EVT 3 Documentation

Table of contents

Using EVT

If you are interested in **using** EVT 3 to prepare an edition right away, you should probably download the ready-to-use release package. See the Installation and use section first, then Configuration to understand how EVT works and how you can use it to publish your editions.

If, on the other hand, you are interested in contributing to the main project, by fixing a bug or adding/modifying a feature, please refer mainly to the Development section to know how things work and to learn more about our development workflow. Note that some technical details are sometimes given in other sections, too.

If you intend to change the source code for personal needs, please **fork** the project and/or contact us at evt.developers@gmail.com. If you want to implement a new feature, or improve an existing one, best results are achieved when there is a dialogue with the developers, especially to avoid duplication of efforts and/or to keep the customized code in sync with general development.

Installation & use

EVT 3 can be used to prepare an edition right away, immediately after downloading the release package on your hard drive: see the *Installation and management of the edition data* section first, then *Configuration*, to understand how EVT works and how you can use it to publish your editions.

If, on the other hand, you are interested in **developing** a specific functionality in EVT 3, or in modifying an existing one, we suggest that you clone the project as it is. The `README.md` explains how to install and configure the development environment needed for this purpose. This step is only needed if you want to start working with EVT source code, so it is in no way necessary for basic users. See Development section and subsections for further information on development guidelines.

Installation and management of the edition data

Installation is quite simple, in fact it is not an actual installation in the traditional sense: you just need to download the compressed archive from the release page (or the EVT home page), unzip it in a suitable location on your hard drive, and you are ready to use it with your edition files. Within the main folder there are only two folders which should be modified by the user:

- `assets/config`: here you will find four different configuration files which can be used to properly configure EVT as needed (see the Configuration section for further details;
- `assets/data`: here you will put all of your edition data, including the TEI-encoded documents, images, and other edition files.

Everything else should not be modified, unless you know what you are doing very well. It is in fact possible to modify the JavaScript parsers, but doing so directly in an EVT release is very difficult, because everything is minified and uglified for performance reasons, and also less efficient than doing it on the development version. Since EVT is an open source tool, you are welcome to fork it, change the existing parsers and/or add your own parsers and eventually open a Pull Request so that your changes will be integrated in the main version of EVT.

Before moving to the configuration, you should have the different edition components ready. As you will see during configuration, the paths to the resources are completely configurable, thus there is no strict obligation to follow the default structure to organize your files;

however we would like to suggest and recommend the default structure to you as a specific way to organize your files, one which will allow you to keep the different contents well separated from each other.

In the `assets/data` folder you can create one folder for each type of data (images, text, etc.). For example:

- `data/text` => put your textual data here, possibly further organized in subfolders such as `documents`, `schema`, `sources`, `witnesses`, etc.
- `data/images` => put your images here, you will find some sub-folders (e.g. data/images/single, data/images/hotspot etc.), create more if needed
- `data/models` => put your 3D models here, again you will find the `multires` and `singleres` subfolders.
- `data/viscoll` => put all VisColl-related files here.

To have your edition parsed and loaded in the browser by EVT you have to point to it explicitly modifying the file_config.json file in the config directory and specifying the name of the main file: `"dataUrls": ["data/text/My_edition.xml"]`. While this is the most important configuration option, since it tells EVT where to start with your edition, note that there are several other options available in that file, so that you can customize the layout and appearance of your edition (see the (Configuration)[https://github.com/evt-project/evt-viewer-angular/wiki/Configuration] section). Also note that some configuration options may be necessary to make desired features available, for instance to add a required edition level, so make sure you read the following section and check the default configuration file.

Open your edition

In order to locally access your edition (for test/study purposes, before publishing it on a web server) you need to enable local files access in your browser. In fact, browsers such as Chrome, Firefox (since v. 67), Safari, etc., have adopted a security-conscious policy that forbids loading local files (= documents available on the user's computer drive) in the browser as a result of the execution of JavaScript programs. The goal is to improve global security when browsing the Web, but the unpleasant collateral effect is that of preventing the loading of digital editions based on EVT, or similar software, from local folders. Fortunately there are several workarounds that can be used to test EVT editions that are located on your hard drive:

- option no. 1: close every window of Chrome and launch it from the command line with the `--allow-file-access-from-files` parameter; then open the `index.html` file;
  - Windows

    ```
    cd C:\Program Files (x86)\Google\Chrome\Application
    chrome.exe --allow-file-access-from-files
    ```

  - Linux

    ```
    google-chrome --allow-file-access-from-files
    ```

  - MacOSX

    ```
    open -a "Google Chrome" --args --allow-file-access-from-files
    ```

- option no. 2: download and install Firefox ESR v. 60: this version predates the new security policy adopted in FF v. 67 and, furthermore, it can be installed in parallel with any other version of Firefox;
- option no. 3: install an extension providing a local web server on Firefox or Chrome, f.i. there is this one available for Chrome.

Note again that this problem, however, only affects local testing: after the edition has been uploaded on a server there are no problems in accessing it with any of the major browsers.

Configuration

There are several configuration options, ranging from setting the folders where edition data is stored to choosing the User Interface layout and the tools to be made available for the final user, that can be set by editing the configuration files in the `assets/config` directory. To facilitate the configuration work, configuration options are divided into three macro groups:

- Edition Configuration (`edition_config.json`), where to set the configurations closely related to the digital edition, such as the title, the edition level(s), etc. See details here.
- File Configuration (`file_config.json`), where to set the path(s) to the file(s) of the digital edition. See details here.
- UI Configuration (`ui_config.json`), where to set the configuration closely related to the UI, such as the default language, the default/available theme(s), etc. See details here.

It is also possible to configurate the style of editorial phenomena (e.g. addition, deletion, etc), in order to override the EVT default layouts. This particular configuration should be defined in the file `editorial_conventions_config.json`. See details here

Technical details for development

Configuration is defined as a `AppConfig` provider and is injected into the main app module. It is loaded during app initialization, so that it will be immediately available for every component. The three groups are gathered (although they will be kept divided) in a single `EVTConfig` object.

```
interface EVTConfig {
    ui: UiConfig;
    edition: EditionConfig;
    files: FileConfig;
}
```

If you want to use a parameter from configuration in your component, you just need to import `AppConfig` and directly use its properties:

```
import { AppConfig } from '../app.config';
@Component({
  selector: 'my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.scss']
})
export class MyComponent {
    private editionTitle: string;
    constructor() {
        this.editionTitle = AppConfig.evtSettings.edition.title;
    }
}
```

Files

File Configuration (`file_config.json`) allows to set the path(s) to the file(s) of the digital edition.

- `editionUrls: string[]`. String array which holds the list of paths to the XML file(s) of your encoded edition. Each path can point either to an internal folder or to an external online resource.
- `editionImagesSource: { manifest: { value: string; enable: boolean; }; graphics: { value: string; enable: boolean; }}`. Indicate the source to be used to retrieve images. If you have a manifest, indicate the IIIF path to manifest.json as value and set `manifest.enable` to `true`. If you want to use information encoded within your XML `<graphic>` element, indicate the attribute in which path to file is encoded and set `graphics.enable` to `true`. If both `manifest.enable` and `graphic.enable` are set to `false`, images path will be guessed starting from pages ids, as it was for EVT1 and EVT2.
- `manifestURL: string`. Allows to write the URL of an IIIF `manifest.json` file to publish a digital facsimile.
- `logoUrl: string`. You can add a custom logo that will appear before the edition title: just indicate the path to it; it can be a URL or a relative path: we suggest that you put it into `data` and point to it (f.i. `"logoUrl": "data/icons/myLogo.jpg"`). If you don't want a logo at all just leave an empty string `"logoUrl": ""`.
- `imagesFolderUrl?: string`. Path to the folder containing all images. Optional.
- `configurationUrls?: { edition: string; ui: string; editorialConventions: string; }`. Paths to the other configuration files. Optional.

[WIP]

Edition

Edition Configuration (`edition_config.json`) allows to set the configuration options closely related to the digital edition, such as its title, the edition level(s), etc.

Edition main information

- `editionTitle: string`. Choose a title for your edition. If you want to allow the translation of the title, use the proper key of the locale json file. If you leave it blank the title from `<titleStmt>` or, if this is empty, the default 'EVT Viewer' title will be shown.
- `badge: string`. The badge text to be visualized following the edition title, it should be a short text string such as `alpha`, `experimental`, etc.
- `editionHome: string`. If you specify an external web site there will be a link pointing to it.

Edition levels

- `availableEditionLevels: EditionLevel[]`. Select which edition levels you want to be available in your edition. You can deactivate a view mode both by deleting it and by setting to `false` the property `enable` (the latter being the suggested method).
  - Each edition level should be indicated as follows:

```
{
  id: 'diplomatic' | 'interpretative' | 'critical';
  label: string;
  title?: string;
  enable?: boolean;
}
```

- `defaultEdition: 'diplomatic' | 'interpretative' | 'critical'`. Select which edition level you want your edition to open on. Note that it must be an active edition level!

View modes

- `availableViewModes: ViewMode[]`. Select which view modes you want to be available in your edition. You can deactivate a view mode either by deleting it or by setting the property `enable` to `false` (suggested method).
  - Each view mode should be indicated as follows:

```
{
  id: 'readingText' | 'imageText' | 'textText' | 'collation' | 'textSou
rces' | 'textVersions';
  icon: string;
  iconSet?: 'evt' | 'far' | 'fas';
  label: string;
  enable?: boolean;
}
```

- `defaultViewMode: 'readingText' | 'imageText' | 'textText' | 'collation' | 'textSources' | 'textVersions'`. Select which view mode you want to your edition to open on. Note that it must be an active mode!

Global tools

- `showLists: boolean`. Indicate whether button for lists (named entities, relations, events, etc.) should be available or not.

Named entities

- `namedEntitiesLists: Partial<NamedEntitiesLists>`. Customize the list of available named entities to be highlighted and to be shown among entities lists, by changing the default list label (to be shown when no `<head>` is defined for the list) and/or by deleting/deactivating one or more element in the list. At the moment EVT can work only with list of person (`<listPerson>`), list of places (`<listPlace>`), list of organizations (`<listOrg>`), list of events (`<listEvent>`) and list of relations (`<listRelation>`). If you need a new kind of named entity to be handled just notify the EVT Development Team.
  - This parameter could be defined as follows (none of the parameter is mandatory).

```
{
    persons: NamedEntitiesListsConfig; // for list of persons `<listPer
son>`
    places: NamedEntitiesListsConfig; // for list of places `<listPlace
>`
    organizations: NamedEntitiesListsConfig; // for list of organizatio
ns `<listOrg>`
    relations: NamedEntitiesListsConfig; // for list of relations `<lis
tRelation>`
    events: NamedEntitiesListsConfig; // for list of events `<listEvent
>`
}
```

    where `NamedEntitiesListsConfig` is an object defined as follows:

```
{
   defaultLabel: string;
   enable: boolean;
}
```

- `entitiesSelectItems: EntitiesSelectItemGroup[]`. Customize the list of available entities to be highlighted by adding a new element to this list: for each element you should define a tagName, which is the XML tag that identify the entity, a label that will be shown in the selector and a color that will be used to highlight the entity within the text. If you don't need an entity that is already inserted in this list you can delete it or just use the property `enable` set to `false` (suggested choice).
  - Each item group should be defined as follows:

    ```
    {
        label: string;
        items: EntitiesSelectItem[];
        enable?: boolean;
    }
    ```

  - And each item should be defined as follows:

    ```
    {
        label: string;
        value: string;
        color?: string;
        enable?: boolean;
    }
    ```

    the `value` will be used to identify the items to be selected, thus you should indicate the tag name and eventual attributes in the form `tagName[attribute='value']` (e.g. `"persName[type='episcopus']"`). Multiple values should be divided by a comma (e.g. `"placeName,geogName"`).

Prose/Verses

- `proseVersesToggler: boolean`. Indicate whether to activate or not the button to toggle text from prose flow to verses flow.
- `defaultTextFlow: 'prose' | 'verses'`. Indicate which flow you want to activate for the text as default.
- `verseNumberPrinter: number`. Indicates the multiplier for the visualization of verses numbers.

Critical edition

- `notSignificantVariants: string[]`. Array of string that should be used to identify not significant variants: element(s) of attribute(s) you used to encode variants that are not significant and you do not want to appear in the main critical apparatus (f.i. , or @type=orthographic). Please divide values using commas.

[WIP]

Editorial Conventions Configuration

Editorial Conventions Configuration (editorial_conventions_config.json) allows to set configurate the style of publishing phenomena (e.g. addition, deletion, etc), in order to override the EVT default layouts.

For each particular phenomena you should choose an identifier that will help you to better orient yourself in the configuration file itself and define an element with very specific properties.

```
{
    "additionAbove": {
        /* configuration for addition above */
    },
    "additionBelow": {
        /* configuration for addition below */
    },
}
```

In particular, every configuration element should present the following properties:

```
interface CustomEditorialConvention {
    markup: {
        element: string;
        attributes: { [key: string]: string; };
    };
    layouts: {
        [key in EditionLevelType]: {
            style: { [cssProperty: string]: any; };
            pre: string;
            post: string;
        };
    };
}
```

The `markup` identifies the element depending on its encoding (tag name and map of attributes), while the `layouts` indicate the output style to be assigned for the indicated encoding for each edition level. In this case you can define:

- `style` can be used to define a list of CSS properties to be assigned to the output element. These rules will be merged with the default ones.
- `pre` can be used to indicate the text to be shown before the element

- `post` can be used to indicate the text to be shown after the element None of these properties are mandatory, thus it is possible to define a single specific configuration, for example only some text to be shown before the text.

If you do not indicate a specific layout for a specific edition level, the default EVT layouts will be used.

Note that for the moment this configuration will only works with specific elements:

- additions (`<add>`)
  - above (`<add place="above">` or `<add place="sup">`)
  - below (`<add place="below">` or `<add place="under">` or `<add place="sub">`)
  - inline (`<add place="end">` or `<add place="inline">` or `<add place="inspace">`)
  - left (`<add place="left">`)
  - right (`<add place="right">`)
- damages (`<damage>`)
- deletions (`<deletions>`)
- original part of text deemed incorrect (`<sic type="crux">`)
- superfluous or redundant part of text (`<surplus>`)

This list will grow as the development of publishing phenomena goes on. If you want to style an element that does not appear in this list you should use a custom CSS rules See details here.

Example

Let's say you want to change:

- the style of `<add place="above">` so that in the diplomatic edition they will figure 10px above the baseline and in the interpretative edition it is shown between the characters `"\"` and `"/"` and has a yellow background.
- the style of generic deletions `<del>` so that in the diplomatic edition they will be hidden in the interpretative edition. Your `editorial_conventions_config.json` will be:

```
{
  "additionAbove": {
      "markup": {
          "element": "add",
          "attributes": {
              "place": "above"
          }
      },
      "layouts": {
          "diplomatic": {
              "style": {
                  "top": "-10px",
                  "position": "relative"
              }
          },
          "interpretative": {
              "pre": "\\",
              "post": "/",
              "style": {
                  "background-color": "#ffcc00"
              }
          }
      }
  },
  "deletions": {
      "markup": {
          "element": "del"
      },
      "layouts": {
          "interpretative": {
              "style": {
                  "display": "none"
              }
          }
      }
  }
}
```

Style customization

Work in progress...

Publish an EVT edition on GitHub

Prerequisites

- a GitHub account
- the git command-line program on your computer. See this tutorial about how to install git on your computer: https://github.com/git-guides/install-git.
- a release version of EVT or a custom build of the EVT development app.

Steps

New repository creation

- Once you've signed in in GitHub, you'll create a new repository to get started (Add > New repository).
- On the new repository screen, you need to give this repository a special name which will also be used to generate your website. Your website's files will live in a repository named `username.github.io` (where "username" is your actual GitHub user name). You can also use a different name; in this case the final link to your GH Page will be slightly different (see below).
- Then you need to set up your repository.

Repository and GH Pages Setup

If you already have a local version of your edition with EVT, you can follow the steps below:

- from the terminal navigate to the main folder of your edition (the one containing the `index.html`)
- edit the `README.md` so that it describes your edition
- then launch the following commands:

```
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/evt-project/evt-demo.github.io.git
```

  The reference to the repository will depend on the name you choose

```
git push -u origin main
```

- Now, navigate to repository in GitHub and cli open the "Settings tab". If you scroll down on the settings page, you'll see the *GitHub Pages* section near the bottom.
- Select the branch `main` as the source and save. Once saved you will be prompted with a message showing the link to your site (note that if you choose the same name of your GH account the link will be shorter).
- Now if you navigate to that link you will see the text of the README you saved and pushed before.
- Go back to the terminal and launch the following commands:

  ```
  git add .
  ```

  The `"."` is used to add every file that has changed. If you want to add a specific file, just indicate its path; you can also add more than one file by performing multiple `git add path/to-file.fileExtension` before the next command.

  ```
  git commit -m "Add EVT edition"
  ```

  You can customize the message as you prefer

  ```
  git push
  ```

Wait one minute at least to give GH the time to update its cache, then reload the GH Page of your project, and you will have your edition published on the Web!

Update your EVT edition

To update the remote version, every time you perform a change to a file, just launch the three steps above (`git add`, `git commit` and `git push`) and the online version will be automatically updated.

References

- GitHub Pages - https://pages.github.com/
- Getting Started with GitHub Pages - https://guides.github.com/features/pages/

Development

Requirements

For development, you will only need Node.js installed on your environment. We also strongly suggest that you use the appropriate TSLint plugin for your Editor (not mandatory).

Node

To install Node, just follow the indication on http://nodejs.org/; it is really easy to install and now it includes NPM installation. After the installation procedure, you should be able to run the following command to check if it was installed correctly (note that versions may differ):

```
$ node --version
v10.15.3

$ npm --version
6.4.1
```

Install development environment for EVT

```
$ git clone https://github.com/evt-project/evt-viewer-angular.git
$ cd evt-viewer-angular
$ npm install
```

Start & watch with development server

```
$ npm run start
```

If nothing happens, then open your browser and navigate to `http://localhost:4205/`. The app will automatically reload if you change any of the source files.

You can also choose the port where to run the application by running `npm run start -- --port=4202` (in this case you should then connect to `http://localhost:4202/`).

Code scaffolding

To generate new components, services, directive, etc you can use Angular CLI. Run `npm run ng generate component component-name -- --module=app.module` to generate a new component. You can also use `npm run ng generate directive|pipe|service|class|guard|interface|enum|module`. You should always add the flag `--module=app.module` since there are multiple modules. You can also indicate the path before component name: `npm run ng generate component path-to-components/component-name -- --module=app.module`

Linting

A very strict linting rule set has been configured for this app. Please run `npm run lint` and correct any indications before any commit.

Please read the following subsection before starting to contribute to the code

- Contribute to the code
- Parsing
- Localization
- Color-theme
- Main-style
- Sample documents

Contribute to the code

Editor

The use of VSCode is strongly recommended. Please check that you have installed and enabled at least the following plugins:

- Angular language service
- Angular template formatter con “close tag same line” impostato
- Beautify css/sass/scss/less
- Npm
- Npm intellisense
- TSLint

Programming style

The code written by each developer must be clear, well formatted and above all well commented, in such a way as to explain its function to those who will have to hand over that specific portion of code. Comments, such as log messages on GitHub and any documentation accompanying the code itself, must be in English.

Commit Messages

For commit messages each developer must follow the guidelines described on https://chris.beams.io/posts/git-commit/. This style of commit message has several advantages and helps keep the repository history clean.

Commit messages that do not follow the chosen style will be rejected (`rebase` is your friend to rename commits). This limits or completely excludes (based on how strictly the guidelines apply) the presence of insignificant commits such as "commit to branch", "I recovered my pc, I commit to checkout".

Branching strategy

Git-flow is used as branching strategy (https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow, https://danielkummer.github.io/git-flow-cheatsheet/)

This strategy has significant advantages:

- it keeps the history of the repository clean and readable, facilitating the identification of

points where bugs appear or the rollback to previous versions;

- the history of the release versions is immediately identifiable;
- it makes the addition of new features and bug fixes systematic.

Merge strategy

Merging of a branch to `master` and `develop` is prevented. The merge on `develop` is managed through the pull request mechanism.

The strategy of merge on `master` should be done only at the time of a new release and only by the managers of the pull request merge.

This mechanism allows to check what goes into develop and to do a minimum of code review (code quality control). This increases the stability of the software, decreases the presence of bugs and increases the overall knowledge of the code by the entire development team.

Merge should always be done not fast forward to keep track of all codebase changes.

Commandments

1. You will have no other language than English for each part of the repository: code, comments, commits, branches, issues, public documentation, README
2. You will comment the code with complex logic explaining what it does, not how it does it
3. You won't commit or push code that doesn't compile
4. You will not use names for variables that do not indicate their function
5. You will write test code for non-trivial components
6. You will check the code for runtime errors before committing or pushing it (the only exception concerns test code that must be rebased before the pull request)
7. You will rebase your branch on `develop` before the pull request

Parsing

After reading the source file indicated in the proper configuration parameter, EVT parses the structure of the edition. At the moment, everything is based on pages (this will probably change when we will add the support for critical edition and pageless editions). A **page** is identified as the list of XML elements included between a `<pb/>` and the next one (or between a `<pb/>` and the end of the node containing the main text, which is the `<body>` in the case of the last page).

Each page is represented in the EVT Model as a `Page`:

```
interface Page {
    id: string;
    label: string;
    originalContent: OriginalEncodingNodeType[];
    parsedContent: Array<ParseResult<GenericElement>>;
}
```

The content of each page is therefore represented as an array of object retrieved by parsing the original XML nodes.

After parsing the structure, for each page identified, we then proceed to parse all the child nodes, by calling the `parse` method of the `GenericParserService`.

Parsers are defined in a map that associates a parser with each supported `tagName`. This map is retrieved by the generic parsing function which chooses the right parser based on the node type and its `tagName`. If a tag does not match a specific parser, the `ElementParser`, which does not add any logic to the parsing results, is used. Tags and parsers are divided by belonging TEI module.

```
type AnalysisTags = 'w';
type CoreTags = 'lb' | 'note' | 'p';
type SupportedTagNames = AnalysisTags | CoreTags;

const analysisParseF: { [T in AnalysisTags]: Parser<XMLElement> } = {
    w: createParser(WordParser, parse),
};

const coreParseF: { [T in CoreTags]: Parser<XMLElement> } = {
    lb: createParser(LBParser, parse),
    note: createParser(NoteParser, parse),
    p: createParser(ParagraphParser, parse),
}
const parseF: { [T in SupportedTagNames]: Parser<XMLElement> } = {
   ...analysisParseF,
   ...coreParseF,
}

export function parse(xml: XMLElement): ParseResult<GenericElement> {
    if (!xml) { return { content: [xml] } as HTML; }
    // Text Node
    if (xml.nodeType === 3) { return createParser(TextParser, parse).parse(xml); }
    // Comment
    if (xml.nodeType === 8) { return {} as Comment; }
    const tagName = xml.tagName.toLowerCase();
    const parser: Parser<XMLElement> = parseF[tagName] || createParser(ElementParser, parse);

    return parser.parse(xml);
}
```

The generic parsing function (`parse`) to be used to parse the children of a specific node is passed to the parser as a parameter (**NB:** it is not retrieved as an import to avoid running into circular dependencies if the individual parsers are defined in different files).

The return type of each parser is defined as follows:

```
type ParseResult<T extends GenericElement> = T | HTML | GenericElement | Attributes | Description | AttributesMap;
```

When handling the content of a node, the basic idea is: when I don't know what there is to parse, I use the generic parser, which has the complete map of all the parsers and automatically manages the checks to choose the right parser. Instead, when I know what to

parse, I directly call the specific parser to compose the object I need.

Each parser implements the interface

```
interface Parser<T> { parse(data: T): ParseResult<GenericElement>; }
```

and it must be created using the `createParser` factory:

```
function createParser<U, T extends Parser<U>>(c: new (raw: ParseFn) => T, da
ta: ParseFn): T { return new c(data); }
```

In order to set up the possibility of automatically parsing nodes (i.e. using the generic map indicated above), each parser must extend the `EmptyParser` and implement the `parse` function according to its needs:

```
class EmptyParser {
    genericParse: ParseFn;
    constructor(parseFn: ParseFn) { this.genericParse = parseFn; }
}

class ParagraphParser extends EmptyParser implements Parser<XMLElement> {
    attributeParser = createParser(AttributeParser, this.genericParse);
    parse(xml: XMLElement): Paragraph {
        const attributes = this.attributeParser.parse(xml);
        const paragraphComponent: Paragraph = {
            type: Paragraph,
            content: parseChildren(xml, this.genericParse),
            attributes,
            n: getDefaultN(attributes.n),
        };

        return paragraphComponent;
    }
}
```

Add a new parser

In order to add a new parser you need to follow some important steps:

- Analyze the element you want to parse and define a data model that will represent it.
  - This data model should be defined as an interface that extends the `GenericElement`.
  - Try to more as more specific as possible and define elements with their own type and interface, but also stay focused on the ultimate goal of your contribution: if an element it is not strictly within the scope of your goal, you can define them as

`Array<ParseResult<GenericElement>>` or `ParseResult<GenericElement>` and add a `TODO` comment that will remind future contributors to add specific type when that particular element will be handled.

- Once you have the interface, you can implement the parser for that specific element.
  - As indicated above, it should be defined as an extension of the `EmptyParser` and an implementation of `Parser<XMLElement>` and should return an element of the type you defined before.
  - Since the interface you defined before is an extension of the `GenericElement` the element returned by the parsed should also have
    - a property `type` with the interface itself as value,
    - a property `attributes` containing a map of all the attributes of the node (check existing functions to see how to parse them without rewriting everything),
    - a property `content` containing all children properly parsed (check existing functions to see how to retrive this list without rewriting everything),
    - possibly a property `class` containing the `tagName` of the node (check existing functions to see how to retrieve it without rewriting everything),
    - possibly a property `path` containing the xpath of the node (check existing functions to see how to retrieve it without rewriting everything).
- Lastly you have to add your parser to the parsers map:
  - check which TEI module the element belongs to and add the tag name to the specific list and the parser in the specific map in `xml-parsers/index.ts`
  - if there is no map for the TEI module your element belongs to, please implement it both as a list for tags and a map for parsers, and add it to the main parsers map, similarly to what has been done for other modules.

ContentViewerComponent

This is a dynamic component that takes a `ParsedElement` as input and establishes which component to use for displaying this data based on the type indicated in the `type` property.

This type is used to manage the component register, to be accessed for dynamic compilation, and also the type of data that the component in question receives as input:

```ts
/* component-register.service.ts */
const COMPONENT_MAP: Map<Type<any>> = {};

export function register(dataType: Type<any>) {
  return (cls: Type<any>) => {
      COMPONENT_MAP[dataType.name] = cls;
  };
}

@Injectable({
  providedIn: 'root',
})
export class ComponentRegisterService {

  getComponent(dataType: Type<any>) {
    return COMPONENT_MAP[dataType.name];
  }
}

/* paragraph.component.ts */

@Component({
  selector: 'evt-paragraph',
  templateUrl: './paragraph.component.html',
  styleUrls: ['./paragraph.component.scss'],
})

@register(Paragraph)
export class ParagraphComponent {
  @Input() data: Paragraph;
}
```

Localization

To handle localization we use the plugin angular-l10n[https://github.com/robisim74/angular-l10n]; in this way we can offer a runtime solution for language switching without fully reload the application.

Translations are defined in a JSON file (one for each language), saved inside the folder `assets/l10n`. This JSON is organized as follows:

```
{
    "textKey": "Text in a particular language",
    "key": "Value"
}
```

The `textKey` is the unique identifier used in the angular application (usually in the HTML template) whenever we need to retrieve a translated text. It should be in camel case.

Add a text in the localization

To add a new text in the localization, you just need to add a pair `"KEY": "Text"` for every language already existing. If you don't know the translation in a particular language use the English and let us know: we will handle the missing translations.

Work with localization in angular templates

To add a text in the UI so that it will be correctly translated when needed, just follow the steps below:

- In the HTML template, use the *pipe* `translate` whenever you need to insert a localized text, referring to the `textKey` that represents the text in question. (*NB: the* `textKey` *must exist in the JSON of the translations!*):

  ```
  <span [title]="'myTitle' | translate">
      {{'myText' | translate}}
  </span>
  ```

- In every existing json file for localization, add the new KEY and its translation:

```
en.json
{
    "myTitle": "My Title",
    "myText": "This is my text"
}
it.json
{
    "myTitle": "Il mio titolo",
    "myText": "Questo è il mio testo"
}
```

Add a new language

To add a new language to the localization so that it is automatically displayed in the language selector, just follow the steps below:

- Add the JSON file of the new language in the `assets/i18n` folder.
  - This file must be named as `"LANGUAGE_CODE.json"`, where `"LANGUAGE_CODE"` is the reference code of the new language;
  - this file must present ALL the keys that exist in the other files, so it is advisable to make a copy-paste of one of the files already present before starting with the translations.
- In the `ui_config.json` file, add an element to the list of `availableLanguages`. This element must be structured as follows:

```
{
    "code": "cd",
    "dir": "ltr",
    "label": "Lang Label",
    "enabled": true
}
```

  - `code` indicates the language code (the one used to name the JSON file);
  - `label` indicates the language label, to be shown in the UI
  - `enabled` indicates if the language should be activated or deactivated
- In every other JSON translations file the key `"language_LANGUAGE_CODE"` to have a translation of the name of the new language in all the others already present and managed.
- Finally, add a `.png` file that depicts the flag identifying the new language in the

`assets/images` folder. This file must be named with the code of the new language and must preferably be a square with not too large dimensions.

- For further information please refer to official documentation of ngx-translate/core plugin

Color theme

This new version of EVT is able to handle multiple themes at runtime. A "theme" is intended as a particular palette or set of color used for the main UI components. The theme can also change dimensions or other properties.

In the file `assets/scss/_themes.scss` we defined a global variable `$theme` where we declare every single color used in the UI components. Each color must exist in every single theme.

The themed rules depend on the `data-theme` attribute added to a `div` element of the `AppComponent` that encloses the whole application. The `data-theme` attribute is constantly linked to the current theme variable. We decided to embody everything in this external `div` in order to lighten the number of bindings of elements that need a connection to the current theme.

Add a new theme

To add a new theme just follow the steps below:

- add an object at the end of the current list of themes, which has all the properties of an existing one (we recommend doing copy-paste to be sure not to lose anything);
- change the color codes as desired;
- add the new theme id to the list of available themes in the `ThemeService` (`themes.service.ts`):

```
{
    value: 'myThemeKey',
    label: 'My Theme Label'
}
```

  - the `value` is the ID of the new theme, the key of the object previously created;
  - the `label` is the label to be displayed in the theme selector in the UI.

Add new themed CSS rules

To add new CSS rules so that colors are retrieved from the current theme (and change automatically when the theme changes at runtime), just follow the steps below:

- import the file `_themes.scss` in the `*.scss` file of the component

```
@import "path/to/_theme.scss";
```

- Embody every css rule to be themed in the following instruction:

```
h1 {
    @include themify($themes){
        color: themed('baseColorDark');
    }
}
```

Within this instruction, every css rule that uses a color and need to be linked to the current theme, must be defined as

```
    themed("colorKey");
```

where `colorKey` is the key of the color within the object representing a theme defined in the file `_theme.scss`.

The `themify` mixin

The `themify` mixin will add a CSS rule for each theme for the CSS rules defined within it. The `@each $theme, $map in $themes` tell Sass to loop over the `$themes` map that was defined above. On each loop, it assigns these values to `$theme` and `$map` respectively.

- `$theme` - Theme name
- `$map` - Map of all theme variables

Then the `map-get()` function is used to get any theme variable from `$map` and output the correct property for each theme. The `&` refer to parent selectors and placing it after `[data-theme="#{$theme}"]` tells Sass to output any parent selectors after the theme name. To use this mixin, just be sure that the element for which you are defining the CSS rules is included in a `*[data-theme]="theme-name"` element and embody every CSS rule that needs to be themified within the mixin:

```
btn-primary {
    @include themify($themes) {
        color: themed('baseColorDark');
    }
}
```

Main style

EVT uses `.scss` files. During development you can use some useful global variables and mixins. They are defined respectively in the files below (to use them, please refer to official scss documentation):

- `/assets/scss/_variables.scss`
- `/assets/scss/_mixins.scss`

If you create a new mixin/function/variable that could be useful to others, please define it in the proper file.

Whenever you need to set a color that should be connected to the main theme, please follow the instructions given in the Color theme section.

Sample documents

The XML file included in this repository is just a pseudo-critical edition and won't probably contain the use cases needed to complete the implementation of a new feature. During development, it is therefore advisable to modify this file with the use case needed, remembering **not to commit such changes**.

In the EVT Sample Documents repository you can find a collection of TEI documents that can be used as edition examples in EVT.