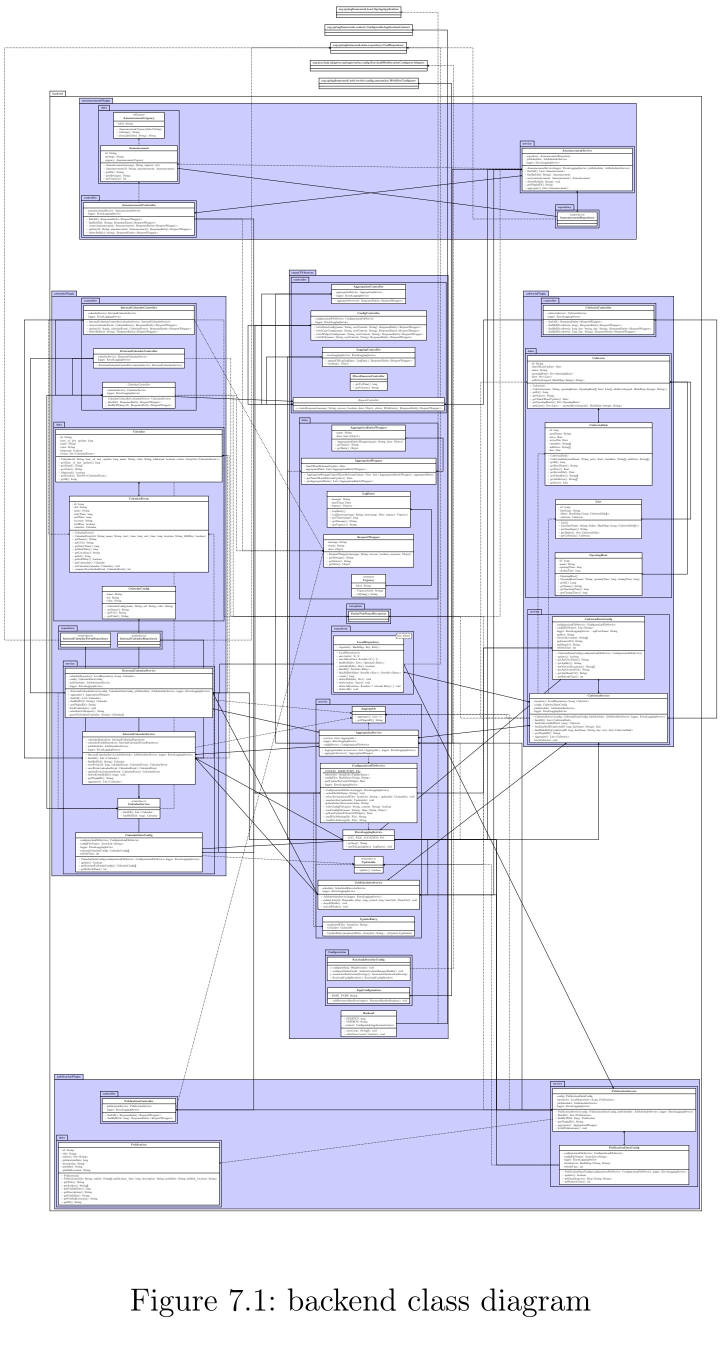


Figure 7.1: backend class diagram

8 Class diagrams - Dashboard

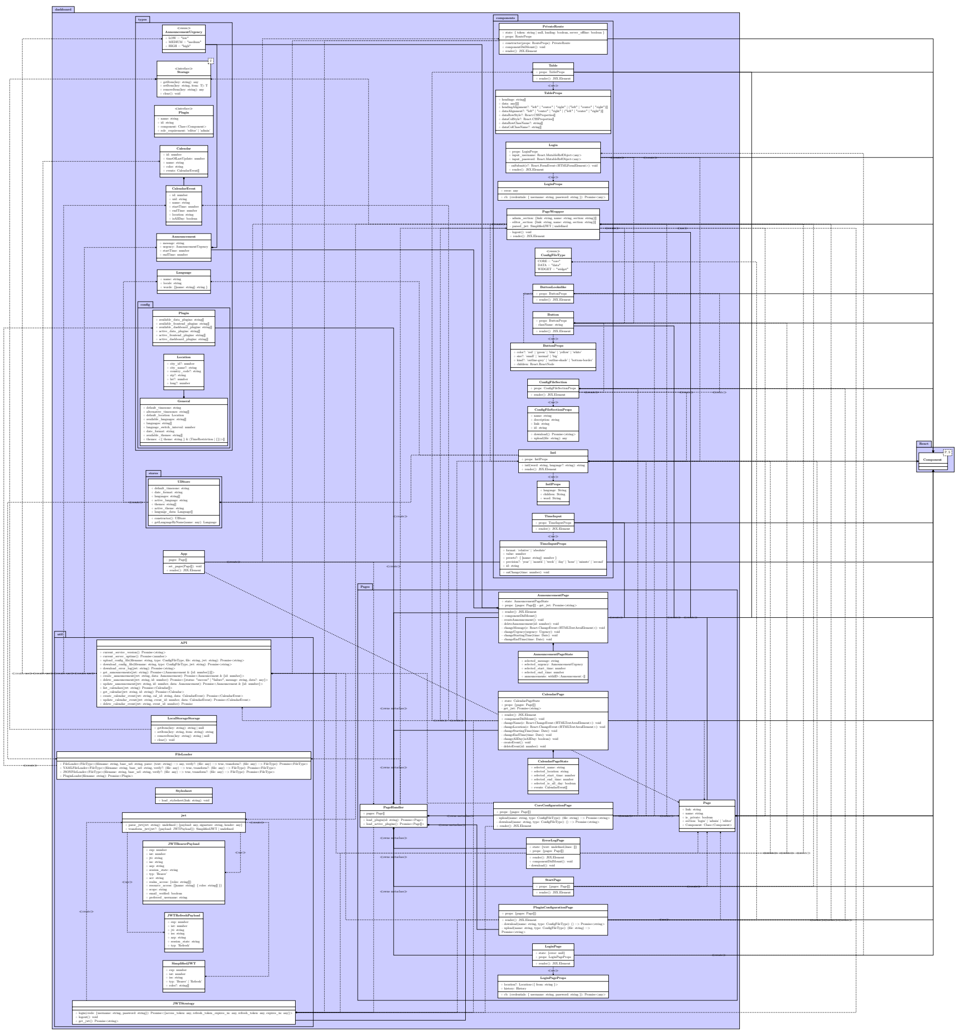


Figure 8.1: dashboard class diagram

9 Design and architectural patterns

9.1 Architectural patterns

9.1.1 Model-view-controller

The model-view-controller architecture or MVC consists of three parts a model, a view and a controller.

The model manages the data of the application.

The view presents the data of the application.

The controller is located between the model and the view. It houses the main logic of the application and relays information between the two.

The overall architecture of this project is MVC. The model and controller is located in the backend, the dashboard and frontend represent the different views. But the internal architecture of the backend, dashboard and frontend also conforms to the MVC architectural pattern.

The backend is structured in three major parts the controllers which represent the view, the services which represent the controller and the repositories which represent the model.

The frontend and backend have a slightly different structure. The essential differentiation is between so called *smart components* and *dumb components* (and other code that isn't a component at all).

Dumb components make up most of what could be called the view.

Smart components are, as well as stores, the brain of the whole system. They do some rendering but most of it is delegated to dumb components. The stores (which aren't components) act as mediators between the backend (through api endpoints) and the rest of the frontend.

Stores and smart components fit (mostly) into the controller category and the code for interacting with the api into the model category.

9.1.2 Framework

The framework is an architectur style by which the program offers interfaces and services that can be uesd by plugins, so they can comunicate and inegrate with the main program.

Examples

`Aggregable`, `ConfigurationFileService`, `JobSchedulerService`, `ErrorLoggingService`

9.2 Design patterns

9.2.1 Observer

The Observer pattern is a software design pattern in which an object maintains a list of its dependents, called *observers*, and notifies them automatically of any state changes.

The pattern is used in the frontend, backend and dashboard.

Frontend

The frontend uses mobx stores which are really similar to both the observer and mediator design patterns. Mobx stores are the single source of truth for some data they manage and everyone accessing this data must subscribe to its updates. The data can only be modified by the mobx store itself. The mobx store exposes some “actions” to the outside through which the data can be updated. Only when updating the data through this method can all changes to the data be observed and all subscribers be notified. React itself is also highly based on the observer pattern, mobx provides however a way of updating and notifying components irregardless of the component hierarchy and without having to propagate updates through a tree of components.

Dashboard

The dashboard has the same kind of Observers the frontend has.

Backend

In the backend the `Updatable` interface is used to notify classes of updates to relevant config files. Every class implementing the interface has to create a method called `update()` which is called when a config file change occured. Using this mechanism individial plugins can be notified of updates to their config files. In case a plugin cannot handle the update it can just return `false` to force a restart. Each implementation of `Updatable` should use the `subscribe()` and `unsubscribe()` methods of the `ConfigurationFileService` Singleton to subscribe to relevant config files.

9.2.2 Singleton

The Singleton pattern enforces that only one instance of a class exists at one time.

The pattern is widely used in the frontend, dashboard as well as the backend architecture to ensure that there exists exactly one instance of some classes or code. Most of the singletons are not technically singletons in the traditional sense with a `getInstance`-method but still act in exactly the same way.

Frontend

Since Typescript (and also Javascript) are not limited in what a file can export, unlike Java is, files can just export objects without ever having to export the class itself. This way the exported object is the only instance created of a class making it a singleton practically.

All mobx stores are singletons.
All plugin entrypoints can also be seen as singletons.
Each instance of Plugin is a singleton of the more specialized type `Plugin<T>` with a unique `T` for every instance.

Examples

`UIStore`, `CafeteriaStore`, `CalendarStore`, `Clock`

Dashboard

The dashboard has the same kind of singletons the frontend has.

Backend

In the backend the pattern is used to have a single object which handles config files instead of multiple potentially creating problems with file descriptors.

All `controlles`, `services` and `repositories` (spring terminology) are also used in a singleton-like way. Only one instance of each is created as it would not make sense to have multiple objects listening for the same network connections or make the same database accesses.

Examples

`AnnouncementService`, `AnnouncementRepository`, `AnnouncementController`

9.2.3 Template

The Template pattern is widely used in backend, frontend and dashboard to extract common logic from different pieces of code into one template method and then let every single instance do the rest of its own unique logic.

Frontend

All react components use the template method.

The `render()` method is the most important of the template methods, it specifies what should be rendered. A react component has more template methods though, all react lifecycle hooks (like `componentDidMount()`) are optional template methods using which default behaviour can be overwritten or hooked into.

Dashboard

Same as the frontend.

Backend

All the classes implementing the interface `Updatable` or the interface `Aggregable` use the Template pattern. `Updatable` and `Aggregable` provide two template methods: `update()` and `aggregate()`.

These two methods need to be overwritten in every single subclasses, such as `PublicationService`. Depending on the widget where they are used, `update()` will update different kinds of data and `aggregate()` provides different identifiers and the data that should be retrieved.

9.2.4 Façade

The Façade pattern is a software design pattern in which an object serves as a front-facing interface or class to mask a larger body of code.

This pattern is used in frontend, dashboard and backend.

Frontend

In the frontend the `index` class is used to dispatch the access to different plugins. It holds a list of all the Plugins and also a `load_plugin` method to load the plugin from a remote location and later inject the corresponding plugin dependencies into it.

Dashboard

The dashboard has a `PageHandler` which manages all built-in pages as well as dynamically loadable pages (through plugins).

It holds a list of pages that already exist in the code like `CalendarPage` or `CoreConfigurationPage` and has a method called `load_plugin` which can load additional plugins / pages and inject relevant dependencies into them.

Backend

In essence the Update endpoint is a facade combining all other (data) endpoints into a single one.

9.2.5 Command

The Command pattern is widely used in the frontend, dashboard and backend to use an object to encapsulate all information needed to perform an action or trigger an event at a later time.

Some components like the `Login`-component use the command pattern to dispatch certain functions after an event has occured. Login has an event handler called `onSubmit` which fires when the "Login"-button has been pressed.

In general, all callback functions use the command pattern aiming to continue code execution after an asynchronous operation has completed.
A good example here is the callback function executed inside a `.then` block from a `Promise` which will be chained onto the end of a Promise after the Promise rejects or

resolves.

Example

Functions passed to `.then` and `.catch` to be executed later when a response from the API is available.

```
this.current_version_promise
  .then(version => window.BACKEND_VERSION = version)
  .catch(() => this.setState({ server_offline: true }))
```

There are more Commands though, a *predicate* which will return a boolean value based on the input und decides often whether an object includes in a list.

Example

Predicate determining if the section of a page is 'admin' or not.

```
const admin_section = props.pages.filter(({ section }) => section === 'admin')
```

9.2.6 Proxy

Proxy pattern provides a proxy between a subject and a client. A proxy controls access to the original object.

9.2.7 Mediator

The Mediator pattern is a software design pattern that defines an object that encapsulates how a set of objects interact.

Frontend

The mediator pattern is used in the frontend in the form of react and mobx store.

The most common example here is the `Observerable` and `Observer` in mobx store. An observer asks to be notified automatically and react alone when something happens in data structure that is marked as observable without the necessity of subscribing.

In fact, the mobx store itself, which knows the most relevant state of the program and manages changes in this state and then forwards it to the appropriate objects when it changes, is also a kind of mediator.

Dashboard

The dashboard also uses react and mobx stores like the frontend.

Backend

Spring is a giant mediator which *autowires* a lot of things together and handles the interactions between many things (Spring connects the controllers with the respective services and much more).

9.2.8 Composite

The composite pattern is a software design pattern in which a group of objects will be treated the same way as a single instance of object in the same type.

A common example of the composite pattern is a filesystem. A filesystem has "objects" which are either folders or files, each folder can have another list of "objects" which in turn can also be either folders or files. Both folders and files share some common actions that can be performed (like deletion) but also have some actions reserved for their specific type.

The composite pattern is used in the frontend for the layout system.

The Layout system of the smart TV is divided into `slots`. A `slot` can hold a widget (`WidgetSlot`), hold nothing (`EmptySlot`), or can contain a list of `slots` to cycle through(`CarouselSlot`). All these `slots` share some common actions like `HorizontalSplit` and `VerticalSplit`, through which a `slot` can be divided into two more `slots`, but also have some actions reserved for their specific type.

9.2.9 Adapter

The Adapter pattern is a software design pattern that converts the interface of a class to another interface that the clients wants. The adapter pattern allows classes that cannot work together due to incompatible interfaces to work together.

The dashboard uses this with Plugins and Pages which are technically equivalent but have a slightly different datastructure which sometimes needs to be converted between.

The method `load_plugin` loads a Plugin and transforms it into a Page.

10 Sequence diagrams

10.1 Backend

10.1.1 Initial startup

This diagram shows how the bakcend sever will be started. The initialisation of services, controllers, configurations and repositories is simplified, because there are more than 20 classes that are initialized.

PropertiesLauncher is used to load all plugins into the classPath.

SpringBeanFactory.createBeans() initializes all services, controllers, configurations and repositories located in the classPath.

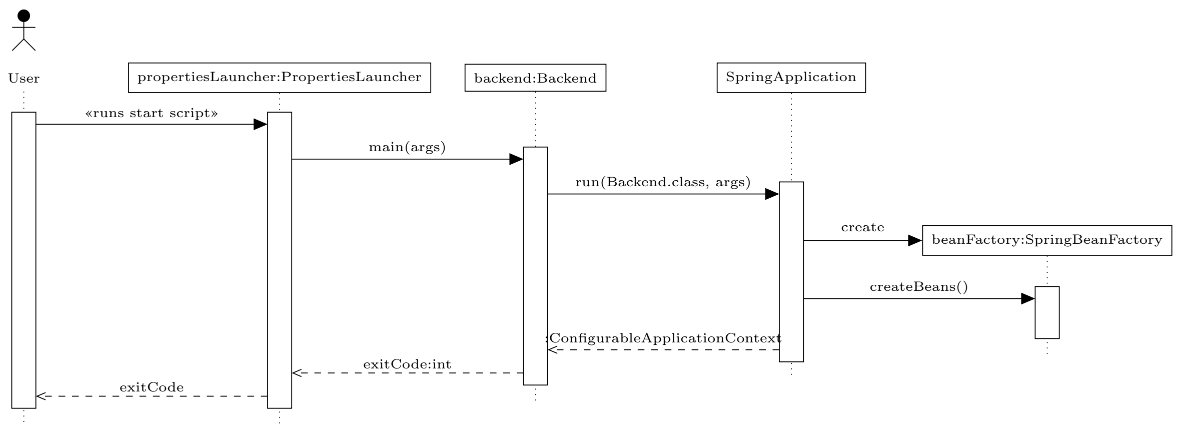


Figure 10.1: backend: initial startup

10.1.2 Create announcement

This diagram is an example for handling create requests from the dashboard. The process to create calendar events is very similar.

To update data the process is very similar as well.

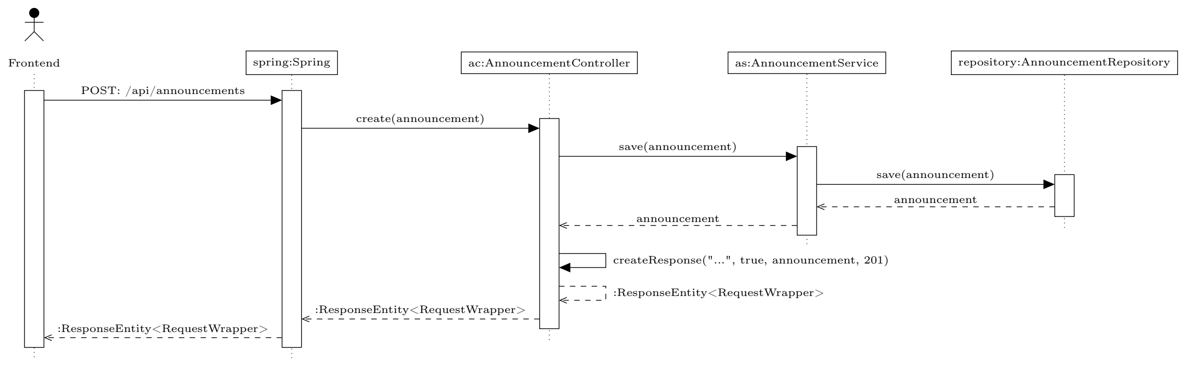

Frontend
spring:Spring
ac:AnnouncementController
as:AnnouncementService
repository:AnnouncementRepository
POST: /api/announcements
create(announcement)
save(announcement)
save(announcement)
announcement
announcement
createResponse("...", true, announcement, 201)
:ResponseEntity<RequestWrapper>
:ResponseEntity<RequestWrapper>
:ResponseEntity<RequestWrapper>

Figure 10.2: backend: create announcement

10.1.3 Update request

This diagram shows how an update request is handled. All find requests are very similar, because in theory the update endpoint just collects all other find endpoints.

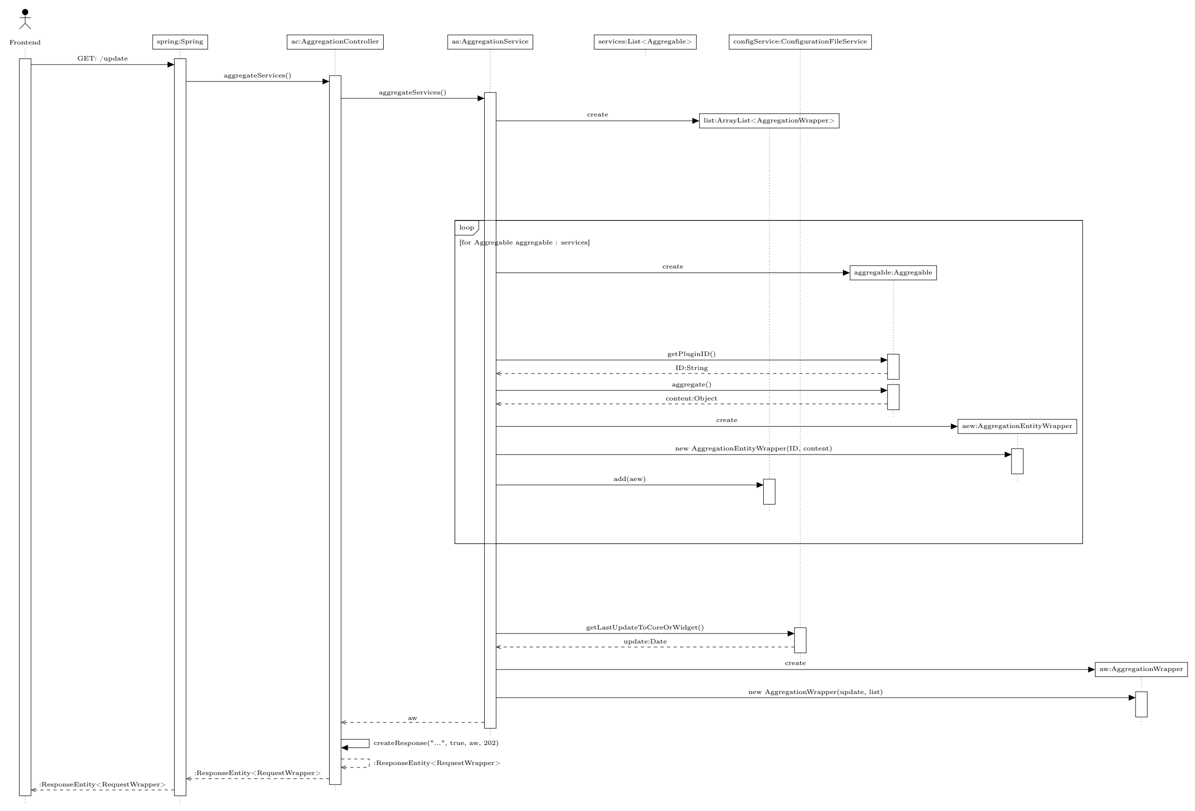
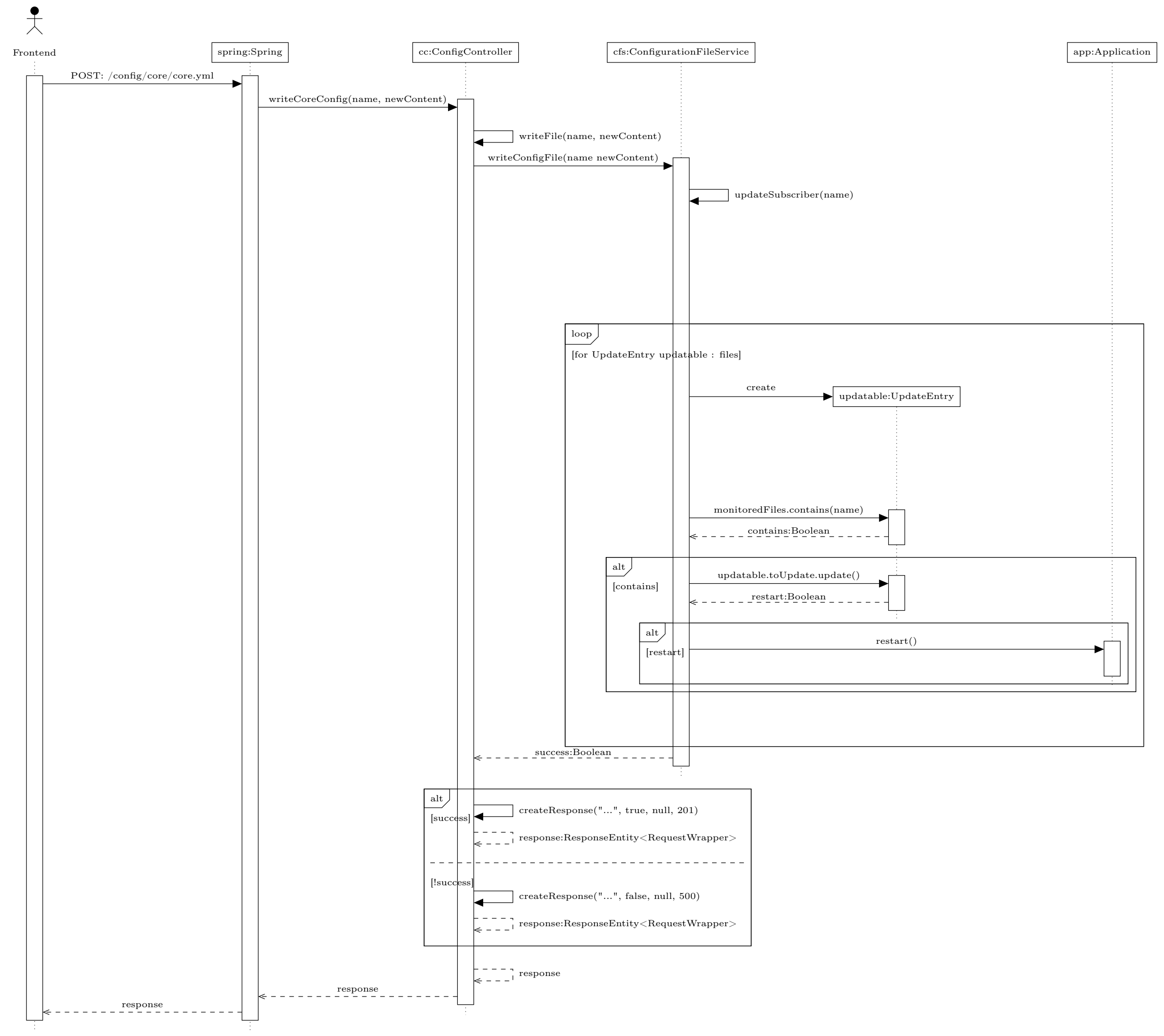

Frontend
spring:Spring
ac:AggregationController
as:AggregationService
services:List<Aggregable>
configService:ConfigurationFileService
GET: /update
aggregateServices()
aggregateServices()
create
list:ArrayList<AggregationWrapper>
loop
[for Aggregable aggregable : services]
create
aggregable:Aggregable
getPluginID()
ID:String
aggregate()
content:Object
create
aew:AggregationEntityWrapper
new AggregationEntityWrapper(ID, content)
add(aew)
getLastUpdateToCoreOrWidget()
update:Date
create
aw:AggregationWrapper
new AggregationWrapper(update, list)
aw
createResponse("...", true, aw, 202)
:ResponseEntity<RequestWrapper>
:ResponseEntity<RequestWrapper>
:ResponseEntity<RequestWrapper>

Figure 10.3: backend: update request

10.1.4 Update core config file

This diagram is an example for updating config files. For other config files this process is very similar e.g. instead of `writeCoreConfig` you would use .



Frontend
spring:Spring
cc:ConfigController
cfs:ConfigurationFileService
app:Application
POST: /config/core/core.yml
writeCoreConfig(name, newContent)
writeFile(name, newContent)
writeConfigFile(name newContent)
updateSubscriber(name)
loop
[for UpdateEntry updatable : files]
create
updatable:UpdateEntry
monitoredFiles.contains(name)
contains:Boolean
alt
[contains]
updatable.toUpdate.update()
restart:Boolean
alt
[restart]
restart()
success:Boolean
alt
[success]
createResponse("...", true, null, 201)
response:ResponseEntity<RequestWrapper>
[!success]
createResponse("...", false, null, 500)
response:ResponseEntity<RequestWrapper>
response
response
response

Figure 10.4: backend: update core config file

10.1.5 Get logs

This diagram shows how the logs are retrieved.

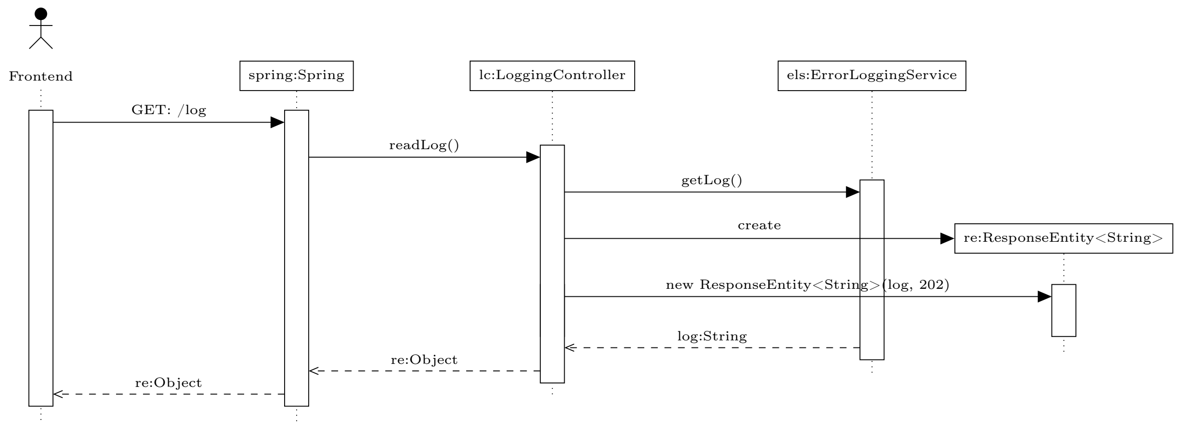


Figure 10.5: backend: get logs

10.1.6 Periodical fetching of publication

This diagram shows the initialization of periodical fetching of publication. The process for submitting any other periodical jobs is very similar.

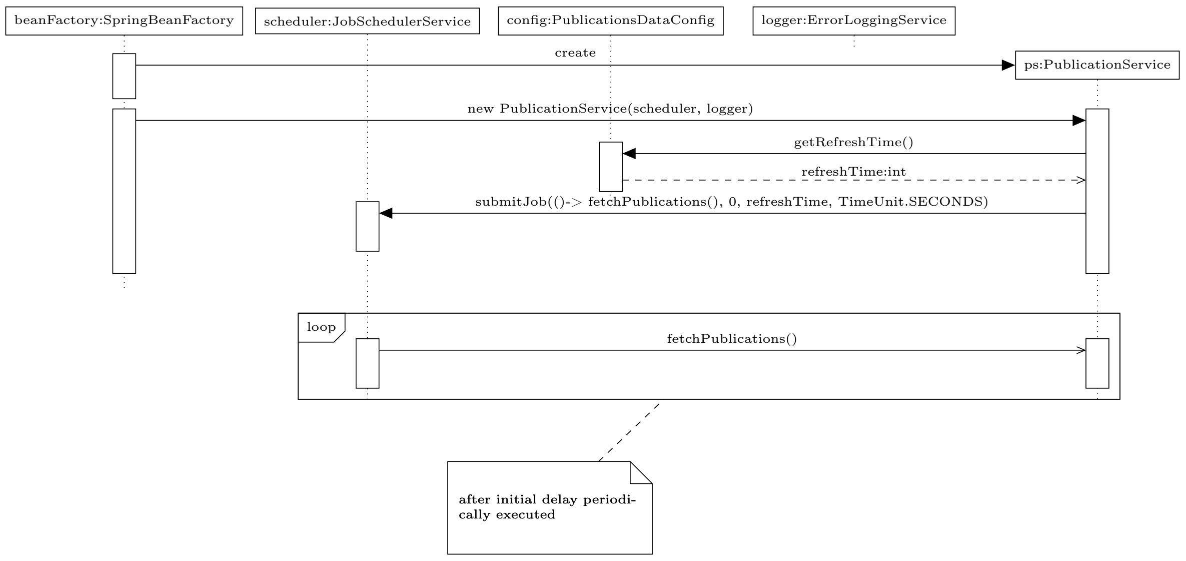

beanFactory:SpringBeanFactory
scheduler:JobSchedulerService
config:PublicationsDataConfig
logger:ErrorLoggingService
create
ps:PublicationService
new PublicationService(scheduler, logger)
getRefreshTime()
refreshTime:int
submitJob(()-> fetchPublications(), 0, refreshTime, TimeUnit.SECONDS)
loop
fetchPublications()
after initial delay periodically executed

Figure 10.6: backend: periodical fetching of publication

10.2 Frontend

10.2.1 Initial startup

The App component loads the plugins config file, loads all active plugins using this as well as their config file and then proceeds to inject the config file into the correct plugin store.

When the component is mounted a timer is started which periodically calls `this.refresh`. This is also done once explicitely because the timer triggers for the first time after the time has elapsed and not initially after starting it.

This behaviour is described in the Refresh data diagram.

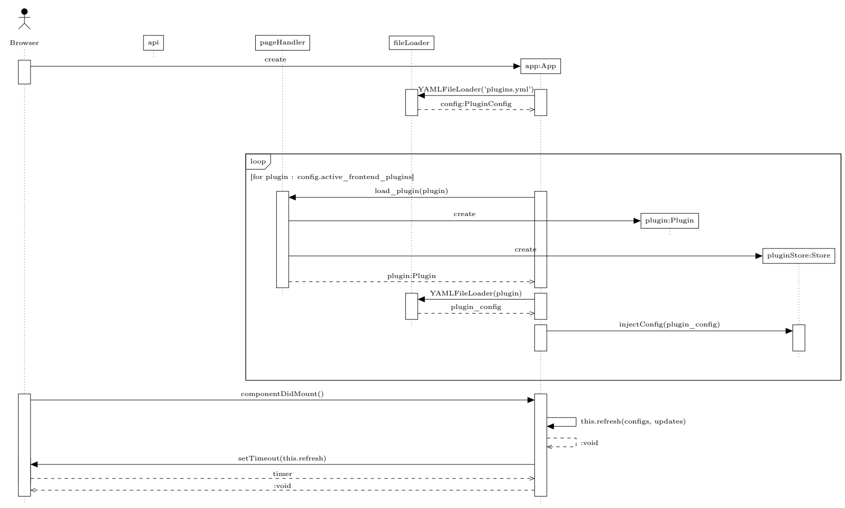


Figure 10.7: frontend: initial startup

10.2.2 Refresh data

This diagram shows how the data collected from the Update is processed. The refresh method is executed periodically.

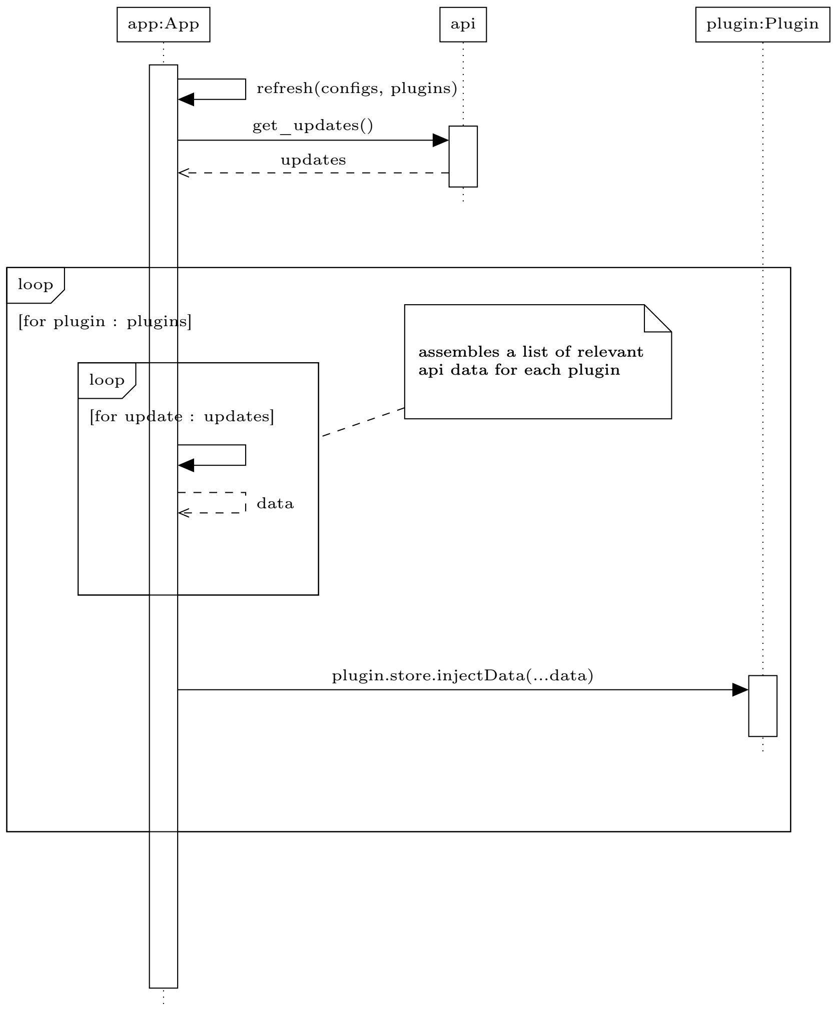

app:App
api
plugin:Plugin
refresh(configs, plugins)
get_updates()
updates
loop
[for plugin : plugins]
assembles a list of relevant api data for each plugin
loop
[for update : updates]
data
plugin.store.injectData(...data)

Figure 10.8: frontend: refresh data

10.3 Dashboard

10.3.1 Initial startup

This diagram shows the initial setup of the dashboard.

The App, UIStore, jwtStrategy, Storage and FileLoader are initialized automatically.

The UIStore begins to load files it needs like the general config file and following that translation files depending on the configured languages.

App is the entrypoint of the application and handles the initialization of the client-side routing/paging system. Part of this is loading all active plugins and adding them to the available routes.

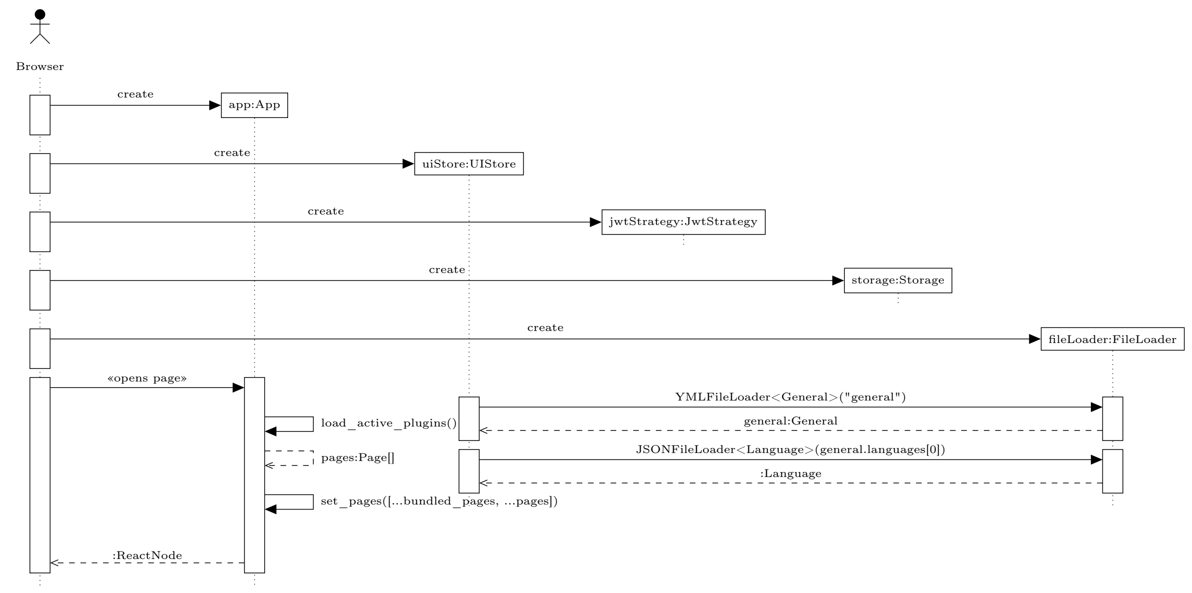


Figure 10.9: dashboard: initial startup

10.3.2 Load "/core" not logged in

This diagram shows the login process and the loading of the “/core” page. The page loading process is very similar for any other dashboard page.

The initial startup has run and all the pages the dashboard can display are now known and available to be shown.

The client side routing determines that “/core” belongs to the CoreConfigurationPage route. This route is private so the PrivateRoute component checks wether the user is authenticated or not and redirects the user to the login page.

After logging in the user is now authenticated and authorized to view the CoreConfigurationPage which is thereafter rendered.

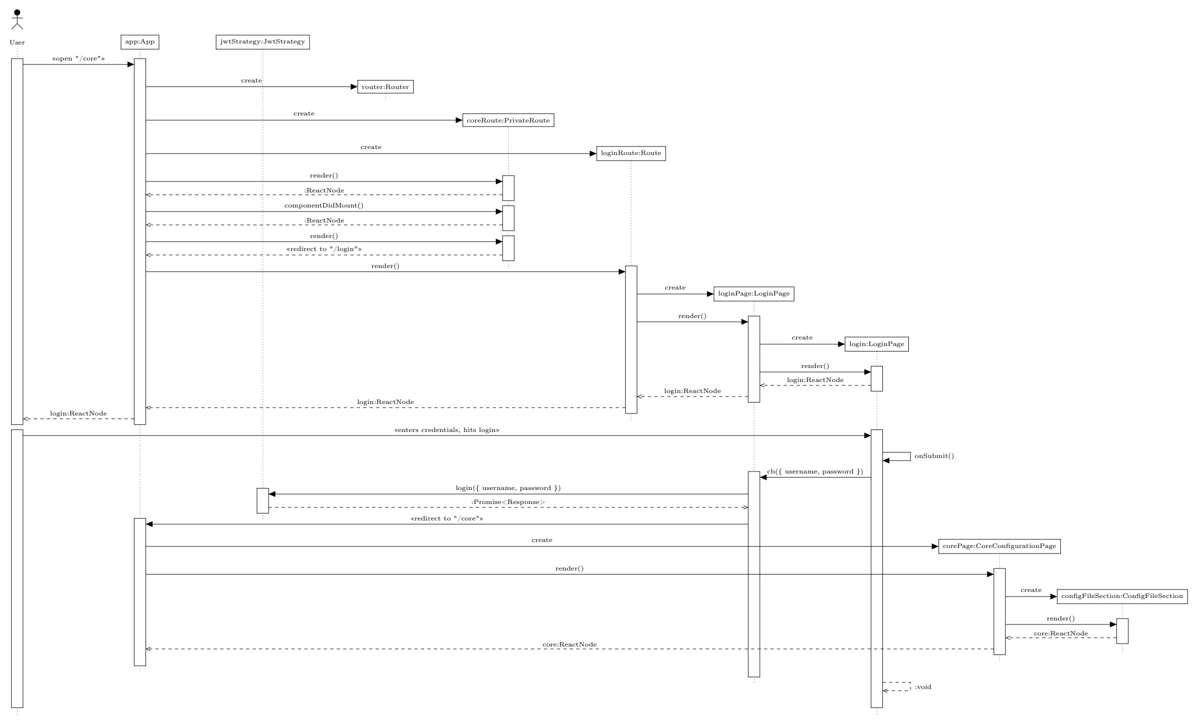


Figure 10.10: dashboard: load “/core” while not logged in

10.3.3 Upload "general" config file

This diagram shows how the “general” config file is uploaded and updated. To upload any other config file you just have to change the parameters containing “general” and sometimes “core” if the type of configuration differs.

When the user clicks upload and picks a file the contents of the file are read and then uploaded. To upload the core configuration page asks the jwtStrategy for the jwt and uploads the file using it afterwards.

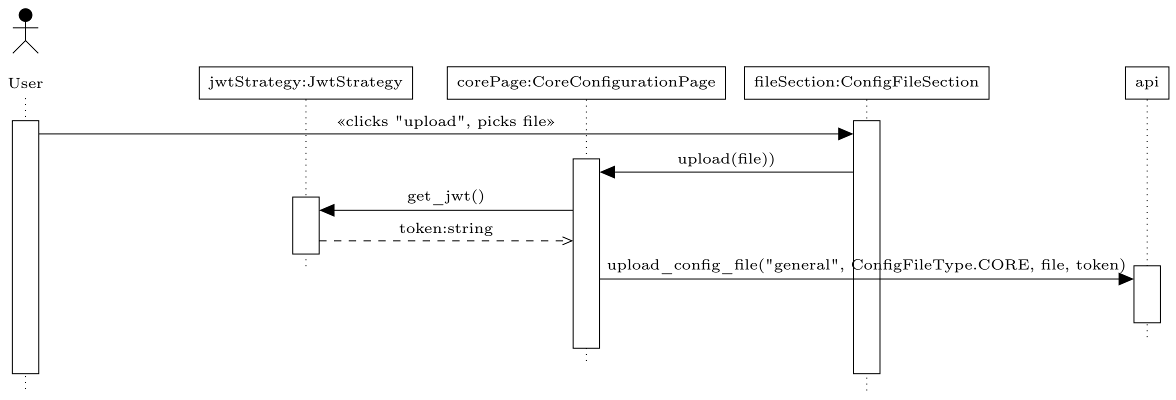

User
jwtStrategy:JwtStrategy
corePage:CoreConfigurationPage
fileSection:ConfigFileSection
api
«clicks "upload", picks file»
upload(file))
get_jwt()
token:string
upload_config_file("general", ConfigFileType.CORE, file, token)

Figure 10.11: dashboard: upload “general” config file

10.3.4 Download "general" config file

This diagram shows how the “general” config file is downloaded. To download any other config file you just have to change the parameters containing “general” and sometimes “core” if the type of configuration differs.

This is almost the same as uploading a file, just using differently named methods, functions and api endpoints.

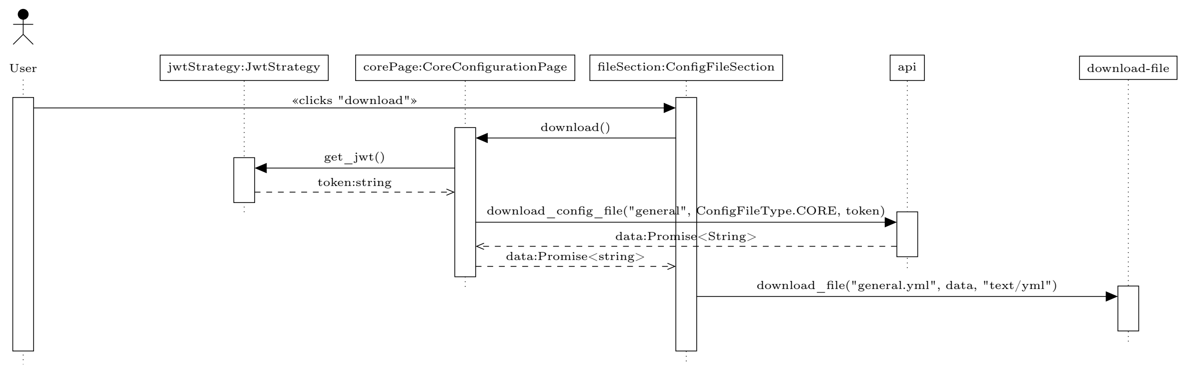

User
jwtStrategy:JwtStrategy
corePage:CoreConfigurationPage
fileSection:ConfigFileSection
api
download-file
«clicks "download"»
download()
get_jwt()
token:string
download_config_file("general", ConfigFileType.CORE, token)
data:Promise<String>
data:Promise<string>
download_file("general.yml", data, "text/yml")

Figure 10.12: dashboard: download “general” config file

10.3.5 Create announcement

This diagram shows the creation of an announcement through the dashboard. The process of modifying any other data is very similar.

When the user enters information about the announcement this is synchronized with the internal state of the component. Once the create button is pressed the internal state is used to create the announcement.

For this to work the announcement page asks the jwtStrategy for the JWT and makes an api call using it afterwards.

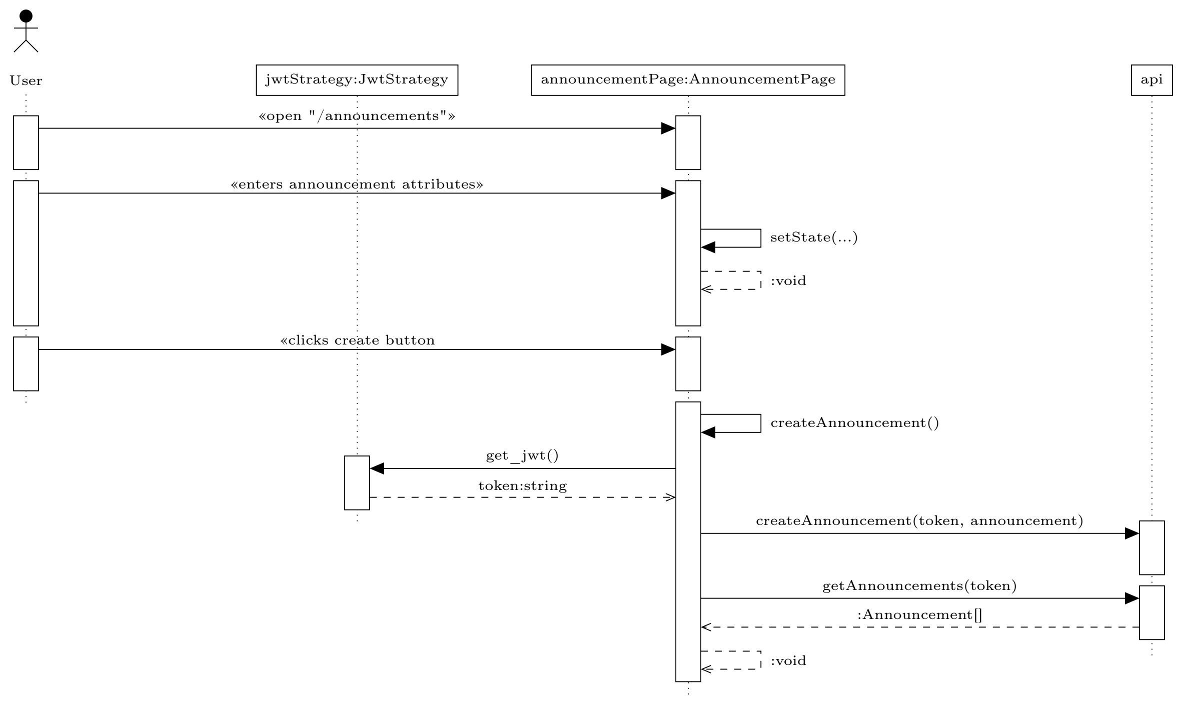


Figure 10.13: dashboard: create announcement

11 Requirements Changes

All mandatory functional requirements are covered by the design with class diagrams and sequence diagrams.

The following optional requirements are either covered completely, or fit into existing classes and structures

- None at the moment

All other optional requirements have not been accounted for in this document. However, the current design allows for easy extensibility to accomodate all optional requirements in the future.

The following has changed throughout the requirements document:

- **GTC-6**: An admin adds a website to track for publications
  This action was removed, because the widget will automatically display all publications.
  "Action: Upload a config file via the upload button beneath the label Publications widget. In the file the newly added publications source is set to active."

- **GTC-9**: An admin configures the Cafeteria widget
  The cafeteria data will be fetched via an API instead of parsing a website.
  "The file should contain a website from which the data can be fetched."
  has been changed to:
  "The file should contain the APIaddress, APIkey and APIUserName of the API from which the data can be fetched."

  "In the file the newly added Cafeteria menu source is set to active."
  has been changed to:
  "In the file the newly added Cafeteria is set."

12 Project Schedule

12.1 Overview

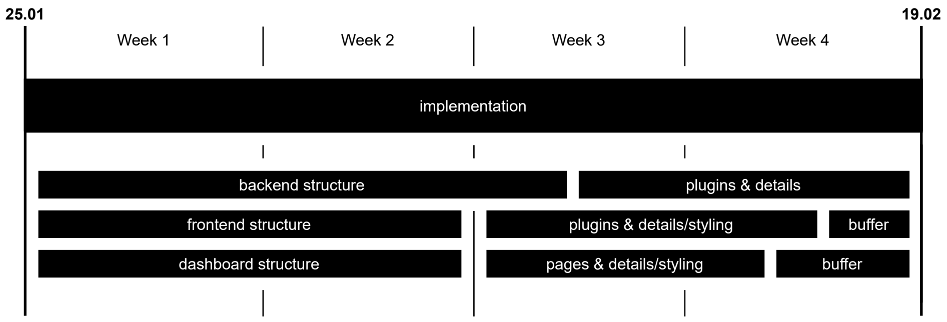


Figure 12.1: Project schedule overview

12.2 Frontend

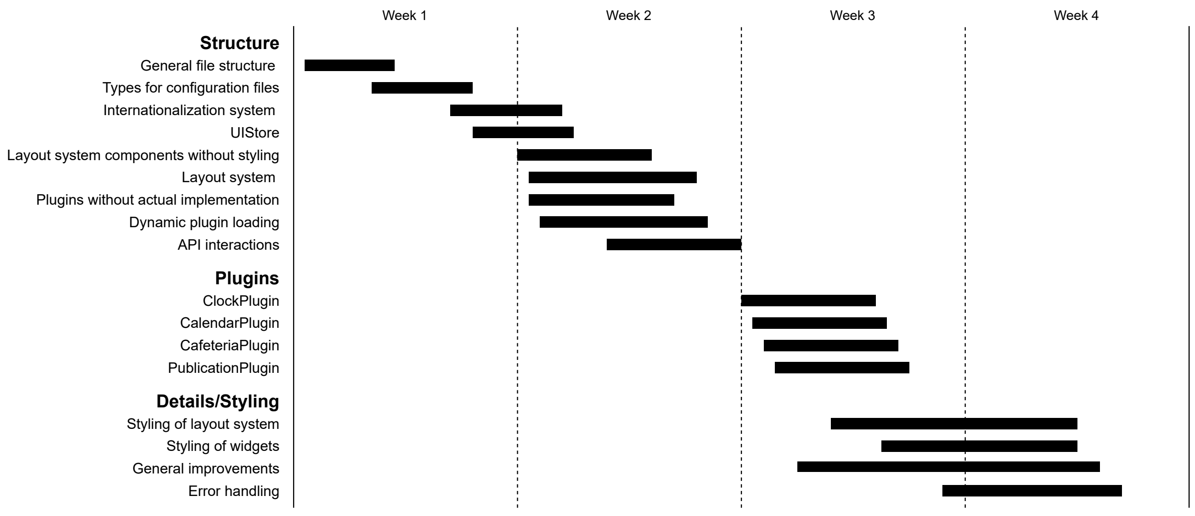


Figure 12.2: Detailed frontend schedule

12.3 Backend

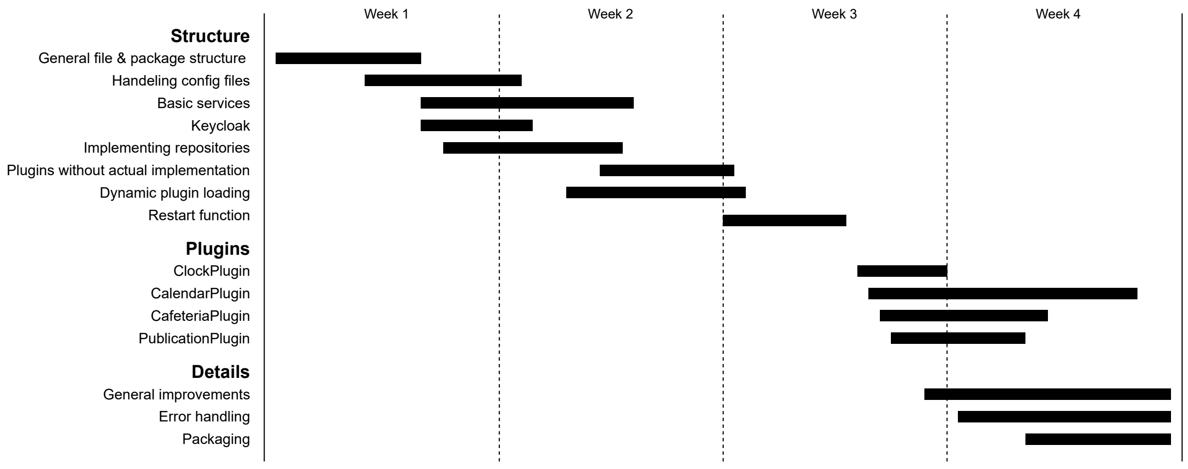


Figure 12.3: Detailed backend schedule

12.4 Dashboard

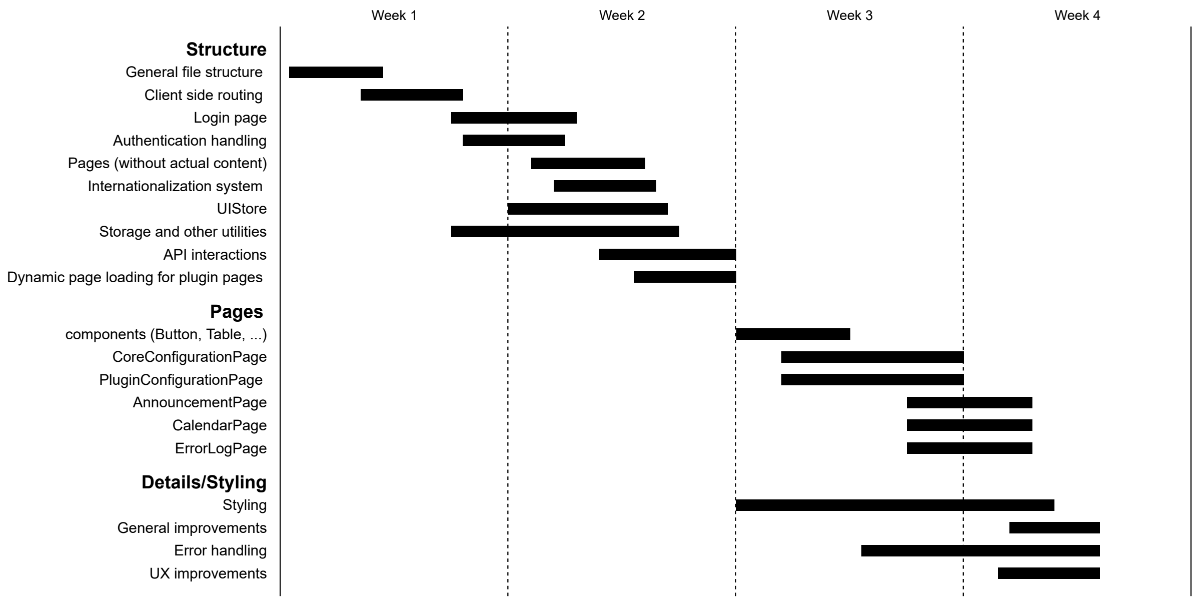


Figure 12.4: Detailed dashboard schedule

13 Libraries and Frameworks

The following libraries and frameworks are used by the project.

13.1 Frontend and Dashboard

- **typescript**
  Responsible for transpiling TypeScript to JavaScript.
- **io-ts**
  Runtime type system for IO decoding/encoding. Used to ensure type safety of config files at runtime.
- **fp-ts**
  Peer dependency of io-ts.
- **ts-brand**
  Type Branding. Used to achieve nominal typing where it is useful.
- **react**
  Used for rendering everything.
- **react router**
  Client-side routing.
- **mobx**
  State-management library.
- **mobx-react**
  Mobx integration for react.
- **react-awesome-toast**
  Library for showing *toast* UI components (see this for an explanation).
- **antd**
  UI component library.
- **yaml**
  YAML parser used for parsing the config files.

13.2 Backend

- **spring boot**
  Framework for building web applications. Used for API endpoints.
- **keycloak**
  Spring boot adapter for integrating with keycloak.
- **jsoup**
  HTML parser used for parsing publications and other external resources..
- **antlr**
  Parser generator used for parsing iCal.
- **snakeyaml**
  YAML parser used for parsing config files.
- **unirest**
  HTTP request library used for making HTTP requests to external resources.
- **JSON-java**
  Json handling library used to process the data given by the cafeteria api.

These lists might need to be extended in the future to accomodate for unit testing or other changes.

Glossary

API An “application programming interface” is an interface with other services like getting weather information for a given location.

backend The API which runs on the server and serves the frontend.

classPath The java classPath contains all compiled class files, that can be loaded and executed by the JVM.

config file A config file is a file which contains properties that describe some sort of setting in a formal and orderly way according to a strict specification. This project relies on multiple config files.

frontend The webapp running on the smart TV.

IDP An Identity Provider is a service that handles authentication and authorization of users.

JVM Any executed java code runs inside of a Java Virtual Machine (JVM).

keycloak keycloak is an IDP.

mobx store mobx is a state management library for JavaScript. It has the concept of a store which houses data and has methods which manipulate that data. It is the single source of truth when it comes to the data it manages and is the only code which modifies it. When a method is called and the data is updated all observers are notified of the update. It is most oftenly used alongside react.

react react is a JavaScript library for buildling user interfaces. It is based on the concept of `Components` which represent UI elements and are then build up into more and more complex UI elements *like Lego pieces*. React is declarative.

react-router react-router is a routing library for react.

REST Stands for Representational state transfer it is a standard for building api endpoints. REST.

server The server running the backend.

smart TV The television for which the webapp is optimized.