Web and back-end utilities for Wikipedia

Work report

Ignacio Casares

May 2024

Contents

Preamble

This piece of work describes the planning, execution and development of Wikipedia web and back-end tools for the Master's thesis of the MsC in Web App and Site Development at the Open University of Catalonia.

In the pages that follow, information is given as to the processes of title elaboration, introduction, description, definition, objectives and tools to carry out the project; as well as proper justification thereof. A thorough description is also given of the different building blocks that make up both the Wikipedia website and the bot developed for this project.

The entirety of this report has been written in Markdown using Pandoc's LaTeX conversion.

This work is subject to a Creative Commons BY-SA 4.0 International license. The code described in the project is released under a GPL 3.0 license.

Introduction

Wikipedia is a free encyclopedia that anyone can edit. As of November 2023, it was placed as the sixth most visited website in the world.[1] Volunteer Wikipedia users often contribute towards the common goal of feeding the encyclopedia with valuable information that is based upon reliable, secondary sources,[1] through the construction of encyclopedic articles. However, due to its voluminous, wide-reaching and accessible nature, there are multiple roles to be carried out: some editors focus on technical roles, consisting on building templates, making UI modifications, developing user scripts, managing bot accounts or creating modules; others fulfill tasks that are related to maintenance, specialising in watching page modifications, tagging articles with maintenance templates, reporting vandalism or posting protection requests to administrator boards, among others. The Wikipedia ecosystem demands the contribution of these specialised users that ensure content quality and functionality. Building web and back-end utilities to aid in both maintenance and article-building tasks constitutes another way of contributing to the final goal of developing a worldwide open-knowledge corpus. In this work, two different utilities are proposed as a way to make a contribution to the Spanish-language Wikipedia: one of them will focus on the construction and deployment of a community-backed bot whose aim is to automatically apply Manual of Style (*MoS* hereafter) rules through interactions with the MediaWiki API; the second consists of a website that summarises, displays and provides visibility to the work Spanish Wikipedia LGBT+ WikiProject users carry out, through the construction of an online platform they can make use of.

[1]See the English Wikipedia guidelines on reliable sources.

Context and justification of the work

This work will be developed in two branches, both will have as main goal to tackle the Wikipedia-related tasks as described in the introductory section:

MoS bot

On the one hand, a community-backed bot that applies MoS rules will be developed. As os 2024, a number of Wikipedia maintainers struggle with the application of some of the MoS rules.[1] This is, in part, due to the fact that Wikipedia newcomers often ignore the existence of these guidelines when writing their first articles. The MoS, however, holds extensive information that provides users with the necessary tools to approach aspects such as article punctuation, formatting and spelling, among many others.

Correctly applying MoS rules is often done manually by dedicated users, and articles that have serious formatting issues are tagged with maintenance template so that other users can contribute to their improvement too. However, there is precedent of development of Web tools that assist in this area. In 2019, user Benjavalero developed a web tool for the Spanish Wikipedia that allows for semi-automatic correction of spelling errors.[2] Replacer, as the tool is called, finds common spelling errors using the search function of the Wikimedia API[3] and asks the user for confirmation through a UI before semi-automatically fixing the error through an API call. However, the MoS covers many other aspects beside word spelling. Many of its rules could potentially be enforced through the construction of rigurously-tested regular expressions and interaction with the API, reducing workload on Wikipedia editors, who could then focus on other less-automatable tasks such as article-building.

Website for the LGBT+ WikiProject

Volunteers working on the different Wikimedia projects often find ways to collaborate, share ideas and plan common goals to enhance their collective productivity. These collaborations usually take the form of two types of entities: user groups or WikiProjects.

[1]The Manual of Style of the Spanish Wikipedia can be consulted through this link.
[2]https://replacer.toolforge.org/dashboard
[3]https://www.mediawiki.org/wiki/API:Search

User groups are typically larger entities often sponsored by the Wikimedia Foundation, while WikiProjects are built around local communities in specific Wikipedias that share common interests.

The LGBT+ WikiProject (in Spanish: **Wikiproyecto LGBT+**) on the Spanish Wikipedia was founded in February 2007 with the aim of "increasing visibility and improving queer and LGBT-related content in the enciclopedia".[2] It is made up by a community of users interested in creating, improving or fixing LGBT-related content on the encyclopedia. As for March 2024, it has 173 members.[4] The LGBT+ WikiProject on the Spanish wiki is considered part of the Wikimedia LGBT+ (WMLGBT+) user group,[5] and it shares many characteristics with larger user groups, such as holding monthly meetings where participants make decisions on editing processes, events and other editing matters.

Many user groups host and maintain their own website. For example, the Wikiesfera Grupo de Usuarixs, founded in 2018, has a website where members post relevant information, present themselves as a group, and provide a contact form for potentially-interested visitors.

A website for the LGBT+ WikiProject can help enhace their visibility and expand their mission and attract potential contributors who might be interested in their goals.

[4]https://es.wikipedia.org/wiki/Wikiproyecto:LGBT/participantes
[5]https://meta.wikimedia.org/wiki/Wikimedia_LGBT%2B/Portal

Objectives

Based on the context provided in the previous section, the following objectives for this Master's thesis were defined:

- To provide the Spanish Wikipedia with a community-supported bot that performs MoS corrections automatically in a manner that is considered safe, consistent and sustainable in the long term.
- To build a website that the LGBT+ WikiProject and/or the Spanish community within the LGBT+ Wikimedia User Group, enhancing their visibility and serving as a platform to display common goals, statistics and achievements.

Impact on sustainability, diversity and socio-ethics

According to the Open Knowledge Foundation, open knowledge is defined as "any content, information or data that people are free to use, re-use and redistribute — without any legal, technological or social restriction".[3] In a report released in 2011, the foundation added that for a work to be considered "open", it cannot be released under restrictions "beyond a requirement for attribution and share-alike".[4]

The Wikimedia Foundation has embraced this philosophy from its inception, and thus all content on Wikipedia, Wikimedia Commons, and the rest of the foundation's projects is subject to an open license. For exmple, a CC BY-SA 4.0 International (formerly CC BY-SA 3.0) is applied on Wikipedia,[5] and any license as resctrictive or less so than CC BY-SA is applied on Wikimedia Commons.[6] The Wikimedia Movement's core code and API, represented by the MediaWiki engine is GPL-licensed.[7] In the essay *Can History Be Open Source? Wikipedia and the Future of the Past* Roy Rosenzweig argues that the power behind Wikipedia's reach can be attributed to its free or open nature, which provides "intriguing opportunities for mobilizing volunteer historical enthusiasm to produce a massive digital archive".[8] A 2020 publication concluded that the use of Wikipedia in combination with conventional learning resources positively impacts student's academic performance, highlighting its free availability to both educators and students as one of the contributing factors to this effect.[9]

Based on the principles established above, it could be hypothesised that developing a bot that enhaces the quality of the content provided on Wikipedia through the application of MoS rules could relieve volunteering editors from minor tasks that can be time-consuming, thus allowing them to invest a greater amount of time in gathering reliable sources and conveying their information into articles and dedicating more time to providing free-knowledge content.

The second subproject within this thesis involves providing a web platform for the LGBT+ WikiProject on the Spanish Wikipedia. According to relevant literature, LGBT+ minority groups are often underrepresented or misrepressented in politics,[10] the media,[11] and literature.[12] A report published in 2021 analysed the actions of the Wikimedia LGBT+ community and its ability to create LGBT+ content through collaborational efforts, for example through WikiProjects,[13] identifying LGBT+ working groups and establishing a positive correlation between the existence of these groups and an increase of the share of LGBT+ content on the encyclopedia.

It could therefore be argued, subject to further analysis in scientific literature, that the work of these content-providing focus groups can positively contribute to narrowing the content gap seen in these areas. Providing for an online space that collects performance statistics and serves as a platform for group publications, materialised as a website, might help enhance cohesiveness and allow them to openly display the results of their work. The idea of minority-group visibilisation through online presence is corroborated in relevant scientific literature. Scholars such as Kollock et al. (2002) hypothesize that "computer-mediated communication and networking is a useful mechanism for disadvantaged groups in their efforts at collective action and empowerment";[14] and Pullen and Cooper (2010) argue that "LGBT identity within online new media offers new scope, particularly when it is reflective, contextual and continuously self aware".[15]

Approach and methods

In order to tackle the challenge described in the previous sections, the following axioms as to the programatic approach and methods were established:

- This work focuses on building two utilities.
- Both utilities are approached from scratch. No existing projects will be adapted.
  - For the MoS Wikipedia bot, there are instances of other bots executing similar correction tasks on Wikipedia. However, there are no bots that focus exclusively in carrying out MoS corrections.
  - For the Spanish LGBT+ WikiProject Website, there are currently no instances of websites for WikiProjects, although there are for specific user groups.
- The work is designed to be part of a broader, free knowledge/software endeavour. Therefore, the result work should too follow the principles of openness that characterise Wikimedia projects and that have been endorsed by the Wikimedia Movement.
- The result of this work should be open to future contributions and development from other community members:
  - The MoS Wikipedia bot should consider input and comments from other users, and its code should be available for interested users to fork and/or continue the work blueprinted by this endeavour.
  - The website should be open to input and comments, and available for use by interested contributors of the LGBT+ WikiProject and/or the Wikimedia LGBT+ user group.
- The result of this work should be viable long-term, and efforts should be made to ensure its feasibility in the long run. This includes making future updates and continuously seeking input from relevant contributors.

Planning

For the drafting and formatting of references, the bibliography manager Zotero was used, along with the Zotero Connector browser extension for Firefox.[1] A new, specific collection was created for the project. The citation format used for the report is numeric APA with notes, utilising a template provided by the Citation Style Language project.[2]

For task management and distribution, the web-based kanban-style application Trello was used.[3]

Regular conversations with the corresponding tutor were arranged through Google Meet, while e-mail exchanges were frequent and carried out through institutional e-mail. Feedback from the LGBT+ WikiProject was gathered through the monthly meetings (calendar) and through their official Telegram group chat. Feedback for the bot was gathered through Wikipedia's discussion pages.

In order to ensure the existence of back-up and a version control system, a repository was set up on GitHub.[4]

A Gantt diagram portraying the envisaged work scheduled can be seen in the corresponding figure.

- For the bot project, it was decided it would be written using TypeScript and run though Node.js; making use of the MediaWiki API and the mwn framework. Testing would be made using Jest. Refer to the technologies and dependencies section for more information.
- For the website project, it was decided that the Angular framework would be used (TypeScript). The VScode code editor was used for programming tasks. The Angular Language Service, Angular Snippets were installed to aid with.

[1] https://www.zotero.org/
[2] https://citationstyles.org/
[3] https://trello.com
[4] https://github.com/nacaru-w/TFM

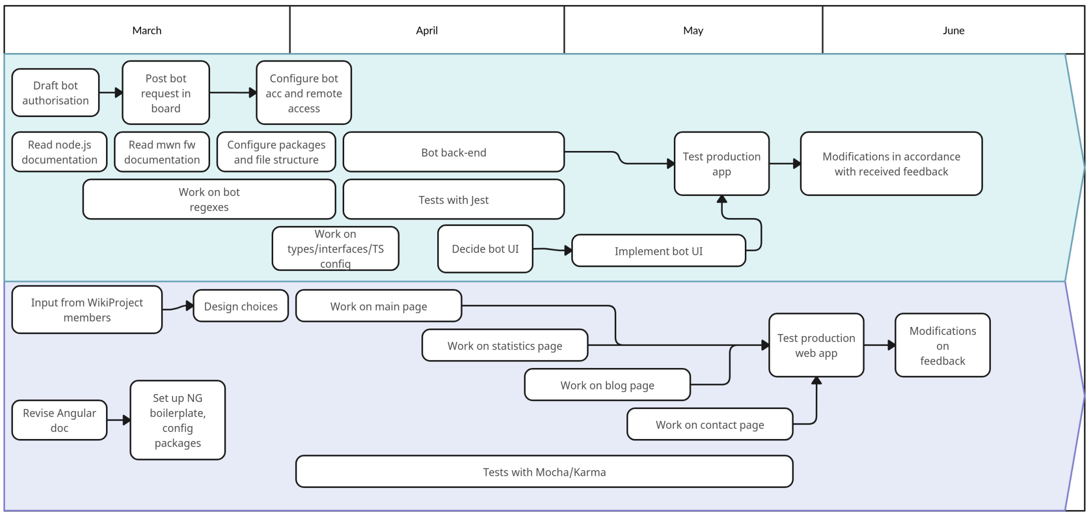

March
April
May
June
Draft bot authorisation
Post bot request in board
Configure bot acc and remote access
Read node.js documentation
Read mwn fw documentation
Configure packages and file structure
Bot back-end
Test production app
Modifications in accordance with received feedback
Work on bot regexes
Tests with Jest
Work on types/interfaces/TS config
Decide bot UI
Implement bot UI
Input from WikiProject members
Design choices
Work on main page
Work on statistics page
Test production web app
Modifications on feedback
Work on blog page
Work on contact page
Revise Angular doc
Set up NG boilerplate, config packages
Tests with Mocha/Karma

Figure 1: Gantt diagram showing project timeline

Materials and methods

Market study

In both cases, this project deals with products whose objective is to better the experience of users that contribute to or make use of Wikimedia projects, especially those who interact with the Spanish Wikipedia.

- LGBT+ WikiProject website:
  - The website will serve as a visibility platform for the work done by Wikipedia users in this community:
    - It will allow users within the group to see the results of their contributions displayed through web visualization.
    - This might also constituye an introductory point for potential newcomers who might be interested in the labour carried out by members of the LGBT+ WikiProject on the encyclopedia.
- MoS Bot:
  - The *bot* carries out maintenance tasks, implementing community consensus through modifying articles to reflect the directives described in the MoS, which is a document summarising the result of years of community decisions based on consensus. This addresses the needs of both Spanish Wikipedia contributors and readers:
    - Contributors will benefit from having a significant portion of their MoS-related maintenance tasks automated by the bot.
    - Readers will experience improved consistency in punctuation, syntax, format, style and structure across the encyclopedia, as most MoS guidelines align with the recommendations of the Real Academia Española (the official institution for Spanish language guidelines).

Architecture

The architecture of the project can, as well, be divided in two: one for the LGBT+ WikiProject website, and another one for the bot:

- On the LGBT+ WikiProject website, the user will interact with a front-end layer that is constructed under an Angular-based web application, which follows an MVC arquitecture: separating model, view and controllers. The application interacts with APIs through services. API interactions include: interaction with a local database (MariaDB) that stores blog post information, interaction with a mail server that will send the contact form information, as well as interaction with the MediaWiki API to obtain the data to construct the statistics webpage.

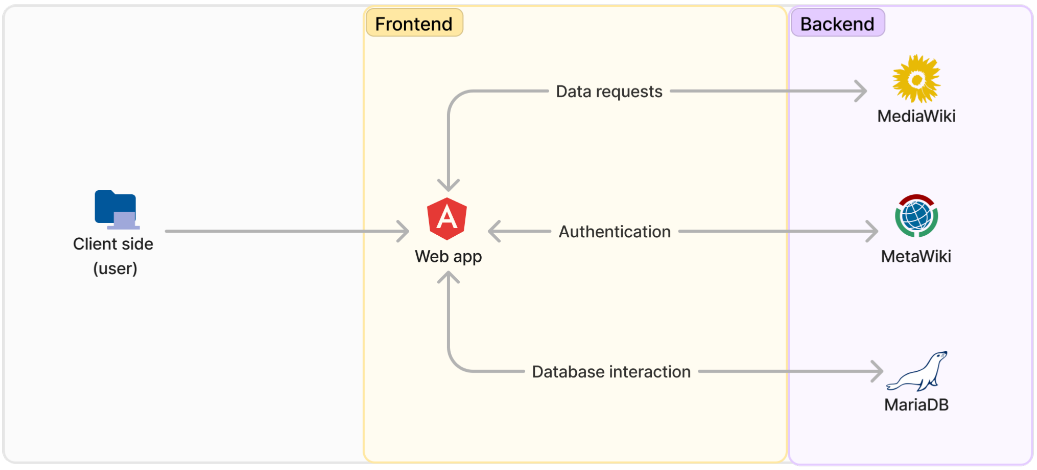


Figure 1: Diagram showing desired web architecture

- For the bot, the user (manager) will, through an interface, interact with the bot options. The application will, in turn, interact with the MediaWiki API.

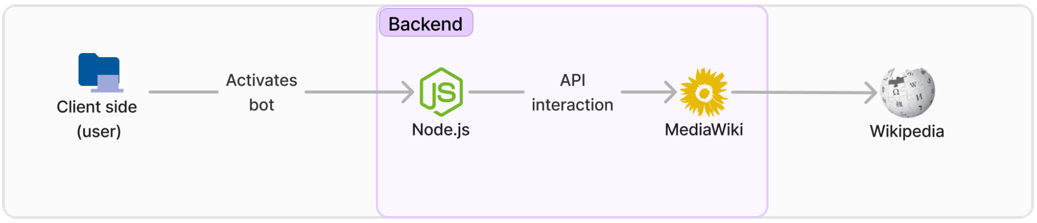


Figure 2: Diagram showing desired bot architecture

Viability

This section analyzes the viability of the project according to three standards: technical viability, resource viability, and legal/ethical viability.

Technical viability

Both projects are open-source based.

- The web project uses Angular 17.3, an open-source web application framework with extensive use.[1] It is well-documented and supported by a robust contributor community. For the back-end of this part of the thesis, a managed MariaDB database was set up in the Toolforge environment, which uses a Node.js server. The MediaWiki API, supported by the Wikimedia Foundation, with extensive documentation and a significant contributor base, will also be employed.
- The bot is Node.js-based, an open-source JavaScript runtime environment with a significant contributor community and extensive documentation. The bot uses the MediaWiki API to interact with Wikipedia pages as a bot user.

Resource viability

Regarding the viability of resources, for the website, the Toolforge hosting environment by the Wikimedia Foundation will be used. Toolforge belongs to the Wikimeda Foundation, which grants users that build services for the Wikimedia Movement the possibilty to make use of a cloud-based architecture for web and bot hosting. It also includes managed MariaDB database services.

The bot will use local hosting as of now as it will not need to be run continuously, but it could be modified to work from Toolforge as well in the future. The website will be maintained by members of the LGBT+ WikiProject.

As to the financial feasibility of the projects, services provided by the WMF, such as hosting through Toolforge, will not have any financial impact. Neither project has a for-profit objective (the website will not include advertising), thus no analysis of potential revenue streams was performed.

[1] According to data from Stack Overflow, it constituted the second most-used JavaScript framework as of 2023.

Legal and ethical viability

The entire project is released under a free license (CC BY-SA 4.0 for text, images, and other media, and GPL-3.0 license for its code). Toolforge requires users to release their content under a free license. The content, approach and scope of the webpage follow the guidelines stablished in the Wikimedia Foundation's Universal Code of Conduct, and align with the Movement Strategy direction within Wikimedia 2030.

Usability

Usability is of paramount importance in the context of web development. Usability principles such as the ISO 9241-11 standard describe how to apply usability in interactive systems and services.

Usability of the LGBT+ WikiProject website

Usability was prioritized during the development of the LGBT+ WikiProject website. Consequently, the website was designed as a single-page application (SPA). SPAs offer faster load times, as the entire web application is initially loaded and content is subsequently updated dynamically as needed. Since resources are cached after the initial load, subsequent interactions are quicker as the application fetches only new or updated data.

Transitions can also consistute an effective way of enhancing usability. SPAs also facilitate the implementation of smooth transitions between different views and sections without the delay caused by full-page reloads, enhancing fluidity.

Consistency in the layout was another priority. The website maintains certain static elements and only modifies specific components to ensure visual homogeneity, which helps reduce cognitive load. Coupled with quicker resource load times, this allows for easier user feedback.

Mobile optimization is also simpler to implement in SPAs, as the consistent UI allows for higher simplicity in designing the elements for display on smaller screens. This is critical in an industry where a mobile-first approach has become essential.

For Wikipedia users, ensuring quick navigation through different sections such as the blog, forms, and statistics pages was a top priority. A significant number of Wikipedia users access the website from mobile devices, and the Wikimedia Foundation has been working to ensure mobile compatibility for their projects.[1] To align with this approach, the website's responsiveness was thoroughly tested throughout its development.

Usability of the MoS bot

In this case, as user interaction does not happen primarily through visual means, the MoS bot's usability was focused around the idea of providing comprehensible logs and an edit history that handlers can use to evaluate the bot's actions.

The bot provides a real-time logging system that allows the controller to see every article it is analyzing and performing actions upon. When the latter fails, the bot also logs the reason why the edit could not be carried out.

Accuracy was also an utmost priority when constructing the bot. Due to the high level or responsibility derived from managing a tool that makes massive contributions to Wikipedia pages, extra efforts were dedicated to ensure that these actions conform to Wikipedia standards. Because of this, the regular expressions that the bots uses are thoroughly tested (refer to the regular expressions and testing sections for more information) through the usage of the Jest framework.

Efficiency is also a component that was taken into account when designing the bot. This is visible through the way the bot interacts with the MediaWiki API. To minimize the load on MediaWiki servers, articles are obtained by sets of 50 (the maximum number of articles allowed by the API) and bot selection of articles is not performed randomly but pseudorandomly (refer to the [genArticles function] section for more information).

[1]A piece of news on the topic was released on *The New York Times* in 2015. It discussed the difficulties the Wikimedia Movement might face tackling the adaptation of their services to a growing smartphone-dominant world. Other new outlets have suggested that the reason why the number of Wikipedia editors is slowly declining might be due to the increased use of these devices.

LGBT+ WikiProject website

The following sections describe the building blocks that make up the website for the Spanish Wikipedia LGBT+ WikiProject.

File structure

File structure is described in the following diagram:[1]

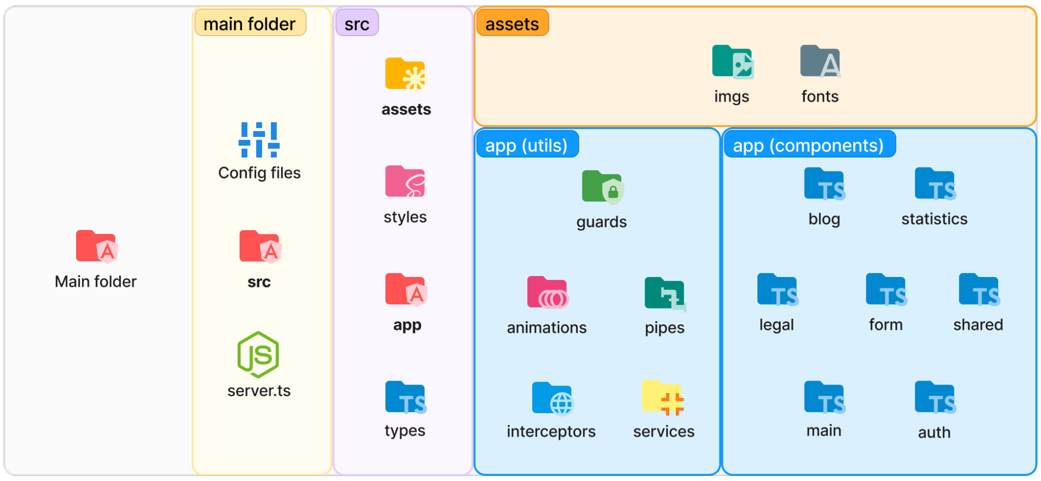


Figure 1: Website folder structure

Models and interfaces are saved as their own folder in each one of the dedicated component's folder. Certain folders that are dedicated to components have their own sub-folders that cover specific purposes. For example, chart data fules are saved inside a `chart_data` folder inside the `statistics` folder.

The schema followed for file structure is based on previous experiences working with Angular projects.

[1]For brevity purposes, certain native Angular folders, such as the `node_modules` and `dist` folders, have been ommited in the representation of the figure.

Dependencies

The following dependencies were used in the project. Dependencies included with the installation of Angular 17.3 were omitted (such as Angular Animations):

- Stylelint
- ESlint
- Passport and Passport-MediaWiki-OAuth
- Bootstrap and Ng-Bootstrap
- Charts.js
- Express and Express-session

Their use in the project is described throughout the sections that follow.

Linting and code analysis

In order to ensure quick detection of potential errors in our code, the static code analysis tool EsLint was installed in the project.

The recommended configuration was enforced.

To ensure consistency in the SCSS code as well, Stylelint was installed as part of the project.

Charting library

The main page of this website consists of a `statistics` component that uses interactions with the MediaWiki API service to obtain up-to-date information about the LGBT+ WikiProject activity, which is then mapped and parsed so as to display it in an accessible, visual manner. This was conceived through the usage of a chart-constructing library that allowed a high degree of customisation, in order to make sure that the graphs follow the the designated neo-brutalistic aesthetic approach.

To achieve this, the chart.js library was chosen. Across all considered options, this library offered the highest degree of customisation, it includes animations (which can also be customised) and is released under a license (MIT) that allows free use.

Animations

The inclusion of animations was conceived as part of the aspects that would enhance user experience. The angular/animations library was primarily used for this purpose. The chart.js library also allows for the configuration of animations in its components.

All the animations that were built for the project are stored in an `animations.ts` file found in the `app` folder. They are then imported into the different components. The importing system allows for the reuse of the same animation in several components without the need to rewrite the code.

A description of the different animations and how they are applied in the components is provided in the following sections.

Router animation

A router animation for every page route was introduced. This allows for the entrace of each main component through a slide-in animation. This was achieved through the modification of the `routes.ts` file. A specific `animations.ts` file was created for this, which handles the animations related to the main `app.components.ts` file.

```
export const routes: Routes = [
    { path: '', redirectTo: '/home', pathMatch: 'full' },
    { path: 'home', component: MainPageComponent, data: { animation:
↪   ['HomePage', 'footerAnimation'] } },
    { path: 'blog', component: BlogMainComponent, data: { animation:
↪   ['BlogPage', 'footerAnimation'] } },
    { path: 'form', component: FormMainComponent, data: { animation:
↪   ['FormPage', 'footerAnimation'] } },
    { path: 'stats', component: StatisticsMainComponent, data: {
↪   animation: ['StatisticsPage', 'footerAnimation'] } }
];
```

The animation is then applied as a directive on the `div.main` element inside `app.component.html`. This `div` element contains the `router-outlet` element in charge of handling route changes and displaying the different components. The animation trigger is provided through the `getRouteAnimationData` function:

```
  getRouteAnimationData() {
    return
     ↪  this.contexts.getContext('primary')?.route?.snapshot?.data?.['animation'];
  }
```

Footer animation

A specific animation was described for the footer. It hides it and makes it reapppear when navigation through routes is carried out. This was carried out through the `footerAnimations` variable in `animations.ts`:

```
export const footerAnimations =
    trigger('footerAnimations', [
        state('visible', style({
            transform: 'translateY(0%)',
            opacity: 1
        })),
        state('hidden', style({
            transform: 'translateY(100%)',
            opacity: 0
        })),
        transition('visible => hidden', [
            animate('0.2s ease-out')
        ]),
        transition('hidden => visible', [
            animate('0.5s ease-in')
        ])
    ]);
```

This animation is the applied in `app.component.html` through the use of a specific animation directive. The directive makes use of the `footerAnimationState` variable to decide whether to show or hide the footer. The variable, typed as a boolean, is changed whenever navigation starts and ends. This is carried out through a subscription to router events and using the `NavigationStart` and `NavigationEnd` events:

```
  ngOnInit() {
    this.router.events.subscribe(event => {
      if (event instanceof NavigationStart) {
        this.hideFooter();
      } else if (event instanceof NavigationEnd) {
        setTimeout(() => this.showFooter(), 500);
      }
    });
  }
```

Last articles animation

A specific animation for the `statistics-last-article` components was designed. This animation allows the cards containing information related to the last three articles built by members of the LGBT+ WikiProject to appear through a pop effect. To achieve this, the pop animation was built:

```
export const popAnimation =
    trigger('popAnimation', [
        transition('* => *', [
            query(':enter', [
                style({ opacity: 0, transform: 'scale(0.5)' }),
                stagger(100, [
                    animate('300ms ease-out', style({ opacity: 1,
                    ↪  transform: 'scale(1)' }))
                ])
            ], { optional: true })
        ])
    ])
```

The `statistics-last-articles.component.html` file uses an Angular `@for` block to iterate over the items and make them appear one by one, imitating a ladder animation. The component uses an `ngIf` directive that listens to the `isDictFull` function, which returns a boolean which is only `true` if the dictionary containing information on three of the last LGBT+ articles has been appropriately filled through Wikipedia and Wikidata API requests. This prevents the HTML card from having its content populated asyncronously. Instead, it displays a loading spinner that disappears only when the cards are ready to load, at which moment the cards appear through with popping effect.

Form button animation

An animation to show users visual form validation feedback was designed. It affects the submit button, which should appear in a `disabled` state when the different form input do not contain information that passes specified the Angular reactive forms validation system.

The animation changes both the opacity and and background colour when the specified element goes from `disabled` to `enabled` state:

```
export const buttonState =
    trigger('buttonState', [
        state('disabled', style({
            backgroundColor: '#cccccc',
            opacity: 0.5
        })),
        state('enabled', style({
            backgroundColor: '#b3efff',
            opacity: 1
        })),
        transition('disabled => enabled', [
            animate('0.5s ease-in')
        ]),
        transition('enabled => disabled', [
            animate('0.5s ease-in')
        ])
    ])
```

In the `form-main.component.html` file, the animation is triggered through a check of the `isFormValid` function, which analyses the form group `webForm` object, specifically by checking the `valid` key; it also checks whether the form controls `readPrivacy` and `readPolicy` are true. This is carried out by making use of a ternary operator in the animation directive as follows:

```
[@buttonState]="isFormValid() ? 'enabled' : 'disabled'"
```

This allows the view to provide the desired style real-time according to the validation status of the form.

Chart animations

An animation was designed to trigger when displaying each animation inside the statistics component. For this purpose, the `chartsSlideInOutAnimation` animation was described:

```
export const chartsSlideInOutAnimation =
    trigger('slideInOut', [
        transition(':enter', [
            style({ transform: 'translateX(-100%)', opacity: 0 }),
            animate('300ms ease-in', style({ transform: 'translateX(0)',
             ↪  opacity: 1 }))
        ]),
        transition(':leave', [
            animate('300ms ease-in', style({ transform:
             ↪  'translateX(100%)', opacity: 0 }))
        ])
    ])
```

It was implemented through the `statistics-shared` component, where a function `showChart()` containing a `switch` statement manages the different chart routes handling the display of the different statistics components.

Additionally, chart.js includes specific animations for each type of chart that was built with the library. These are configured through the `options` object that is passed to each chart and not through the Angular Animations library.

Design

For the design of the website, a colour palette was established in consensus with the LGBT+ WikiProject. For the general design approach, a neo-brutalist focus was decided.

Neo-brutalism is a web design trend that draws inspiration from the brutalist architectural movement. It emphasizes raw, straightforward design elements characterized by bold colours, strong contrasts, and a focus on functionality over aesthetics. This trend is a reaction against the polished minimalism that has dominated web design in recent years.

Colour palette

Uninamous consensus was reached via direct feedback with WikiProject members that the website should bear the same colour palette as the project's Wikipedia page (which is, in turn, based on the colours of the Rainbow Flag and the transgender right flag).

A theme was created through Bootstrap's native theme creation system, with the following colours used as palette:

```
$theme-colors: (
    "light": #ffe3ea,
    "dark": #A8284A,
    "primary": #b3efff,
    "secondary": #ffc0cf,
    "info": #ffb65c,
    "success": #54de7d,
    "warning": #fff574,
    "danger": #fa7c7c,
);
```

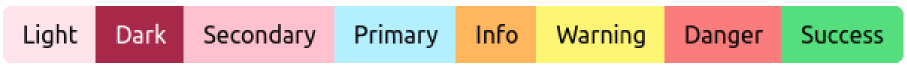


Figure 2: Theme colours of the Website's palette

Font

For the font, several options were proposed to the LGBT+ WikiProject, all following the neo-brutalist approach, following the reccommendations of the article in Medium published by web designer Sepideh Yadzi:

- Lexend
- Lexend Mega
- Lexend Deca
- Public Sans
- Mabry Pro
- Archivo

It was decided that the main font would be Lexend Deca. While the font for bodies of text would be Lexend.

Buttons

In order to make buttons approach the desired neo-brutalist web style, a pitch-black (#00000) shadow was added to the buttons:

```
button.btn {
box-shadow: black 5px 5px 0px
}
```

Neo-brutalist shadow is characterized by having thick `x` and `y` offsets (first and second values), and no `blur radius` (third value). In buttons, as in the rest of the components, the borders are rounded. Bootstrap's button component default border radius is stablished at `0.375rem`. This value was kept as it was considered round enough for the desired design.

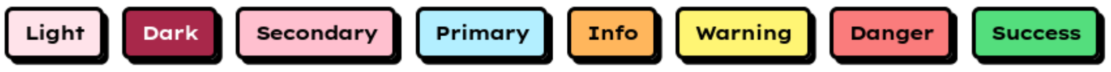


Figure 3: button design

The `letter-spacing` and `font-weight` properties were also changed for this component. The former being set at `-0.1rem` and the latter at `bold`, following neo-brutalist recommendations. A `font-size` reduction to `90%` was also set. The `border` property was added, with a value of `solid black 3px`.

Accordions

As with buttons, accordions were adapted to produce a neo-brutalist style. This implied certain modifications to the `.accordion` class:

```
.accordion {
    border: solid black 3px;
    border-radius: 10px;
    box-shadow: black 4px 4px 0px;
}
```

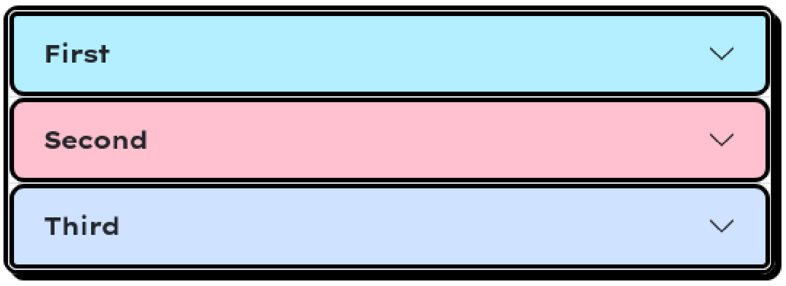


Figure 4: accordion design with primary, secondary and primary-subtilised colours

The borders were established similarly to the button's borders, with thick outline and shadow and generous radius. The individual buttons inside each section of the accordion were also given thick and rounded borders through the modification of the same properties, the spacing was reduced and the font was made bold to make it more visible:

```
.accordion-button {
    font-weight: bold;
    letter-spacing: -0.1rem;
    border: solid black 3px;
    border-radius: 10px;
}
```

Furthermore, the description text inside each accordion-body had its font modified in order to enhance readability:

```
.accordion-body {
    font-family: "Lexend", sans-serif;
}
```

This would adapt the design of this component to the desired aesthetic, an early concept is provided in the relevant figure.

Alerts

Alert messages were also conceived as part of the design process of the website. The messages follow the theme colour palette and use the Lexend font family, but applied pastelized versions of them.

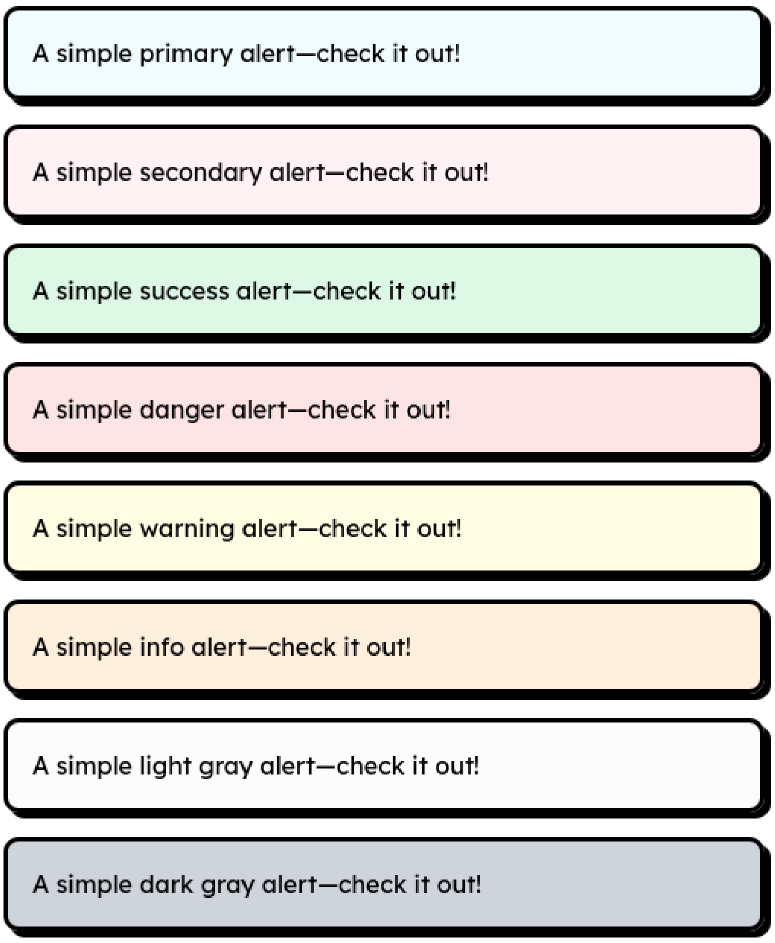

A simple primary alert—check it out!
A simple secondary alert—check it out!
A simple success alert—check it out!
A simple danger alert—check it out!
A simple warning alert—check it out!
A simple info alert—check it out!
A simple light gray alert—check it out!
A simple dark gray alert—check it out!

Figure 5: alert design with palette and monochromatic colours

As per neo-brutalist design guidelines, they were given a black font colour, emphatic `border radius` value at `10px` and stark `box shadow`.

Dropdowns

Dropdown menus were also customized. In this case, two versions were made: one in dark that stays within the primary colour range; and a light one, that bears the secondary colour palette (which only shows when hovering over the component).

The font established was the main website title font, Lexend Deca. The main dropdown button would have its `font-weight` modified to bold, while the options inside the menu would keep a normal appearance.

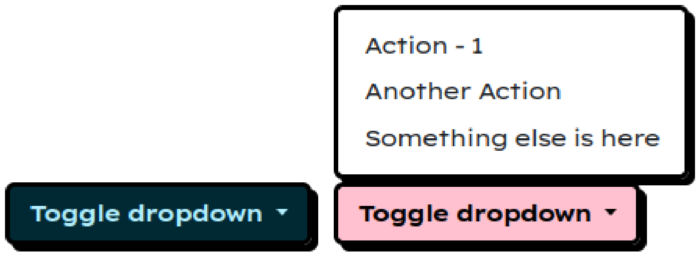


Figure 6: dropdown designs

In accordance with the rest of the design decisions, the component was given a thick black border and thick bow shadow:

```
    .dropdown-menu {
        border: solid black 3px;
        box-shadow: black 4px 4px 0px;

        .dropdown-item {
            letter-spacing: -0.1rem;
            font-size: 0.9em;
        }

    }
```

Carousel

In order to introduce the frontpage of the website, a carousel component was designed. In order to follow the established design conventions and give it a neo-brutalist appearance, its characteristics were adapted through the following SCSS code:

```
.carousel-inner {
    width: min(95%, 900px);
    height: min(95%, 400px);
    margin: 0 auto;
    border-radius: 40px;
    border: solid black 4px;
    box-shadow: black 6px 6px 0px;

    .img-wrapper {
        overflow: hidden;

        img {
            width: 100%;
            height: 100%;
            object-fit: cover;
        }
    }

    .carousel-caption {
        color: black;
        font-weight: bold;
    }
}
```

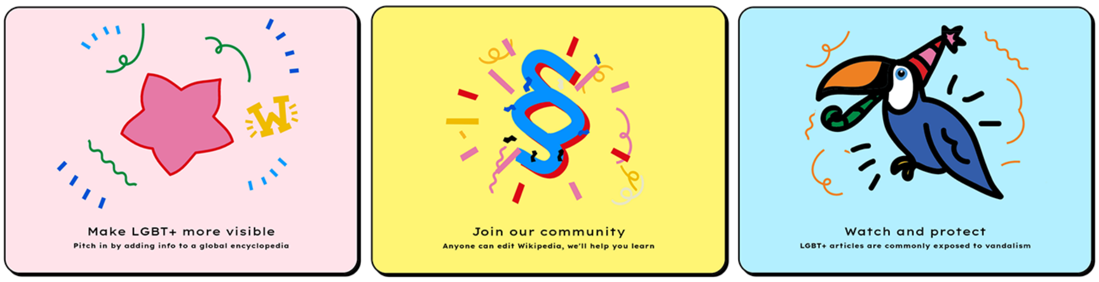

Make LGBT+ more visible
Pitch in by adding info to a global encyclopedia
Join our community
Anyone can edit Wikipedia, we'll help you learn
Watch and protect
LGBT+ articles are commonly exposed to vandalism

Figure 7: Example of the images in the carousel. On the website, these slide to the side except when they are under focus or being hovered on. The text shown on the image above is a placeholder.

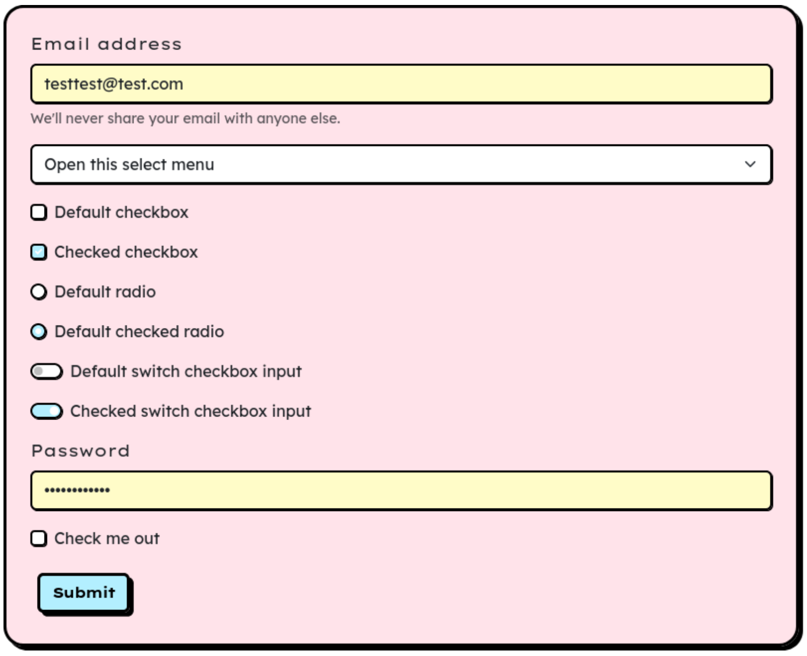
As it can be seen in the code described above, a greater `border-radius` property was given to this element, due to the increased size of the component compared to the others. The image inside the carousel was adapted so that it would occupy the totality of the container. The size of the carousel was given responsive size adaptability through the use of the `min` function, which was given the arguments `95%` and `900x`, allowing a greater margin in bigger screens and reducing its size in smaller screens. The border shadow property was given greater values compared to the rest of the components as well.

Forms

A form component, consisting of an input, checkbox, switch, radio, select and button elements was introduced. As per the rest of the components described in this section, the form follows the design directives of neo-brutalism. The following list describes various features within this component:

- Labels were given the Lexend Mega font with slightly reduced `letter-spacing`.

- The rest of the text within the form was provided with the secondary font, Lexend.
- Borders were accentuated, their shadows were given stark colours.
- The primary colour was employed when marking/checking one of the elements within the component. It was also employed when on focus state.
- Select dropdown follows the same characteristics as described in the drop-down section.



Email address

testtest@test.com

We'll never share your email with anyone else.

Open this select menu

☐ Default checkbox

☑ Checked checkbox

○ Default radio

◉ Default checked radio

Default switch checkbox input

Checked switch checkbox input

Password

••••••••••••

☐ Check me out

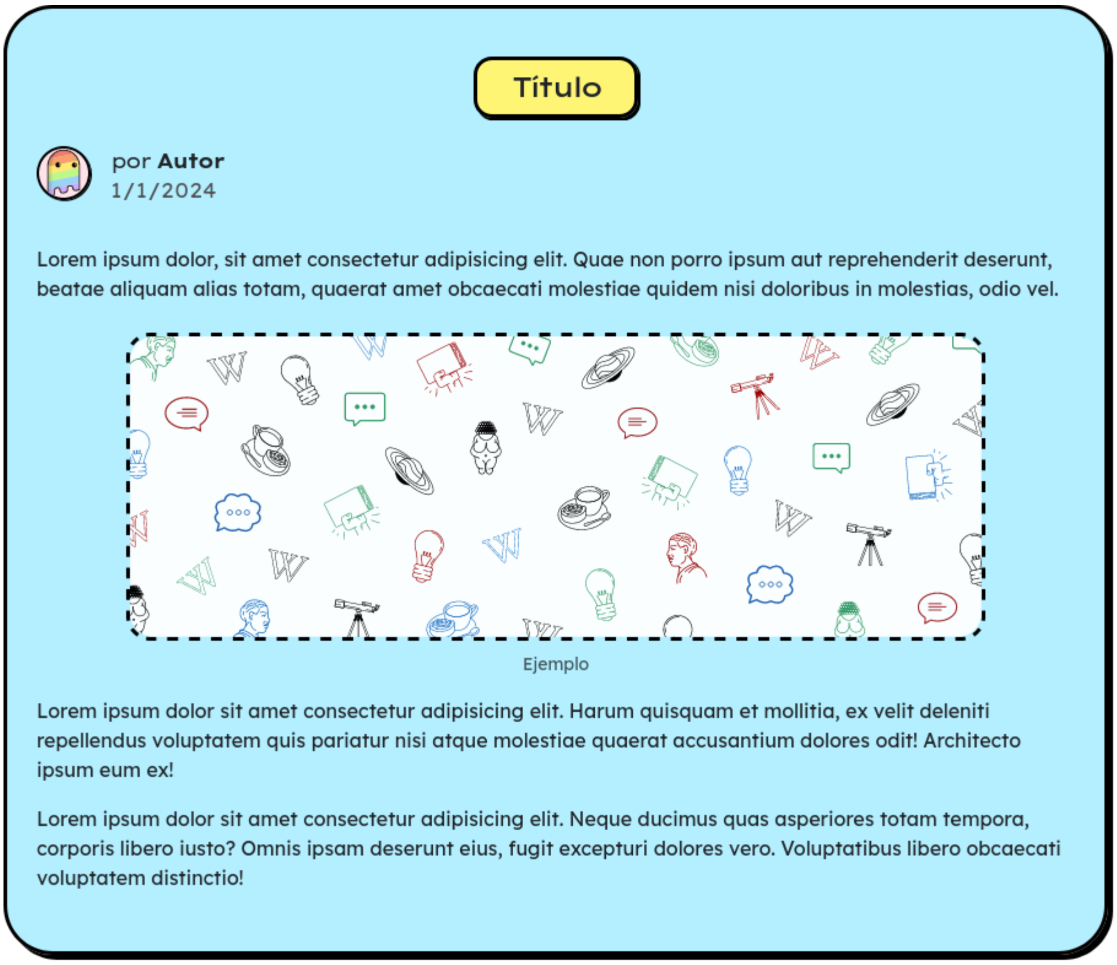
Figure 8: Example of the form elements

The form adapts its `width` property automatically so that it takes a narrower space when viewed from smaller screens.

Blog post model

A specific blog post model was designed to hold information that members of the group would like to publicly share through this means. The following elements were designed as part of it:

- A title element, encompassed within an `h2` element. Its width was made adaptable through the application of the `width: fit-component` declaration. Letter spacing was reduced to `0.125rem` in accordance with the usage of the main title font of the website. Borders were made dense and black, and were given a substantial `box-shadow`, to adapt the style to the rest of the components.



Título

por **Autor**
1/1/2024

Lorem ipsum dolor, sit amet consectetur adipisicing elit. Quae non porro ipsum aut reprehenderit deserunt, beatae aliquam alias totam, quaerat amet obcaecati molestiae quidem nisi doloribus in molestias, odio vel.

Ejemplo

Lorem ipsum dolor sit amet consectetur adipisicing elit. Harum quisquam et mollitia, ex velit deleniti repellendus voluptatem quis pariatur nisi atque molestiae quaerat accusantium dolores odit! Architecto ipsum eum ex!

Lorem ipsum dolor sit amet consectetur adipisicing elit. Neque ducimus quas asperiores totam tempora, corporis libero iusto? Omnis ipsam deserunt eius, fugit excepturi dolores vero. Voluptatibus libero obcaecati voluptatem distinctio!

Figure 9: Blog post model design

- The element, attributed the `post-info` class, comprising author information, a small profile picture and post date. This includes two elements inside, arranged through the application of the `display: flex` declaration:
  - A `div.thumb-img-container` element, which contains an image of the post author, which was given a round border through the usage of the `clip-path` property. Its `border` and `box-shadow` were also adapted to neo-brutalist conventions.
  - A `div.author-and-date` element, which contains two `span` elements: containing author and date information, respectively. This were arreanged as a column through the use of the `display: flex;` and `flex-direction: column` declarations. Date `font-color` property was given a value that made it slightly lighter compared to the usual black font.
- Paragraphs were organised in `p` elements. The `font-family` property inside of the body was changed in accordance with the rest of the text bodies of the website, which use a Lexend font.
- Images inside the blog post body were included as part of a `figure` element, which contains the image (inside a `div` container) and a `figcaption` element. The image was given a responsive `width` through the use of the `min()` function. The borders of these images are characterised by having a `dashed` value. `figcaption` text was also made slightly lighter.

The cards used to represent blog snippets in the `blog-main` component make use of a similar design approach and thus utilise the same styles for their construction, with minimal modifications.

Styles

The project uses Sass as preprocessor scripting language for the style sheets. It uses the `SCSS` syntax.

Variable management

To ensure managing consistency and maintanability across the application's styling, a dedicated file `_variables.scss` was created to store `scss` code as sass variables. This file centralises the definition of colour, font variables and other design tokens which are then improted into other SCSS files as needed. It acts as a single source for design-related values, making it easy to update and manage the project's styling. The following code exemplifies how variables are declared in the file:

```scss
$white: #ffffff;
$light: #ffe3ea;
```

```
$dark: #e0e0e0;
$primary: #b3efff;
$secondary: #ffc0cf;
$info: #ffb65c;
$success: #54de7d;
$warning: #fff574;
$danger: #fa7c7c;
```

The `@use` Sass rule is employed to make use of the variables defined in `_variables.scss`. It loads the Sass file and namespaces its contents, preventing global scope pollution and making it clear where variables are coming from:

```
@use 'variables';

.blog-post {
  background-color: variables.$primary;
  /.../
}
```

This approach aligns with modern Sass methodologies and promotes a homogeneous, organized style codebase.

Bootstrap mapping

Bootstrap mapping is utilized to generate contextual modifier classes for colourising alerts. The mapping is facilitated through the `maps.scss` file, which defines specific Sass rules to generate these classes based on Bootstrap's theme colours. This ensures that alerts can be easily colourised according to different states or themes defined by Bootstrap, but also adapts them to our chosen colour palette.

```
@each $state in map-keys($theme-colors) {
    .alert-#{$state} {
        --#{$prefix}alert-color:
↪   var(--#{$prefix}#{$state}-text-emphasis);
        --#{$prefix}alert-bg: var(--#{$prefix}#{$state}-bg-subtle);
        --#{$prefix}alert-border-color:
↪   var(--#{$prefix}#{$state}-border-subtle);
        --#{$prefix}alert-link-color:
↪   var(--#{$prefix}#{$state}-text-emphasis);
    }
}
```

The mapping is applied after Sass compiles the native Bootstrap colours, ensuring these are overriden with our chosen values.

Routing

Angular routes configuration sets up the navigation paths for the application. This is configured through the `app.routes.ts` file inside the `app` folder.

The first route, the empty `''` route is configures as a redirection the the `home` component. This allows users of the website to be taken to the main page when searching the webpage through the standard url path. Components are mapped through their specific routes.

Angular also allows for dynamic routing. This is handled through parameter-dependent routes, in this case: `/blog/:id` and `/blog-edit/:id`. This is build in conjunction with the API endpoints (refer to the backend section for more information) so that certain information can be loaded dynamically depending on the parameter.

Some routes, such as `/blog-admin` and `/blog-edit`, are protected by an `authGuard` resolver, which ensures only authorised users can access them. Refer to the guards for more information on how Angular guards are configured in the application.

Services

In order to manage the API calls, a `services` folder was included in the project. The following services were implemented:

MediaWiki service

A MediaWiki service, in charge of dealing with the calls to the MediaWiki API, including calls to Wikidata, which serves as a central repository for structured data that can be used by other Wikimedia projects, like Wikipedia. It includes the following methods:

- `getPageContent(title: string)`, the method receives the title of a page as a parameter and makes a call to the MediaWiki API which returns the full content of a Wikipedia page as a string, in wikicode format.
- `getPageExtract(title: string)`, the method receives the title of a page as a parameter and makes a call to the MediaWiki API that, through the use of the TextExtracts extension, it returns the parsed, html-less text at the beginning of a Wikipedia page.
- `getLGBTArticleList()`, it makes a call to the MediaWiki API that obtains the content of a page (as a string) where every article belonging to the LGBT+ WikiProject is posted. This is not done in an standarized way, so the method uses a pipe that maps the content and parses it into an array.

- `getParticipantNumbers()`, it makes a call to the MediaWiki API that obtains the content of a page (as a string) where participants to the WikiProject can add themselves as part of a list. Again, this is not done in an standardised manner, as there are a significant number of ways in which a Wikipedia user can add their username as wikicode, so the received content is mapped to be turned into an object that includes the total count of participants in the WikiProject and the number of people that have joined in the current year.
- `getWikidataEntity(title: string)`, this method receives the name of a Wikipedia article as an argument and returns an object that, after parsing, returns the item code of the Wikidata element associated to the article. This is done through the use of the MediaWiki API parameter `wikibase_item` of the `pageprops` method.
- `getImageUrlFromWdEntity(Q: string)`, in this case, the method takes a Wikidata item as a parameter (the query is done through the Wikidata API through Wikibase, and not through Wikipedia). The `wbgetclaims` method is used in this case. It returns an object which is parsed to return the P18 property of the item, corresponding to an image associated it. If the query cannot find such property, it returns `undefined`.

API service

The API service is designed to deal with the interactions between our Angular app and our database through defined API endpoints specified in the `server.ts` file. Please refer to the backend section for more information on how the backend was configured.

The `BlogPostInfoModel` interface was created to manage the blog post model homogeneously, and is used as type enforcement in the service and relevant components functions:

```
export interface BlogPostInfoModel {
    id: number,
    date: Date,
    author: string,
    title: string,
    content: string
}
```

In order to avoid overloading the database server with requests a cache system was developed whereby the API service methods will first check whether there is a cache variable stored and then proceed to make the call to the API otherwise.

The `getPosts` method makes a GET request to the database server to obtain an array of object with the information on each row of the blog posts table.

```
getPosts(): Observable<BlogPostInfoModel[] | string> {
    if (this.postsCache) {
      console.log('using postscache', this.postsCache)
      return of(this.postsCache)
    } else {
      return this.http.get<BlogPostInfoModel[]>(this.endpoint +
      ↪  'blog_posts').pipe(
        tap((posts: BlogPostInfoModel[]) => this.postsCache = posts),
        catchError((error: HttpErrorResponse) => {
          console.error('An error ocurred: ', error.message);
          return error.message
        })
      )
    }
}
```

Then, it proceeds to send the information as an array so that the `blog-main` component can make use of an Angular native `@for` block to interate over the resulting posts and display them on the view. As it can be noted, the function makes use of the `tap` rxjs operator to perform the caching of the information.

The `getPostInfo` method is used to find a specific post based on an `id` which is passed an an argument to the function:

```
getPostInfo(id: string): Observable<BlogPostInfoModel | string> {
    if (this.postCache[id]) {
      // Return the cached post if it exists
      console.log('using postcache', this.postCache[id])
      return of(this.postCache[id]);
    } else {
      return this.http.get<BlogPostInfoModel[]>(this.endpoint + 'blog/'
      ↪  + id).pipe(
        map((res: BlogPostInfoModel[]) => {
          if (!res || res.length == 0) {
            return 'No se ha encontrado el post';
          }
          // Cache the fetched post
          this.postCache[id] = res[0];
          return res[0]
        }),
        catchError((error: HttpErrorResponse) => {
          console.error('An erorr ocurred: ', error.message);
          return error.message
        })
      )
    }
}
```

As described in the code above, the function makes use of the rxjs `map` parse the received array (containing one object) and return a sole object instead. The caching of the post is carried out by storing the post in an dictionary.

The `addPost` function manages the interaction with the API when it comes to POST requests. These are used to add new rows to our `blog_posts` table in the MariaDB database:

```
addPost(date: string, author: string, title: string, content: string):
↪  Observable<any> {
    const postData = { date, author, title, content };
    console.log(postData);
    const headers = { headers: { 'Content-Type': 'application/json' } }
    return this.http.post<any>(this.endpoint + 'blog', postData,
    ↪  headers).pipe(
      map(response => {
        // invalidate cache for all posts
        this.postsCache = null;
        console.log('Post added succesfully with title:',
        ↪  response.title);
        return response;
      }),
      catchError((error: HttpErrorResponse) => {
        console.log('An error occurred: ', error.message);
        return of(error.message)
      })
    )
}
```

The function draws the information from the `blog-edit` component and it builds a body to be sent to the corresponding API endpoint. It also adds the appropriate headers to the POST request.

The `editPost` function is in charge of dealing with PUT requests that modify post information. The function takes the post `id` as an identifying parameter, and then the `date`, `author`, `title`, and `content` values. These are all drawn from the `blog-edit` component and encapsulated into a `putData` object. It uses headers to ensure the body is parsed into `json` and is then sent to the API endpoint by appending post `id` to the base URL.

```
editPost(id: string, date: string, author: string, title: string,
↪  content: string):     Observable<any> {
  const putData = { id, date, author, title, content };
  const options = { headers: this.getHeaders() }
  return this.http.put<any>(this.endpoint + 'blog/' + id, putData,
  ↪  options).pipe(
    map(response => {
      // invalidate cache
      this.postsCache = null;
      delete this.postCache[id];
      return {
        success: true,
        message: `Post edited successfully with id: ` + response.id
      }
    }),
    catchError((error: HttpErrorResponse) => {
      console.error('An error occurred: ', error.message);
      return of(error)
    })
  )
}
```

It processes the response through the `map` operator, and deletes the information of cached posts by setting `this.postCache` to `null`, as well as the specific post cache entry in the `postCache` dictionary. The returned object sends a message of success as a boolean as well as a message containing the edited post's `id`.

The `deletePost` function carries out the deletion of a blog post row in the database by sending a DELETE request to the appropriate endpoint. Similarly to the previous case, it makes use of the `id` parameter of the post and sets up the appropriate hedaers through the `getHeaders` method.

```
deletePost(id: string): Observable<any> {
  const options = { headers: this.getHeaders() }
  return this.http.delete<any>(this.endpoint + 'blog/' + id,
  ↪  options).pipe(
    map(response => {
      this.postsCache = null;
      delete this.postCache[id]
      return {
        success: true,
        message: 'Post deleted successfully with id: ' + response.id
      }
    }),
    catchError((error: HttpErrorResponse) => {
      console.error('An error ocurred: ', error.message);
      return of(error)
    })
  )
}
```

Once the response is received, the function processes it, and, in a similar fashion, resets the posts cache and deletes the post from the post dictionary by using the `id` as an index signature. It returns the same type of object as the `editPost` function: including a sucess boolean and a message.

The last `getLoginStatus` and `isAdmin` functions are used to verify the authentication status of the user. The former makes a GET request to the `api/user/` endpoint, which obtains either an object containing a a key `reason` with the `Not logged in` text; or a different object containing the `displayName` and `isAdmin` keys, which contain a string and a boolean.

```
getLoginStatus(): Observable<any> {
  const options = { headers: this.getHeaders() }
  return this.http.get<any>(this.endpoint + 'user', options).pipe(
    map(response => response),
    catchError((error: HttpErrorResponse) => {
      console.error('An error ocurred: ', error.message);
      return of(error);
    })
  )
}
```

Refer to the backend section to read more about how the API endpoints that were designed for this service to interact with.

Components

The components are the main pillars that hold the website. They are built through using the designed elements described in the «Design» section above and integrating logic and interactions with the services. The following components make up the website:

Main page component

The component used to build the main page holds a carousel presenting Wikimedia Movement artwork, as well as two cards: one promotes the LGBT+ WikiProject monthly *País del Mes* event, and the other one holds easily-substitutable information.

The carousel is build through a `ngb carousel` component. It then uses an Angular `@for` block to loop through the images. These images are stored as an array to allow easy modification.

The card promoting the LGBT+ WikiProject monthly *País del Mes* event is dinamically built. To do this, it uses the `getPaisDelMesInfo` function to make use of the `getPageContent` service method that obtains a string from a Wikipedia page (in Wikicode format). It is then parsed through the `processPaisDelMesString` function, which uses the `indexOf` and `substring` JS native methods to locate the relevant part and then crop the string, respectively. Both the image and the country subject for the event are then used to build the corresponding card, whose text is obtained through the `findImage` and `findCountry` methods.

Components that are part of the statistics module

The components holding data on the work done by the LGBT+ WikiProject on the Spanish Wikipedia are part of the statistics module. Its components are the main interactors with the MediaWiki service. These are:

- `statistics-shared`: it is the homepage of the module. It includes navigation to each one of the charts.
- `statistics-monthly-articles`: it holds a line chart and information on monthly article creation by members of the LGBT+ WikiProject.
- `statistics-yearly-articles`: it holds a bar chart and information on article creation by members of the project, divided by year.
- `statistics-participants`: it holds a doughnut chart and information on newcomers to the LGBT+ WikiProject.
- `statistics-notable`: it holds a star count and information on articles that have a notable status.[2]
- `statistics-last-articles`: it shows the last three articles that were added by members of the LGBT+ WikiProject.

Apart from the `statistics-shared` and the `statistics-last-articles` components, each component's html code is composed of a `div.chart` element holding visual information and a `chart.text` containing a descriptive piece of text. Each components' specifics will be described throughout the following sections.

`statistics-shared` component

The component that holds each one of the chart contains the `nav` element that allows for navigation, as well as a container for the chart.

In order to show the proper component upon selection, the `showChart(chart: string)` function was developed. This function works by, first, hiding all the charts through a specific `hideAllCharts()` function. Then, it makes the selected chart visible, by

[2]As of 2024, notable status on the Spanish Wikipedia is achieved through the «Artículo Bueno» and «Artículo Destacado» certifications.

modifying a variable with the chart's name that holds a boolean. A small delay has been implemented before this change occurs to allow for smoother transitions. The selected chart then reads its corresponding variable through a `@ngIf` directive, showing the right component.

The transition after each chart is selected is animated through the `@slideInOut` transition. See the «transitions» section for more information.

`statistics-monthly-articles` component

This component holds information on how many articles are created by the LGBT+ WikiProject each month. The data is dynamically obtained through the MediaWiki API in Wikicode format and then parsed to obtain the necessary data, which is then used to build a chart.js line chart.

The `getMonthlyArticlesInfo()` method uses the `getPageContent` MediaWiki service method to obtain the full string (in WikiText) of the page that holds article information. It then uses the first article of the 2024 year to create a substring with the current year information. Then, it calls a specific method `getMonthlyArticleArray` that uses a regular expression to build an object containing key/value pairs where each key corresponds to each month of the year (in Spanish) and each value contains the number of articles that were created by WMGLBT+ members in each month. It then uses this information to animate and populate the data for the monthly count Chart.js chart.

The chart is created through the specific `createMonthlyArticlesChart` method, which imports data from dedicated files. Another specific method, `animateThisMonthArticleCount`, is reserved to animate the number that holds the count of the articles that have been created this month.

`statistics-yearly-articles` component

This component holds information on how many articles were created during the last few years and displays it through a bar chart. Similarly, the data for the last year is obtained through MediaWiki API in Wikicode format and processed to build the chart..

The method that interacts with the API is the `getAllArticles` method. After it obtains the response, it parses it through the `getThisYearARticles` method. It uses the first article of 2024 to find the index, because the `getLGBTArticleList` only returns a list of articles with no dates associated to them.

Then, it uses the obtained information to update the chart.js chart, which is built through the `createYearlyArticlesChart`. The number on the `.char-text` element (`.count-text`) is animated through the `animateTotalArticleCount` method, which does so through the `setInterval()` native JS function.

`statistics-participants` component

This component deals with the chart that holds information on participants of the LGBT+ WikiProject. Because of inconsistencies in the way dates for newcomers were stored on the WikiProject in the years going from 2007 to 2020, these have not been included separately in the chart. They have, however, been included in the total count.

In order to do this, it interacts with the MediaWiki API through the `getParticipantNumbers` method. This method calls the API, treats its response, and builds and returns an object that follows the `Participants` interface:[3]

```
export interface Participants {
    totalCount: number,
    thisYearCount: number,
}
```

The reason why not every year's new participants are given a key/value pair in the object above is, as mentioned in the introduction, due to early data not being consistent. The members of the project are aware of this and are working on a solution that can allow for easier treatment of the data. Participant numbers obtained for the 2021, 2022 and 2023 years in the chart are imported from a local file.

The received object is process through the `getParticipantInfo()` method, which then maps the response and assigns values to the chart.js doughnut chart, which is built through the `createParticipantCountChart` method. Animation of the number in the `.chart-text` element is done through the `animateTotalParticipantCount` method.

`statistics-notable` component

This chart portrays articles considered under the *artículo bueno* (good article) and *artículo destacado* (featured article) denominations that were written by members of the LGBT+ WikiProject, it then uses this information to populate containers with a dynamic animation.

In order to obtain the information, the `getNotableArticlesInfo()` method makes a request to the MediaWiki API through the `getPageContent` method of the `MediawikiService`. This return a string of data logged by members of the WikiProject in a wiki page (in Wikicode). The string is then processed to separate between good and featured articles through the `split` native method.

The array with the two strings is then passed to the `extractnotableElements` method, which uses a regular expression to push each good and featured article into an array. The two arrays are then used to build an object that holds the information.

[3]The object returns a number and not an array of names in order to avoid having to treat personal information through this method.

```
export interface NotableArticles {
    AB: string[],
    AD: string[]
}
```

The HTML loops through this object through an Angular `@for` block to obtain the articles and assign a link to each one of them. This is animated using Angular Animations through the `@popAnimation`.

`statistics-last-articles` component

This component retrieves the last three articles created by members of the project through MediaWiki API. It then uses this info to search for the corresponding Wikidata item to check whether there is an available image to assign to each of them.[4]

The `getLastArticlesInfo` method makes use of the `getLGBTArticleList` service to build a dictionary object with the following structure:

```
this.cardDict[title] = {
extract: '',
image: ''
}
```

This is done through the following methods:

- `assignExtracts(title: string)`, it receives a text extract from an article through the `getPageExtract()` function in the MediaWiki service and then proceeds to assign the extract to a key/value pair in the dictionary that holds article information. This method makes use of the `cropString` function to return only a part of the extract; and of the `removeRefNumbers` function to remove residual characters that conform reference links in the Wikipedia article.
- `assignImages(title: string)`, this method uses both the `getWikidataEntity()` and `getImageUrlFromWdEntity()` functions to obtain a representative image of an article, first by obtaining the Wikidata item ID and then looking for the P18 property in the item if it exists. In the latter situation, it assigns the Wikimedia Commons hotlinked image to the corresponding key in the article dictionary; otherwise it does so with a placeholder.

The component then uses a specific method, `isDictFull` to set a boolean that is used by the view to choose whether to show the loading spinner through an `ngIf` directive.

A `@for` block in the html then loops through the three dictionary entries and populates the corresponding elements.

[4]Each Wikidata item can have an associated image that might not necessarily be used in the corresponding Wikipedia article. Conversely, images on Wikipedia articles can be directly fetched through Wikimedia Commons and not be obtained through Wikidata property fetch.

Form component

The form component consists of the component holding the form that webpage users will make use of in order to contact the LGBT+ WikiProject. It is developed using the native Angular reactive forms module and validators, which is configured in the constructor. The next paragraph explains how each validator is appliead on each input:

- The `pronouns` select field is used to ask contacting users about their preferred pronouns. The only validator this input is subject to is `Validators.required`.
- The `otherPronouns` input field only appears once the `pronouns` option «Sin determinar/otro» has been selected. This is achieved by the existence of a variable `showOtherPronounsField` that is tied to an `@NgIf` structural directive. The variable is modified through the use of the `otherPronounsChosen` function, which is activated every time a change in the `pronouns` select menu happens. This is achieved through the use of the `(change)` directive.
- The `name` pronoun is a normal input box to which the requirement validator is applied. It is also affected by a `maxLength` validator that prevents using more than 70 characters. This is done to prevent cross-site scripting.
- The `email` input field is affected by requirement validation but also by the specific `Validators.email`.
- The wider `reason` textarea element has a `maxLength` validator that prevents it from having more than a set number of characters for security reasons.
- The `wikimediaAccount` checkbox does not have any validators tied to it. The input field below, `wikimediaAccountName` uses a `@NgIf` directive to decide whether it will be shown or not, depending on whether the checkbox above is ticked or not. This is managed through the `showValue`, which receives the name of the field to show as a string, in this case `wikimediaAccount`. The placeholder is dynamically modified through the `concordWikimediaAccountNamePlaceholder` function, which uses a `switch` statement to decide the concordance of the noun «usuario» in the placeholder text based on the selection made by the user in the `pronouns select` element.
- The `attendedEvent` checkbox works similarly, using the same function as described above but having the field `attendedEventName` input field appear only when the relevant checkbox is checked.
- The `readPrivacy` and `readPolicy` checkboxes, which are required for the form to be considered valid.

Other functions that have been implemented include `isFieldInvalid`, which takes the name of a field as an argument and through native form validation variables returns a boolean which is employed to show error messages through an `@NgIf` directive. The `isFormValid` returns a quick boolean that defines whether the whole form can be sent.

Components that are part of the blog module

blog-main component

The aim of the blog-main component is to display a summary of the blog entries so that can users can choose to delve into them and have the full version in another page provided by the blog-post component.

To achieve this, the component makes use of the API Service to get the full list of posts through the getPosts function.[5] This will get the list either through interaction with an API endpoint or through an already cached variable.

```
getPosts() {
    this.apiService.getPosts().subscribe((res) => {
      if (!res || res.length == 0 || typeof res == 'string') {
        this.error = 'Oops, se ha producido un error';
      } else {
        this.sortedPosts = res.sort((a, b) => new Date(b.date).getTime()
        ↪  - new Date(a.date).getTime());
        setTimeout(() => {
          this.loaded = true;
        }, 500);
      }
    })
}
```

As it can be observed in the code above, the function makes use of the sort native function to put the list in ascending order using the date key.

The view then makes use of an Angular @for block to iterate over the post array, which contains objects representing each row. It uses variable interpolation to display each field in the view:

```
@for (item of sortedPosts; track $index) {
    <a class="fill-link" routerLink="/blog/{{item.id}}">
        <div class="blog-post">
            <h2>{{item.title}}</h2>
            <div class="author">{{item.author}}</div>
            <div class="date">{{item.date | dateFormat }}</div>
        </div>
    </a>
}
```

[5]If, in the future, the number of posts grows too significant to handle in a single page, pagination can be introduced to handle this problem.

It includes the use of the date format pipe to avoid showing the timestamp and display a more user-friendly date format instead.

This component makes use of the `authGuard` guard to limit the access of non-authorised users, refer to section guards for more information.

`blog-post` component

The `blog-post` component's purpose is to show a specific blog post fully. This is done by using a dynamic url that will make an SQL query into the `blog_posts` table of the database and fetch a row with a specific `id` parameter.

Similarly to `blog-main`, this component obtains the data for a specific blog post through an API service function that performs the query on the database. To be able to do this, the constructor fetches the value of the url parameter id in the constructor:

```
constructor(
    private apiService: ApiService,
    private activatedRoute: ActivatedRoute
  ) {
    this.postId = this.activatedRoute.snapshot.paramMap.get('id')
    console.log(this.postId)
}
```

Then, once the GET request is done, it assigns the response to local variables so that they can be more easily managed.

The view then uses Angular-specific variable interpolation to build a `.blog-post` element.

```
<div class="blog-post" [@popAnimation]="loaded" [ngClass]="{hidden:
↪   !loaded}">
    @if (loaded) {
    <h2>{{title}}</h2>
    <div class="post-info">
        <div class="thumb-img-container">
            <img src="./../assets/imgs/Barba-wikiproyecto-lgbt.svg"
            ↪   alt="Título" class="post-image">
        </div>
        <div class="author-and-thumb">
            <span class="author">por <span
            ↪   class="author-name">{{author}}</span></span>
            <span class="date">{{date | dateFormat }}</span>
        </div>
    </div>
    <div class="post-content">
        {{content}}
    </div>
    }
</div>
```

Similarly to blog-main, the view makes use of the date format pipe to avoid showing the date as a timestamp.

blog-admin component

This component's purpose is to display the full list of blog posts so that site administrators and contributors can modify, add or remove a specific blog post according to their needs.

It works in a similar way to the blog-main component: it makes use of the API service to fetch the full list of articles and the stores them locally as an array. It then uses a @for block to iterate over the array in the view to, in this case, build a table displaying each one of the existing blog posts as a row.

```
<tbody class="blog-entries">
    @for (item of infoArray; track $index) {
    <tr id="post{{item.id}}">
        <td>{{item.id}}</td>
        <td>{{item.date | dateFormat }}</td>
        <td>{{item.author}}</td>
        <td>{{item.title}}</td>
        <td><a href="/blog/{{item.id}}" target="_blank">Enlace</a></td>
        <td>
            <button><a
            ↪  routerLink="/blog-edit/{{item.id}}">Editar</a></button>
            <button id="delete{{item.id}}"
            ↪  (click)="deletePost(item.id)">Eliminar</button>
        </td>
    </tr>
    }
</tbody>
```

In this case, however, variable interpolation is used to build the link that takes to the full text as well as the *edit* and *remove* button links. The former takes to the blog-edit component that takes to the corresponding edit page for that blog entry by including the id parameter in the URL. The latter carries out a DELETE query to the database so that the specific row is removed.

```
deletePost(id: number): string | void {
  const deleteButton = document.querySelector(`#delete${id}`)
  if (deleteButton?.textContent == 'Eliminar') {
    return deleteButton.textContent = '¿Seguro?';
  }
  let stringId = id.toString()
  this.apiService.deletePost(stringId).subscribe((res) => {
    console.log('Component response:', res);
    if (res?.success) {
      this.removeRow(stringId);
    } else if (deleteButton) {
      deleteButton.textContent = 'Error';
    }
  })
}
```

In this latter process, an additional confirmation step was added. The function in charge of of this action in the logic is the `deletePost` function, which performs two primary actions: asking for confirmation before deletion and then making an API call through the API service to delete the post if confirmed. Depending on the API response, it either removes the post's row from the UI or displays an error message.

Additionally, the `getAdminName` function is designed to fetch the login status of the logged user through the API service `getLoginStatus` function and display a personalized greeting based on the response:

```
getAdminName() {
  this.apiService.getLoginStatus().subscribe((res) => {
    console.log(res);
    this.adminUsernameText = res?.displayName ? `¡Hola,
     ↪  ${res.displayName}!` : '';
  })
}
```

In the view, this is handled through variable interpolation:

```
<div class="admin-username">{{adminUsernameText}}</div>
```

It also includes an option to log out as a button which is located at the bottom of the page, as well as a choice to create a new entry, which implies navigating to the `blog-edit` component and while not passing an `id` in the URL.

`blog-edit` component

The aim of this component is to allow administrators and contributors to fill out new post or edit existing post information, which will in turn make an INSERT or PUT request to the database so that the corresponding rows in the `blog_posts` table can be appropriately updated.

Similarly to `blog-post` component, it fetches the parameter from the URL. In this case, however, it uses the information to populate the values of a form, which are stored as local variables after obtaining the response from the database server:

```
this.blogForm = this.formBuilder.group({
    author: new FormControl(this.postId ? this.author : '',
↪   [Validators.required, Validators.maxLength(75)]),
    date: new FormControl(this.postId ? this.date : '',
↪   [Validators.required]),
    title: new FormControl(this.postId ? this.title : '',
↪   [Validators.maxLength(255)]),
    content: new FormControl(this.postId ? this.content : '',
↪   [Validators.maxLength(10000)]),
})
```

If no id parameter is entered through the URL, the form will be free of information, thus allowing to insert a new row instead of modifying an existing one.

Insert and put requests to add or modify, respectively, blog posts (as rows) stored in our database are managed through the `onSubmit` function.

To build the body of the insert request, the function calls the API service `addPost` function and populates the corresponding arguments with form information gathered through the `formBuilder` object.

```
  onSubmit(): void {
    this.showSubmitSpinner = true;
    const formValues = this.blogForm.value
    const formParams = [
      formValues.date || this.getCurrentDate(),
      formValues.author,
      formValues.title,
      formValues.content
    ] as const;
    if (!this.postId) {
      this.apiService.addPost(...formParams).subscribe((res) => {
        this.showSubmitSpinner = false;
```

```
          this.responseMessage = res?.success ? '¡Tu post ha sido
           ↪  publicado!' : '¡Parece que ha habido un error al añadir el
           ↪  post, prueba de nuevo más tarde!';
        })
      } else {
        this.apiService.editPost(this.postId,
         ↪  ...formParams).subscribe((res) => {
          this.showSubmitSpinner = false;
          this.responseMessage = res?.success ? '¡Tu post ha sido
           ↪  editado!' : '¡Parece que ha habido un error al editar el
           ↪  post, prueba de nuevo más tarde!';
        })
      }
    }
```

`onSubmit` manages both PUT and POST requests, as it uses the existence of an id in the URL parameters (which is obtained in the constructor) to determine whether a new post is being created or an existing one is being modified. In the latter case, the form fields are populated through a call to the API service, specifically to the `getPostInfo` method, which carries out a query in the database and retrieves corresponding blog post information. Population happens through the Angular reactive forms native `patchValue` method. This is carried out through a call in the constructor.

The `getErrorMessage` function deals with the possible error messages that arise through form control analysis. The view listens to this function in order to decide which error to show, in a similar way as in the form component.

Three functions were specifically created to deal with situations where dates needed to be treated:

- `timeStampToDate` transforms a timestamp format through the `toISOString` JavaScript method.
- `dateToTimestamp` does the opposite. In this case, it employs the `getTime` method.
- `getCurrentDate` returns the current date in `YYYY-MM-DD` format, which is needed for the `date` field to be able to process the string in the adequate format.

In the view, an `@if` block is used to determine whether the `id` of the post is show or not, which is decided upon the existence of such parameter in the URL. This information is stored in the `postID` function.

```
@if (postId) {
<div class="post-id">ID: {{postId}}</div>
}
```

A bootstrap modal is used to provide feedback after submission. The modal's main body of text is managed through the `responseMessage` variable (consult `onSubmit` code above). A `navigateToPanel` function was also implemented so that the process of closing the modal through the option that allows to return to the administrator panel and navigating to it happens through a smooth transition.

The component makes use of the `authGuard` guard to restrict the access of non-authorised users, refer to section guards for more information.

`login` component

The login component holds a simple interface that lets users know that authentication will be carried out through the MediaWiki OAuth system. It consists of a simple container that holds a brief description and a login button that redirects to the API endpoint specified in charge of dealing with the authentication through the MediaWiki services.

```
<button type="button" class="btn btn-primary mx-auto"
↪   data-bs-toggle="modal" data-bs-target="#formConfirm">
    Iniciar sesión
</button>
```

OAuth configuration and authentication is described in the corresponding MediaWiki OAuth configuration section.

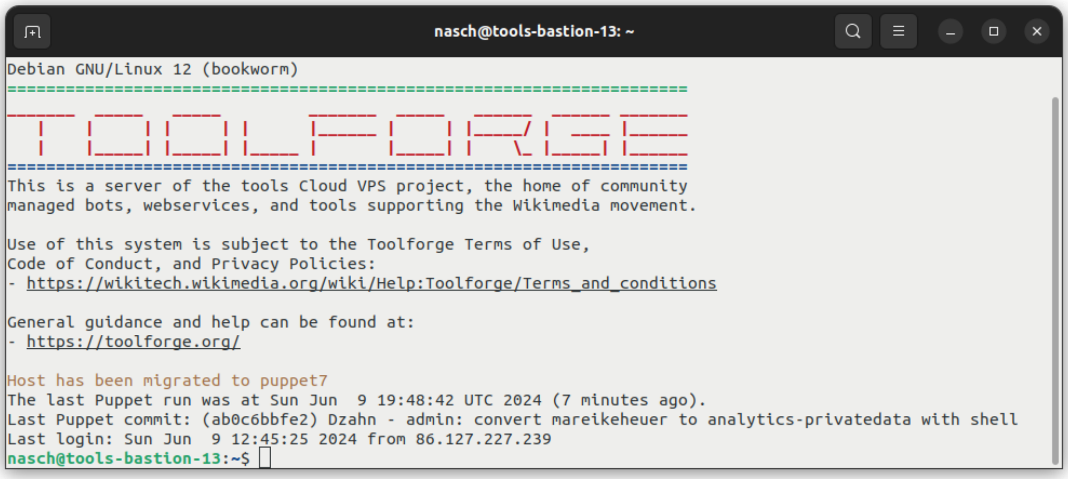
Web hosting and deployment

A request to Toolforge, the Wikimedia cloud service, was made in late March 2024 in order to sustain web hosting for the project. Toolforge is a hosting environment that is designed for developers that create services that are considered beneficial to the Wikimedia movement. The cloud system enables its users to perform ad hoc analytics, manage bots, operate web services and develop tools that assist Wikimedia editors and volunteers. It also allows for access to various data services and an option to build databases. It is maintained by a dedicated team of Wikimedia Foundation staff and volunteers. Access to toolforge was granted on the 10th of May.

Access to the service and deployment

In order to be able to access the remote server locally, an SSH key was linked to the relevant Toolforge account. This was done through OpenSSH. Remote access to Toolforge is carried out through the following command.

```
ssh -i /home/nasch/.ssh/<ssh> <login name>@login.toolforge.org
```



```
Debian GNU/Linux 12 (bookworm)
=====================================================================
_______  _____   _____         ______  _____  _____  _____  ______
   |    |     | |     | |      |______ |     | |____/ |  ___ |______
   |    |_____| |_____| |_____ |       |_____| |    \_|_____| |______
=====================================================================
This is a server of the tools Cloud VPS project, the home of community
managed bots, webservices, and tools supporting the Wikimedia movement.

Use of this system is subject to the Toolforge Terms of Use,
Code of Conduct, and Privacy Policies:
- https://wikitech.wikimedia.org/wiki/Help:Toolforge/Terms_and_conditions

General guidance and help can be found at:
- https://toolforge.org/

Host has been migrated to puppet7
The last Puppet run was at Sun Jun  9 19:48:42 UTC 2024 (7 minutes ago).
Last Puppet commit: (ab0c6bbfe2) Dzahn - admin: convert mareikeheuer to analytics-privatedata with shell
Last login: Sun Jun  9 12:45:25 2024 from 86.127.227.239
nasch@tools-bastion-13:~$
```

Figure 10: Screenshot of Toolforge after login

An alias `toolforge` was established in `~/.bashrc` to accelerate the remote login process.

A new tool was created through the toolsadmin application. Access to the tool inside Toolforge is carried out through a Toolforge-specific `become` command which switches to the appropriate user. In this case:

```
become wmlgbt-es-web
```

Once inside the tool, the required configuration to be able to set up a Node.js server inside Toolforge was carried out. In order to be able to do this, a `package.json` file inside the `wwww/js/` path was created with the following content:

```
{
  "name": "wikiproyecto-lgbt-webpage",
  "version": "1.0.0",
  "description": "",
  "main": "dist/server/server.mjs",
  "scripts": {
    "start": "node dist/server/server.mjs",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Nacaru",
  "license": " GPL-3.0-only"
}
```

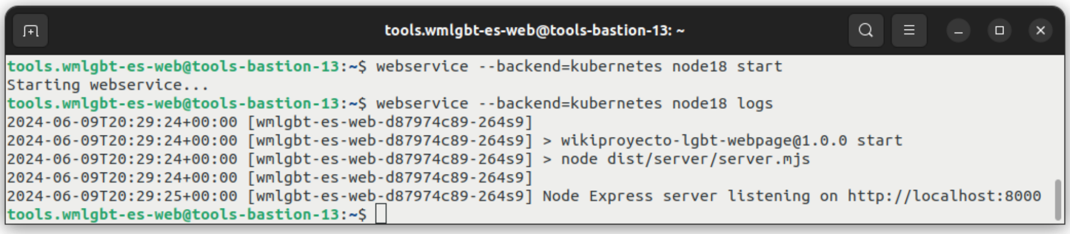
The `server.mjs` file corresponds to the main executable file for the deployed version of the Angular project. This approach (moving the deployed application into Toolforge) was taken after being unable to build the application through conventional means inside the Toolforge server due to lack of memory resources.

In order to do this comfortably, `rsync` was used to copy the deployed `dist` folder into the tool's toolforge environment as follows:

```
rsync --delay-updates --delete-after -F --compress --archive --no-owner
↪  --no-group --rsh='/usr/bin/ssh -S none -o StrictHostKeyChecking=no
↪  -o UserKnownHostsFile=/dev/null' --rsync-path='sudo -u
↪  tools.wmlgbt-es-web  rsync' --chmod=Dug=rwx,Dg+s,Do=rx,Fug=rw,Fo=r
↪  <path to local files> wikiproyecto-lgbt-web/
↪  login.toolforge.org:/data/project/wmlgbt-es-web/www/js/dist
```

Once this has been carried out, the project is run by a node18 container within Toolforge's Kubernetes infrastructure, which is accessed through Toolforge's `webservice` utility.

```
webservice --backend=kubernetes node18 start
```



```
tools.wmlgbt-es-web@tools-bastion-13:~$ webservice --backend=kubernetes node18 start
Starting webservice...
tools.wmlgbt-es-web@tools-bastion-13:~$ webservice --backend=kubernetes node18 logs
2024-06-09T20:29:24+00:00 [wmlgbt-es-web-d87974c89-264s9]
2024-06-09T20:29:24+00:00 [wmlgbt-es-web-d87974c89-264s9] > wikiproyecto-lgbt-webpage@1.0.0 start
2024-06-09T20:29:24+00:00 [wmlgbt-es-web-d87974c89-264s9] > node dist/server/server.mjs
2024-06-09T20:29:24+00:00 [wmlgbt-es-web-d87974c89-264s9]
2024-06-09T20:29:25+00:00 [wmlgbt-es-web-d87974c89-264s9] Node Express server listening on http://localhost:8000
tools.wmlgbt-es-web@tools-bastion-13:~$
```

Figure 11: Calling the `logs` command allows us to see the state of the execution

In order to facilitate the building-moving-deployment mechanism, two Unix Shell executable files (`.sh`) were created. One is to be executed inside Toolforge's remote access server, which calls the `webservice` code above and restarts the hosting service. The other one is located in the project files and was configured to be callable locally: it calls the `ng build` command, moves the deployed files through `rsync` as explained above and then runs the shell script inside Toolforge. This allows the whole process to be carried out with one single command.

Database

In order to store blog information, the setup of a database was envisaged as part of the development of the project.

The hosting service that we have used to deploy the project, Toolforge, includes a user database service for small databases of up to 25 giga bytes of storage. Taking into account that our database would only be used to store blog information, it was estimated that such quantity would be sufficient for our purposes.

Toolforge databases are user-managed: this means users have all privileges and access to all grant options on their databases.

Setup

To set up the database, our tool account was accessed from within Toolforge (see previous section for more details). Once in our tool, we use `mariadb` commands to set up the required `replica.my.cnf` credential file:

```
mariadb --defaults-file=$HOME/replica.my.cnf -h
↪   tools.db.svc.wikimedia.cloud
```

Following this action, a new database was created. In this case, it was named `wmlgbt_es`:[6]

```
MariaDB [(none)]> CREATE DATABASE CREDENTIALUSER__wmlgbt_es;
```

Once the database was created. It was thoroughly tested through MariaDB's CLI, which can be accessed from inside Toolforge's tool:

```
mariadb --defaults-file=$HOME/replica.my.cnf -h
↪   tools.db.svc.wikimedia.cloud CREDENTIALUSER__wmlgbt_es
```

Once inside the database CLI, we created a new table to store blog post information:

```
MariaDB [CREDENTIALUSER__wmlgbt_es]> CREATE TABLE blog_posts ( id INT
↪   AUTO_INCREMENT PRIMARY KEY, date TIMESTAMP DEFAULT
↪   CURRENT_TIMESTAMP, author VARCHAR(75), title VARCHAR(255), content
↪   TEXT );
```

As it can be observed in the command above, certain specifications were provided for the columns:

- The `id` (type `integer`) column was configured as the primary key and its value incrementally assigned automatically.
- The `date` (type `timestamp`) column uses the current timestamp as default value.
- The `author` (type `string`) column employs a max-length character limit of 75.
- The `title` (type `string`) column has a max-length character limit of 255.
- The `content` (type `string`) has no character limit.

[6]For security reasons, database username and other similar sensitive information have been ommited from this report, using the `CREDENTIALUSER` placeholder instead.

Once this was done, a call was made to check that the columns were correctly constructed:

```
MariaDB [CREDENTIALUSER__wmlgbt_es]> SHOW COLUMNS FROM blog_posts;
+---------+--------------+------+-----+--------------+----------------+
| Field   | Type         | Null | Key | Default      | Extra          |
+---------+--------------+------+-----+--------------+----------------+
| id      | int(11)      | NO   | PRI | NULL         | auto_increment |
| date    | timestamp    | NO   |     | ctimestamp() |                |
| author  | varchar(75)  | YES  |     | NULL         |                |
| title   | varchar(255) | YES  |     | NULL         |                |
| content | text         | YES  |     | NULL         |                |
+---------+--------------+------+-----+--------------+----------------+
5 rows in set (0,002 sec)
```

Administrators table

Additionally, the MariaDB database holds a table for authentication purposes. This was created through Toolforge's MariaDB client, with the following columns:

```
CREATE TABLE administrators (
    user_id INT,
    AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
↪   CURRENT_TIMESTAMP
  );
```

The `administrators` table holds information related to administrator's usage of the page. This information is used to allow certain users with administrators rights to create, modify and delete blog posts. The provided API endpoint is used to request the existence of an administrator's username based on the information described in the MediaWiki OAuth token as detailed in the backend section.

Backend

Backend setup was managed through Angular's server-side rendering (SSR) configuration file: `server.ts`. It was built using the framework Express, since our server is built using Node.js. This allows us to eventually separate the `/api` endpoints into a different backend applications if necessary.

To make database interaction more dynamic, the mysql2 dependency was also installed. The library allows for simpler syntax when using methods that facilitate the task of

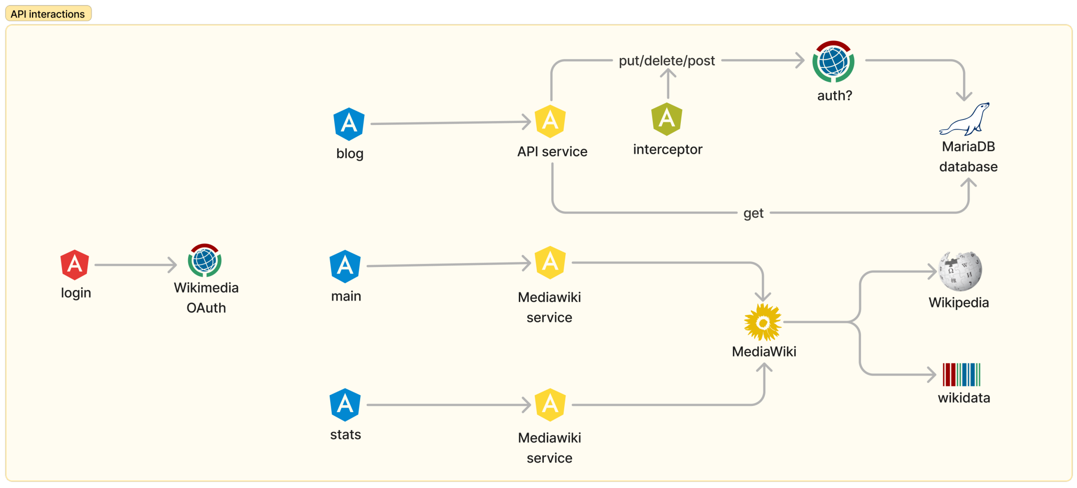


Figure 12: Diagram showing MariaDB and MediaWiki API interactions

requesting, inserting, deleting and modifying information on the database through SQL commands that are passed to the methods as strings. When parameters are used in database queries these are passed through prepared SQL statements to prevent SQL injection attacks.

The following methods are described throughout this section.

The `getRow(id: string)` method takes an `id` as a string and returns the row of the `blog_posts` table that corresponds to that `id`. Allowing the method to take the argument will facilitate future service interactions:

```
async function getRow(id: string) {
  const connection = await mysql.createConnection(
    credentials
  );

  try {
    const [rows, _] = await connection.execute('SELECT * FROM blog_posts
    ↪  WHERE id = ?', [id]);
    return rows;
  } catch (error) {
    console.error('Error retrieving row:', error);
    throw error;
  } finally {
    await connection.end();
  }
}
```

The getTable method makes a GET request to obtain every row in the blog_posts table:

```
async function getTable() {
  const connection = await mysql.createConnection(
    credentials
  );

  try {
    const [rows, _] = await connection.execute('select * FROM
    ↪  blog_posts');
    return rows
  } catch (error) {
    console.error('Error retrieving rows:', error);
    throw error;
  } finally {
    await connection.end();
  }
}
```

The insertRow function, which takes blog date, an author, a title and content information to construct an insertion query to the database table.

```
async function insertRow(date: string, author: string, title: string,
↪  content: string) {
  const connection = await mysql.createConnection(credentials);

  try {
    const [result] = await connection.execute(
      'INSERT INTO blog_posts (date, author, title, content) VALUES (?,
      ↪  ?, ?, ?)',
      [date, author, title, content]
    )
    return result
  } catch (error) {
    console.error('Error inserting row:', error);
    throw error;
  } finally {
    await connection.end();
  }
}
```

The updateRow function sends an update to the blog_posts table to find a specific blog post in the MariaDB database and, if authorised, executed an SQL UPDATE statement to set new values for the corresponding columns. If successful, it returns the result of the execution and closes the connection.

```
async function updateRow(date: string, author: string, title: string,
↪  content: string, id: string) {
  const connection = await mysql.createConnection(credentials);

  try {
    const [result] = await connection.execute(
      'UPDATE blog_posts SET date = ?, author = ?, title = ?, content =
      ↪  ? WHERE id = ?',
      [date, author, title, content, id]
    );
    return result;
  } catch (error) {
    console.error('Error updating row:', error);
    throw error;
  } finally {
    await connection.end();
  }

}
```

This `deleteRow` function deletes a specific blog post in the MariaDB database through the identification of the `id` column, which is passed as a paremeter. After it executed the `DELETE` statement, it returns the result of this action.

```
async function deleteRow(id: string) {
  const connection = await mysql.createConnection(credentials);

  try {
    const [result] = await connection.execute(
      'DELETE FROM blog_posts WHERE id = ?', [id]
    );
    return result;
  } catch (error) {
    console.error('Error deleting row', error)
    throw error;
  } finally {
    await connection.end();
  }

}
```

The checkAdminStatus function verifies if a given username belongs to an administrator in the administrators table. This time, it uses SELECT to check for the presence of the username (passed as an argument) in the table. If that is the case, it returns a boolean indicating whether the user is an admin or not.

```
async function checkAdminStatus(username: string) {
  const connection = await mysql.createConnection(credentials);

  try {
    const [rows]: [any[], mysql.FieldPacket[]] = await
    ↪  connection.execute('SELECT 1 FROM administrators WHERE username
    ↪  = ? LIMIT 1', [username]);
    return rows.length > 0;
  } catch (error) {
    console.error('Error retrieving row:', error);
    throw error;
  } finally {
    await connection.end();
  }
}
```

The isAuthorized function checks whether there is an user object in the stored in current session (as a cookie that has been passed as a parameter). If a user is found, it logs the attempt and calls the previously-described checkAdminStatus function. If the user is not logged in, it logs an attempt by a logged-out user to access the endpoint and returns false.

```
async function isAuthorized(req: express.Request) {
  if (req?.session?.user) {
    console.log("Checking admin status for user",
    ↪  req.session.user.displayName)
    return checkAdminStatus(req.session.user.displayName)
  }

  console.log("Logged out user attempted to use protected endpoint")
  return false;
}
```

API endpoints to interact with MariaDB

Using these functions, the Rest API endpoints were defined as follows:

The /api/blog/:id endpoint is used for PUT, GET and DELETE requests that modify, obtain and remove blog post rows on the website, respectively. For the GET which makes use of the above-mentioned `getPostInfo` function described above; for the PUT request, it uses the `updateRow` function; for the DELETE request, it uses the `deleteRow` function.

```
server.get(`/api/blog/:id`, (req, res) => {
    const id = req.params.id
    getPostInfo(id).then((rows) => {
      res.send(rows)
    })
});
```

Take into account that DELETE and PUT requests are only active if proper admin authentication has been carried out. To check this, the functios make use of the `isAuthorized` function as follows:

```
isAuthorized(req).then((authorized) => {
  if (!authorized) {
    res.status(403).json(unauthorizedError);
    return
  }
  /.../ )
```

The `/api/blog_posts` endpoint uses the `getTable` function to return all the posts:

```
server.get('/api/blog_posts', (req, res) => {
    getTable().then((rows) => res.send(rows))
});
```

The `/api/blog` endpoint is used to add new rows to the `blog_posts` table through a POST request by using the `insertRow` function described above:

```
server.post('/api/blog', (req, res) => {
    const { date, author, title, content } = req.body;
    insertRow(date, author, title, content)
      .then((result) => res.status(201).send({ id: result }))
      .catch((error) => res.status(503).send('Error inserting row: ' +
      ↪  error.message))
})
```

The `/api/user/` endpoint handles GET requests to return information about the currently logged-in user by calling the `checkAdminStatus` function when a user session exists. The latter then uses the user's `displayName` to check whether the user is an administrator in the corresponding table in the database. The response is given as a JSON object that can be consulted through the provided API endpoint. It returns a non-speciifc 400 status code if the user is not logged in.

```
server.get("/api/user", (req, res) => {
  if (req?.session?.user) {
    checkAdminStatus(req.session.user.displayName).then((isAdmin) => {
      res.json({
        displayName: req.session.user.displayName,
        isAdmin: isAdmin,
      })
    })
  } else {
    res.status(400).json({ reason: 'Not logged in' });
  }
})
```

MediaWiki OAuth endpoints

Additional endpoints were provided to deal with the actions related to MediaWiki authentication. These are described as follows:

The `/login-mediawiki` endpoint, which redirects to the URL accepted as described in the MediaWiki OAuth documentation:

```
server.get("/login-mediawiki", function (req, res) {
  res.redirect(req.baseUrl + "/auth/mediawiki/callback");
});
```

The `/logout` endpoint. It deletes information related to user session and then redirects to the homepage.

```
server.get("/logout", function (req, res) {
  delete req.session.user;
  res.redirect(req.baseUrl + "/");
});
```

The `auth/mediawiki/callback` endpoint. As required by MediaWiki configuration, the URL that carries out the OAuth protocol must follow this path. The function makes use of the `passport` library to carry out authentication through a specific MetaWiki page. When a user is redirected back to this route after attempting to log in via MediaWiki, it uses `passport.authenticate` to verify the authentication result. If there is an error, it passes the error to the next middleware. If authentication fails (i.e., no user is found), it redirects the user to the login page, otherwise it logs the user in, stores the user information in the session, and then redirects the user to the blog administration page.

```
  server.get("/auth/mediawiki/callback", function (req, res, next) {
    passport.authenticate("mediawiki", function (err: any, user: any) {
      if (err) {
        return next(err);
      }
      if (!user) {
        return res.redirect(req.baseUrl + "/login");
      }

      req.logIn(user, function (err) {
        if (err) {
          return next(err);
        }
        req.session.user = user;
        res.redirect(req.baseUrl + "/blog-admin");
      });
    })(req, res, next);
  });
```

Please refer to the API service section for more information on how the service in charge of interacting with the functions above was designed.

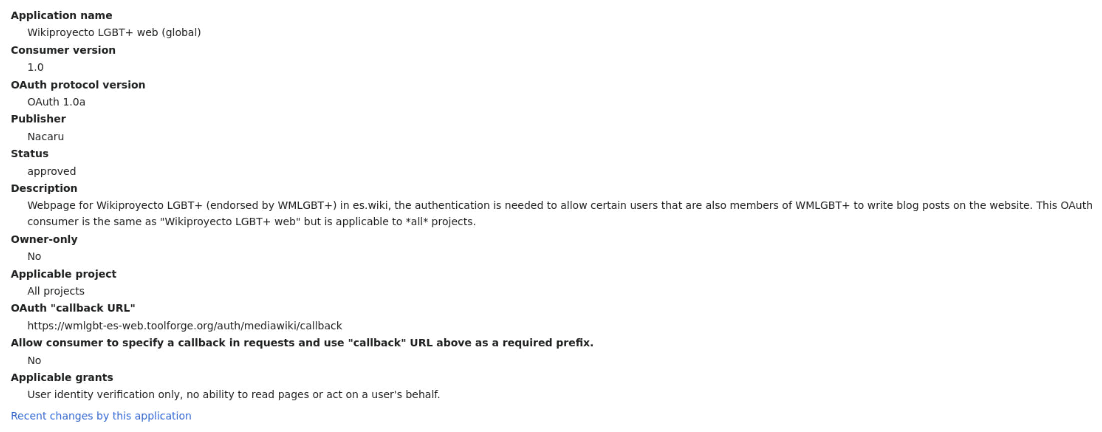
MediaWiki OAuth configuration

In order to provide a reliable authentication method that did not require users to go through the process of creating a specific account for the website while maintaining a system they would be familiar with, the MediaWiki OAuth system was used. OAuth is a widely used standard, that well-known web projects such as Facebook or Google use; the fact that it has a high level of usage aids in long-term support, security and lowers maintenance for the website. In this case OAuth 2 was used.

OAuth allows for the request of permissions to a specific user that holds an account in any of the MediaWiki projects. The system does not require acknowledging the user password, nor does it make use of it throughout the authentication process; it requires consent from an already-signed in user account to perform a verification process that the website uses to ascertain administrator rights.

To do this, an application to register the application as consumer was carried out. In the request, it was specified that it would be use only for verification purposes.

The system relies on the use of two types of credentials: an application token, which acts as a public ID; and an application secret, which acts as a hidden password. Each user that goes through MediaWiki OAuth gets a user-specific credential: the access token. The token is obtained by sending a GET request to the MediaWiki Meta page `Special:OAuth/initiate`. It returns a `JSON` object with the `token` and `key` fields, which correspond to the request token and secret.



Application name
Wikiproyecto LGBT+ web (global)
Consumer version
1.0
OAuth protocol version
OAuth 1.0a
Publisher
Nacaru
Status
approved
Description
Webpage for Wikiproyecto LGBT+ (endorsed by WMLGBT+) in es.wiki, the authentication is needed to allow certain users that are also members of WMLGBT+ to write blog posts on the website. This OAuth consumer is the same as "Wikiproyecto LGBT+ web" but is applicable to *all* projects.
Owner-only
No
Applicable project
All projects
OAuth "callback URL"
https://wmlgbt-es-web.toolforge.org/auth/mediawiki/callback
Allow consumer to specify a callback in requests and use "callback" URL above as a required prefix.
No
Applicable grants
User identity verification only, no ability to read pages or act on a user's behalf.
Recent changes by this application

Figure 13: Submitted consumer OAuth details

In this case, the system will only be used with the aim to acknowledge the authenticity of the user, and compare that information to what is described in the administrators table through a SQL query.

To be able to deal with the process more easily on our server, the following dependencies were installed:

- Passport (version 0.5.0)
- Passport-MediaWiki-OAuth
- Express session

In order to allow our application to use cookies for the purposes described in this sections, `commonEngine` providers were configured as follows:

```
providers: [
  { provide: APP_BASE_HREF, useValue: baseUrl },
  { provide: REQUEST, useValue: req },
  { provide: RESPONSE, useValue: res },
],
```

Wikimedia OAuth was setup in the backend through a specific strategy that makes use of the Passport framework to build a profile with the appropriate credentials, made up by the correpsonding consumer key and secret.[7] When a user attempts to authenticate, the strategy receives an access token and a token secret, along with the user's profile information from MediaWiki.

```
passport.use(new MediaWikiStrategy(
  {
```

[7]This information is not available in the public repository for security purposes. It is imported as a variable `oauthCredentials` from a file that is excluded from the repository files.

```
    consumerKey: oauthCredentials.consumer_key,
    consumerSecret: oauthCredentials.consumer_secret,
  },
  function (token: string, tokenSecret: string, profile: any, done: any)
  ↪  {
    profile.oauth = {
      consumer_key: oauthCredentials.consumer_key,
      consumer_secret: oauthCredentials.consumer_secret,
      token: token,
      token_secret: tokenSecret
    };
    return done(null, profile);
  }
));
```

The function provided as the second argument processes the authentication result and attaches the OAuth credentials to the user's profile object and then calls the done callback. Session is provided through Passport sessions.

Refer to the MediaWiki OAuth endpoints sections for a description of how session authentication is then used by the functions that defined the API endpoints to allow or refuse certain API interactions.

`with-credentials` interceptor

In order to simplify the action of attributing options to certain methods that send queries through the API service, an Angular interceptor was created.

```
export const withCredentialsInterceptor: HttpInterceptorFn = (req, next)
↪  => {
  if (req.method === 'PUT' || req.method === 'POST' || req.method === '
  ↪  DELETE') {
    const clonedRequest = req.clone({
      withCredentials: true
    });
    return next(clonedRequest)
  }

  return next(req)
};
```

The `withCredentials` Interceptor ensures that Angular HTTP requests with methods PUT, POST, or DELETE include credentials (such as cookies or authorization headers) by cloning the request and setting the `withCredentials` to true. For all other request methods, it simply forwards the original request unchanged. This ensures that credentials as cookies are sent together with the appropriate requests, as they are configured to need the authentification to work in the backend.

Pipes

Date format pipe

In the database, blog posts dates are stored as timestamps. This format is handy when it comes to storing and treating data long-term, however, it might not be considered user-friendly due to its difficult readability.

To solve this issue, a pipe aimed at formatting dates in different components, was constructed.

```
export class DateFormatPipe implements PipeTransform {

  transform(value: Date): string {
    if (!value) return '';
    const date = new Date(value);
    const day = ('0' + date.getDate()).slice(-2);
    const month = ('0' + (date.getMonth() + 1)).slice(-2);
    const year = date.getFullYear();

    return `${day}-${month}-${year}`;
  }
}
```

The main function `transform` takes a `Date` type of argument and then applies the `getDate`, `getMonth` and `getFullYear` methods to modify the format, which is then returned, with each corresponding variables placed through string interpolation.

Guards

Angular guards are interfaces that allow users to control navigation to and from a route, acting as gatekeepers that enable or prevent access based on a number of conditions.

For this project, a guard that acts on pages that require authentication was setup.

Auth guard

In this case, the guard is to be applied to the pages are that are supposed to be for admin user only: these being the `blog-edit` component and the `blog-admin` component. To do this, the `resolveFn` guard function was used, as it allows the prefetching of data before a route is activated, ensuring data is available when the route is loaded.

```
export const authGuard: ResolveFn<boolean> = (route, state) => {
  const router = inject(Router);
  const apiService = inject(ApiService);

  return apiService.isAdmin().pipe(
    map((res) => {
      console.log(res)
      if (res) {
        console.log('Allowing access to admin panel: ', res)
        return true; // Allow navigation
      } else {
        console.log('Forbidden access to admin panel', res)
        router.navigate(['/login']); // Redirect to login
        return false; // Deny navigation
      }
    })
  );
};
```

The function uses the `isAdmin` method of the API service to determine if the user is an admin through subscription. It returns an observable boolean, using the `map` operator to process the result: If the user is an admin, navigation is allowed by returning `true`, otherwise the user is redirected to the login page via router navigation and it returns `false`, denying navigation to the user.

Future improvements

Due to the limited amount of time alloted for this project, certain implementations had to be ommited from the final version. The following list describes them:

- Dark mode: a dark mode that allows users to switch the colour palette of the website according to either browser preferences or through manual means.
- Social network feed: a display of social media activity by the WMLGBT+ user group could be displayed on the website in the shape of a news feed provided by a specific gadget.

- Interactive map: the LGBT+ WikiProject celebrates a monthly event called «País del mes», which encourages focusing their monthly activity to improving the quality of LGBT+ articles of our specific country or event. An interactive map using the charts.js-chart-geo extension could be built for display on the website.
- Adding new blog admins from the web: as of the last version, new blog administrators can only be added through an INSERT query into MariaDB's database through MariaDB CLI. A functionality to add new web administrator from the web could be implemented.
- Mail server: the last version of the website does not have a mail server. This is because the WMLGBT+ user group does not own an email server at the project's release time.
- Domain: the current domain could be simplified by using an acquired web domain. This would imply, however, funding the costs though financial support from the WMLGBT+ user group.

Manual of style bot

The project that implements a bot in the Spanish Wikipedia that applies changes to the manual of style is coinceived under a Node.js application that uses a comprehensive framework that simplifies bot code.

Technologies and dependencies

This section describes the different technologies that have been used to build the MoS bot.

The bot is written in TypeScript, using the Node.js runtime environment, divided into modules and compiled into a single Javascipt file to be run through Node.js. Interactions are done through the Wikimedia API, which is facilitated through the mwn framework.

Other dependencies include:

- Jest, which is used for regular expression testing. Jest needs the babel-jest and types/jest dependencies to work with typeScript.
- Ora, which is used to display a rotating spinner in the console.
- Cli-progress, which is used to display a loading bar in the console.

File structure and build pipeline

For the file structure, the project follows the construction as described in the next list:

- The **src** folder contains the main files that describe the script of the tool. They have compartimentalised so as to make it easier to understand for potential contributors to the code. The **categories** folder contains the Wikipedia categories used to filter the action of the bot, the **models** folder contains interfaces and type definitions, the **regex** folder contains the regular expressions that the bot makes to filter page wikicode.
- The **tests** folder includes testing code. The following section provides further information as to how testing is carried out.
- The **build** folder contains the build application after it has been compiled by TypeScript into JavaScript files.

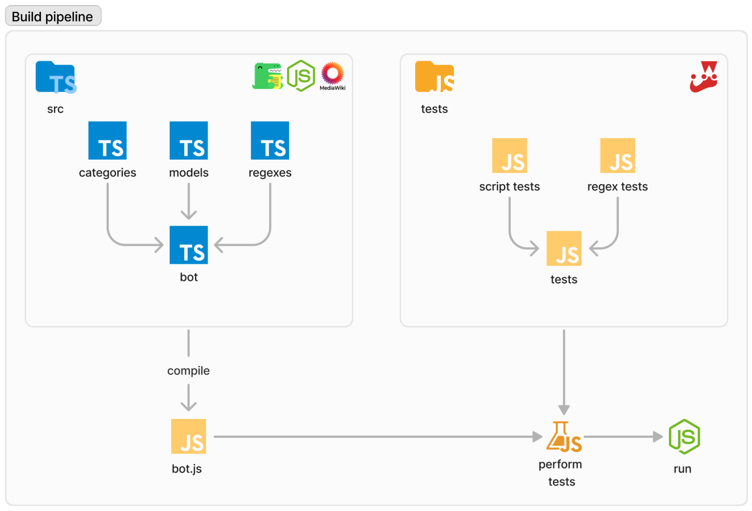


Figure 1: MoS bot build pipeline

The relevant figure describes as a diagram the build pipeline for the MoS bot project.

Bot action: removing internal links that take to date articles

An action for the bot was requested through the bot authoritation board.[1] The purpose for this action is to systematically apply the *WP:ENLACESFECHAS* rule on the Spanish Wikipedia.[2]

The rule states the following:

> *As a general rule, internal links should not be added to dates: not in the introductory section of any article (whether it is a biography or not), nor in the rest of the body of the article or in the infobox. As an exception to this rule, they can be added in articles that deal with topics related to the calendar (such as «Gregorian calendar», «January», «14th of July», «1808», «XIX century», etc.), following the rest of the rules as stablished in this section.*

In the articles, internal links can be added either manually or through a template. A standard internal link in Wikicode is represented though double square bracket notation: `[[<link>]]`. They can also be piped to take to a different article than the one displayed in the text, following the structure `[[linked article|text]]`. For example:

[1]The application for bot purpose was evaluated by the Spanish Wikipedia community and approved unanimously. The process is described on this page.

[2]The rule can be consulted in the corresponding section of the eswiki Manual of Style.

- `[[1994]]` will take to the article about the year 1994.
- `[[1994|that year]]` will take to the same article, but, instead of 1994, the text «that year» will be displayed on the page.

Internal links can also be expressed through templates that contain internal links. The following templates have an option to bear an internal link to a date as part of their code:

- The `{{siglo}}` template. In this case, the internal link to a date can be added by assigning a value of `1` to the fourth parameter.
- The `{{fecha}}` template, which can have an internal link to a date by using the parameter `link` and giving it the value of `sí` or `si`.
- The `{{julgregfecha}}` template, which requires the parameter `link` to have a value of `"true"` (quotation marks included) to implement an internal link to a date.

The bot then uses regular expressions to analyse the string and identify the existence of any of the cases above. This process will be explained in the following section.

Bot code

As described in the build pipeline section, the bot is divided in different files to allow potential contributors to understand the code more easily. These files correspond to two of the filters the bot uses to identify: on the one hand, it makes use of one regular expression to identify the existence of a link (as described in the previous section) in an article's Wikicode; however, it also uses a regular expression for the title, which seeks to identify whether it might belong in the calendar scope. The filter also works by identifying potential calendar-related categories, and another publicly-available `json` file for articles that are considered exceptions to the rule is fetched through MediaWiki API.

Exceptions

Exceptions (as article titles) are fetched through the MediaWiki API each time the bot begins its work. This is processed by the `getExceptions` function:

```
async function getExceptions(): Promise<string[] | null> {
    /.../
    try {
        const JSONList = await
         ↪  getContent('Usuario:NacaruBot/exceptions.json');
        message.succeed('Exceptions retrieved!');
        return JSON.parse(JSONList);
    } catch (error) {
        message.fail(`Error retrieving exceptions: ${error}`);
        return null
    }
}
```

Exceptions are added to the list based on community feedback. Once the list is obtained through the function call described above, the bot applies them as filter. In cases where the API fetch returns an error, execution is stopped:

```
exceptions = await getExceptions();
console.log('Starting search...');
if (exceptions == null) {
    return console.log('Could not load exceptions, stopping execution');
}
```

In the `sanitiseArticle` function, `null` is returned if the article's title corresponds to any of the exceptions on the list. This is carried out through the `some` native function:

```
if (exceptions.some(e => article == e)) {
    return null;
}
```

This ensures the bot will not remove internal dates in articles that are included on the list.

`genArticles` function

The main function in charge of making API requests and deciding whether an article becomes edited by the bot is the `genArticles` function. In order to allow the function to run indefinitely, is is built inside an endless `while` loop.

The function makes call to the MediaWiki API, specifically through the `query` action, in order to obtain 50 pseudorandom articles:

```
let params: QueryParams = {
    action: 'query',
    format: 'json',
    formatversion: "2",
    generator: 'random',
    grnnamespace: '0|104',
    grnlimit: '50',
    prop: 'revisions',
    rvprop: 'content',
    rvslots: 'main',
};
```

The articles are random because the parameter `random` is being used as a generator here. They are not entirely random but *pseudo*-random, however, because of the use of a continue-type parameter. This parameter allows for the saving of the seed information, which is used in the next call inside the loop. After the first call and once the response is obtained, the bot saves the `grncontinue` value in the response as a global variable and reuses it in the next call to the API:

```
grncontinue = result.continue?.grncontinue;
```

The bot the runs the `sanitiseArticle` through each one of the 50 articles from the response. This function runs the three filters (explained in the previous «exceptions», «categories» and «title regex» sections).

```
if (titleRegex.test(article)) {
    return null;
}
if (exceptions!.some(e => article == e)) {
    return null;
}
if (categories.some(e => content.includes(e))) {
    return null;
}
```

If the article passes the three filters, it proceeds to analyse whether it contains an internally-linked date. This is carried out through the use of the `test` native method. If any of the tests turns positive, article content and information on which regexes have tested positive is stored in a dictionary. The article is the pushed into an array (`sanitisedArray`).

`makeEdits` and `editArticle` functions

Once the array has been constructed and if any articles among the 50 obtained have tested positive on the date regexes, the `makeEdits` function will be called, which, in turn, calls the `editArticle` for each one of the articles contained in the array. `editArticle` uses the mwn method `save` to edit the page.

```
async function editArticle(article: string): Promise<void> {
    const message = ora(`Editing: ${article}`).start();
    try {
        await bot.save(
            article,
            replaceText(article),
            'Bot: eliminando enlaces según [[WP:ENLACESFECHAS]]'
        );
        message.succeed(`Success: ${article}`);
    } catch (error) {
        message.fail(`Failure: ${article}. ${error}`);
    }
}
```

It the proceeds to call the `replaceText` function to provide for the next text in the edited article. The function applies the regular expressions discussed in the relevant section through the `replace` method.

Once this is done, the bot displays a *succeed* message or an *error* if it has not been able to perform the action.[3]

After the process, the script cleans up the `sanitisedArray` global variable and calls the `genArticles` function again.

Regular expressions

Regular expression used by the bot to filter or identify dates are publicly available on the project's public repository file. The following sections explain what each of the constructed regular expressions does.

Title regex

The bot makes use of a regular expression to determine whether an article belongs to the calendar scope or not. In order to achieve this, a regular expression was designed. The entire regex is wrapped in a non-capturing group to allow for alternation between multiple complex patterns. The type of articles the regular expression can identify are the following:

- Days of the month, from 1-31, followed by the month itself. It also identifies those that are written using an ordinal
- The months themselves, optionally followed by a year
- Days of the week
- Decades
- Centuries
- «Anexo» pages that are followed by any of the above
- Chronologies
- International or national celebration days and weeks

[3]The most common reason why the bot is unable to perform an edit action is because the page is under total protection.

- Years up to 9999
- Calendars
- Millenia

Regular and pipe regexes

These regular expressions are aimed at identifying the existence of an internally linked date in the string. These two regex are only used after the exceptions, categories and title filters have been passed. The regexes identify the following elements as a date:

- Days of the month, from 1-31, followed by the month itself. It also identifies those that are written using an ordinal
- The months themselves, optionally followed by a year
- Days of the week
- Decades
- Centuries
- Years up to 9999
- Calendars
- Millenia

Template regex

Similarly to the regular and pipe regexes, the template regex only acts after the filters have all passed. It aims at identifying internal links that have been introduced in the article through templates, as explained in the «bot action» section. The regular expression identifies:

- The existence of the `1` parameter in the `{{siglo}}` template.
- The existence of the parameter `link` with a value of `sí` or `si` in the `{{fecha}}` template.
- The presence of a parameter `link` with a value of `"true"` in the `{{julgregfecha}}` template.

Categories

One of the filters that the bot uses to decide whether it should act on a page or not are the categories at the end of the section. Wikipedia articles use a category system to easily tag the scope of each article. Since the *WP:ENLACESFECHAS* rule should not apply in articles that are related to the calendar, the bot avoid acting on a page if it finds any of the following categories in an article:

```
export const categories: string[] = [
    '[[Categoría:Anexos:Tablas anuales',
    "[[Categoría:Calendario",
    "[[Categoría:Celebraciones del día",
    "[[Categoría:Efemérides no oficiales",
    "[[Categoría:Días",
    "[[Categoría:Festividades",
    "{{Ficha de mes",
    "{{solsticio-equinoccio}}",
    "{{desambiguación}}",
    "{{Desambiguación}}"
]
```

Security

The code of the bot is publicly available and released under the GPL 3.0 license, which allows for free use. However, bot accounts can make massive contributions due to the fact that their rate limit is non-existent. This makes them a powerful tool that can be use for malicious purposes in the wrong hands. In order to avoid this, the credentials needed to connect to the bot account through the specific mwn method are obtained through the reading of a `json` file that is not publicly available and that people making use of the code will have to configure appropriately:

```
const bot = await Mwn.init({
    apiUrl: 'https://es.wikipedia.org/w/api.php',

    // Can be skipped if the bot doesn't need to sign in
    username: credentials.username,
    password: credentials.password,

    // Set your user agent (required for WMF wikis, see
    ↪  https://meta.wikimedia.org/wiki/User-Agent_policy):
    userAgent: 'Nacarubot/1.0
↪   JavaScript/:w:es:User:Nacaru/date-link-remover-control-panel.js',

    // Set default parameters to be sent to be included in every API
    ↪  request
    defaultParams: {
        assert: 'user' // ensure we're logged in
    }
});
```

The bot also makes use of a Public Key Infrastructure file (`.pem`) to provide authenticity when making API requests.

Testing

Testing is done through the JavaScript testing framework Jest. Testing is divided in modules and then grouped together into one file.

In order for Jest to work in TypeScript-based projects, it needs to use the Babel compiler.

Thorough testing is done on the regular expressions that the bot uses to analyse article content. In order to do this, the `expect` Jest module is used, together with the `.toBeFalsy()` and '.toBeTruthy() methods.

Testing comprising the same concept subject is carried out through various examples of situations where the regular expression should or should not apply. This is done thorugh the `every()` and `.some()` native JavaScript array methods.

Tests on the general and pipe regular expressions

The main regex works on internal links that take to dates, whereas the pipe regex works on internal links that are piped with a different text.

The following cases are tested, for both regular expressions:

- They should match years with up to 4 digits
- They should match «a.C.» and «d.C.» dates
- They should match years with words attached
- They should match non-breaking spaces in the «a. C.»/«a. D.» cases
- They should match days of the month
- They should not match days of the month that are above 31
- They should match centuries
- They should match decades
- They should match millenia
- They should match millenia if the number has been spelled out
- They should not match decades that do not exist (such as 8000 or 99).

The template regex has more specific tests:

- It should match the last argument `1` in the `{{siglo}}` template.
- It should match the `link` paramter in the `{{julgregfecha}}` template.
- It should not match `{{siglo}}` templates that do not include a date.
- It should not match `{{julgregfecha}}` templates that do not include a date.
- It should not match `{{fecha}}` templates that do not include a date.

Tests are performed upon each compilation of the code, with compilation stopping if any of them gives a negative result.

Future improvements

Due to time restrictions, a series of implementations could not be included in the project. This list summarises them:

- Ideally, the bot could have a real-time web interface. This could be used to provide a publicly available action log or the possibility of controlling the bot from the web. It was planned in the early phases of the project, but it could not be completed due to time restrictions.
- The list of actions carried out by the bot could be increased. This would require making a new request in the Wikipedia bot authorisation board, as new functions must go through community evaluation before their implementation.

Attribution

Images

- Tucán image is a derivative of Wikipedia 20 Tucán by Alehurtado Design, licensed under the Creative Commons Attribution-Share Alike 4.0 International license (link).
- Star image is a derivate of Wikipedia 20 pink star by Delphine Ménard, licensed under the Creative Commons CC0 1.0 Universal Public Domain Dedication (link).
- WP20Symbol image is a derivate of WP20Symbols 2004 aktenzeichen by Wikimedia Deutschland e. V., licensed under the under the Creative Commons Attribution-Share Alike 4.0 International license (link).

References

1. *Global top websites by monthly visits 2023. Statista.* (n.d.). Retrieved March 8, 2024, from https://www.statista.com/statistics/1201880/most-visited-websites-worldwide/
2. *Wikiproyecto LGBT. Dosmanzanas.* (n.d.). Retrieved March 7, 2024, from https://www.dosmanzanas.com/2009/05/wikiproyecto-lgbt.html
3. *What is open?* (n.d.). Retrieved March 12, 2024, from https://okfn.org
4. Molloy, J. C. (2011). The open knowledge foundation: Open data means better science. *PLOS Biology*, *9*(12), e1001195. https://doi.org/10.1371/journal.pbio.1001195
5. Commons, C. (2023, June 29). *Wikipedia moves to CC 4.0 licenses. Creative commons.* https://creativecommons.org/2023/06/29/wikipedia-moves-to-cc-4-0-licenses/
6. *Commons:licensing - wikimedia commons.* (n.d.). Retrieved March 12, 2024, from https://commons.wikimedia.org/wiki/Commons:Licensing
7. Voss, J. (2005). *Measuring wikipedia.* http://eprints.rclis.org/6207/
8. Rosenzweig, R. (2006). Can history be open source? Wikipedia and the future of the past. *Journal of American History*, *93*(1), 117–146. https://doi.org/10.2307/4486062
9. Meseguer-Artola, A., Rodríguez Ardura, I., Ammetller, G., & Rimbau-Gilabert, E. (2020). *Academic impact and perceived value of wikipedia as a primary learning resource in higher education.* https://doi.org/10.3145/epi.2020.may.29
10. Magni, G., & Reynolds, A. (2021). Voter preferences and the political underrepresentation of minority groups: Lesbian, gay, and transgender candidates in advanced democracies. *The Journal of Politics*, *83*(4), 1199–1215. https://doi.org/10.1086/712142
11. McInroy, L. B., & Craig, S. L. (2017). Perspectives of LGBTQ emerging adults on the depiction and impact of LGBTQ media representation. *Journal of Youth Studies*, *20*(1), 32–46. https://doi.org/10.1080/13676261.2016.1184243
12. Snyder, J. (2020). Representation of LGBTQ characters in 2019 young adult literature. *Graduate Research Papers*. https://scholarworks.uni.edu/grp/1531
13. Ribé, M. M., Kaltenbrunner, A., & Keefer, J. M. (2021). Bridging LGBT+ content gaps across wikipedia language editions. *The International Journal of Information, Diversity, & Inclusion*, *5*(4), 90–131. https://www.jstor.org/stable/48641981
14. Kollock, P., & Smith, M. (2002). *Communities in cyberspace.* Routledge.
15. Pullen, C., & Cooper, M. (2010). *LGBT identity and online new media.* Routledge.