REST API

Public REST API for accessing the database.

- API design
- API routes
- API implementation
- API use cases

API design

Principles of the API design:

1. All documented routes should be appended to https://ddd.cjvt.si/api/.
2. All the routes are available as POST calls, even if they do not result in changes in the database, because:
   - some routes will have non-trivial input parameters (structured data, arbitrary strings), which are difficult and clunky to encode as path parameters, and expecting request body parameters in GET calls can be problematic and misleading
   - we can have short clear URLS for all routes, and there are limits on URL length in some contexts.
3. Some routes also have a GET counterpart, which behave the same way as the POST call but do not allow for response body parameters (default values are used instead).
4. All request parameters are provided as JSON request body parameters, except for the object's id, which is used to identify a given object and provided as an obligatory path parameter for certain types of calls (e.g., retrieve).
5. The following HTTP response codes are used:
   - **200**: for most successful requests
   - **201**: for successful get-or-create requests where no matching object was found and a new one was created
   - **400**: an error occurred due to invalid or unexpected request parameters or combinations
   - **401**: authorisation denied (suitable credentials are needed for routes which write to the database)
   - **404**: objects were not found for the value (usually id) provided
   - **501**: the specifications for this route are designed but it has not yet been implemented
6. Each route falls under a particular type of operation identified with a particular verb as the first part of the route. The verbs include:
   - **retrieve**: return data for a given object
   - **search**: return all the objects which match the set of search parameters
   - **export**: return all object ids by minimal filter and with minimal data, using cursor pagination
   - **get-or-create**: get the object matching the parameters provided, creating one if necessary, along with any other missing objects it depends on
   - **update**: update the properties of a given object based on the parameters provided
   - **delete**: delete a given object
   - **attach**: attach the given object to a particular resource, if not yet attached
   - **detach**: detach the given object from a particular resource, if attached
   - **process**: process the input data with an appropriate independent tool (e.g., the CLASSLA NLP library)

7. If the operation verb has a "-batch" suffix, it differs from its non-batch counterpart as follows:
    - users can make 1 API call instead of N API calls for N items
    - the input data should be a list, with each element in the format expected by the non-batch route
    - the output data is a list, with each element corresponding to the element at the same position in the input, where each element has three fields:
        - status: the HTTP response code that would be used if the element was processed in a non-batch call
        - message: a message describing the results of the operation (e.g., whether an object was found or created, or the cause of the warning or error)
        - data: the output data (for successful calls), in the same format as non-batch output
8. Routes which do not change data in the database (retrieve, search, export, process) are publicly available. Routes which may result in changes in the database (get-or-create, update, delete, attach, detach) require authentication credentials.

API routes

The API is being designed and developed, with priority on current needs. Specifications are available in redoc (which is better formatted visually) and swagger (which allows you to try the API via the interface).

Here is a list of the current routes (last update: 31.03.2025). All routes are available with POST, while some of them also have GET or batch POST alternatives (ref). The routes that are not read-only have restricted access.

Route	Read-only	Description
/search/lexical-unit/	yes	search for lexical units based on their properties and parts
/retrieve/lexical-unit/	yes	get a lexical unit's basic data
/get-or-create/lexical-unit/	no	get or create a lexical unit based on properties and components
/export/lexical-unit/	yes	get all lexical units by type
/search/lexeme/	yes	search for lexemes
/retrieve/lexeme/	yes	get a lexeme's data
/get-or-create/lexeme/	no	get or create a lexeme based on defining properties
/export/lexeme/	yes	get all lexemes by category
/retrieve/lexical-unit-lexemes/	yes	get the lexical unit's component lexemes
/search/category/	yes	search for a lexeme's category (part of speech) by string
/search/form/	yes	search for word forms by string
/search/sense/	yes	search for senses
/retrieve/lexical-unit-senses/	yes	get the senses of a lexical unit
/retrieve/lexical-unit-definitions/	yes	get the sense definitions of a lexical unit
/retrieve/lexical-unit-sense-relations/	yes	get the sense relations of a lexical unit's senses
/retrieve/lexical-unit-collocations/	yes	get the collocations of a lexical unit

Route	Read-only	Description
/retrieve/lexical-unit-translations/	yes	get the translations of a lexical unit
/retrieve/lexical-unit-sense-examples/	yes	get corpus examples for the senses of a lexical unit
/export/lexical-unit-sense-examples/	yes	get all corpus examples by lexical unit type
/get-or-create/resource/	no	get or create a dictionary or other resource
/search/resource/	yes	search or list resources available
/attach/lexical-unit/	no	attach a lexical unit to a resource
/detach/lexical-unit/	no	detach a lexical unit from a resource
/search/syntactic-structure/	yes	get the XML definitions of syntactic structures
/process/string-to-tokens/	yes	parse a Slovene string to get a list of tokens

API implementation

The public API is being implemented using the Django REST Framework and APIViews in particular. It is part of the Python codebase, Django project and Git repository that is used to manage the database in general. We are striving to keep the business logic and API route definitions in separate modules, so that different APIs (e.g., editor API, internal API) can use the same utils module.

Most of the logic and processing of the API is internal. However, there are a few aspects that rely on other tools, such as Slovene string parsing and fetching of corpus examples.

API use cases

In addition to providing general public access to the database, the REST API can also be used to integrate data and services with external organisations in a coordinated, structured and systematic way. Two current examples of this are integration with terminology portals and speech technologies, both of which use a mix of public (read-only) and restricted (read-write) routes of the API.

Terminology Portal

One of the main parts of the Development of Slovene in a Digital Environment is a terminology portal that will feature various terminological resources and offer an openly accessible tool for term extraction from specialized corpora, as well as the server infrastructure needed to create new terminological resources. The main components of the portal include a search engine for all integrated resources and a terminology resource editor, and the resources are designed to be easily integrated with other language tools and services, including the Digital Dictionary Database.

As such, the portal uses API routes to register its dictionaries in the database, search and create terms, attach/detach them to/from the dictionaries, and fetch their forms and statuses. The API supports this as follows (see the route links for full examples):

- Register the dictionary as a resource in the database using /get-or-create/resource/, providing a code name for the dictionary (e.g., `"slm"`) as input. The API will return the resource's ID (e.g., `87`), first creating it if it does not yet exist.
- Get IDs of terms in the database, creating them if necessary, using /get-or-create/lexical-unit/. Input can be either the term's raw string (e.g., `"okrogla miza"`), or its pre-analysed sequence of tokens, with each token represented with corpus-style data (e.g., `[{"lemma":"okrogel", "msd":"Ppnzei", "form":"okrogel"}, {"lemma":"miza", "msd":"Sozei", "form":"miza"}]`). If a raw string is provided, the API uses a standard tool to get a sequence of tokens itself. Either way, it then checks if a matching lexical unit exists in the database, creates one if necessary, and returns its basic data, including its ID (e.g., `54321`). If many terms need processing, this can be done by using the /get-or-create-batch/lexical-unit/ call instead and providing a list of inputs.
- Get the word parts and their forms with statuses for a specific term, by using /retrieve/lexical-unit-lexemes/. Input would be the term's ID (e.g., `54321`) and specifying that form statuses and all form types are also requested (e.g., `"extra-data":["status-form-types", "forms-orthography", "forms-accentuation", "forms-pronunciation"]`). The output then a list, with each element is one of the term's word constituents, represented with both its basic data (such as id, lemma, part of speech, basic word-level features) and the extra

requested data (the list of all the forms of all the types for the word and aggregated statuses for each type).

- Search for term candidates in the datases, using /search/lexical-unit/. This has similarities to /get-or-create/lexical-unit but differs in a few key ways. First, it does not create a lexical unit if no match exists, and thus does not require authentification, so less central components of the portal can also make use of it. Second, it does not require complete data in the input (e.g., perhaps only one or two of lemma/msd/form are specified for one or more of the term's components), making the search more flexible and potentially returning multiple matches (e.g., {"lemma":"klop"}).
- Attach the lexical unit to a resource, using /attach/lexical-unit/. The input would be the IDs of the term as a lexical unit (e.g., `54321`) and dictionary as a resource (e.g., `87`). This would then connect the two in the database.
- Detach the lexical unit from a resource, using /detach/lexical-unit/(https://blisk.ijs.si/api/redoc/#tag/detach). The input would be the IDs of the term as a lexical unit (e.g., `54321`) and dictionary as a resource (e.g., `87`). This would then remove the term from the resource in the database, without deleting the term from the database in general.

Speech technologies

The project Tolmač (Eng. Interpreter) is focused on developing of a system for automatically translating lectures from Slovene to other languages, coordinated at the Faculty of Computer and Information Science at the University of Ljubljana, in close collaboration with the Centre for Language Resources and Technologies. The results of the project will be important for a wide range of people: real-time translations will make it easier for foreign students to follow lectures in Slovene, automatic subtitles will help people will hearing loss, and lecture excerpts and recordings will be accessible at a dedicated website. The speech technologies underlying the system rely on search and retrieval of both orthographic and pronunciation word forms of Slovene words.

To that end, the system can use API routes to preprocess text, search for different kinds of word forms, retrieve the forms of word and create new words along with their forms. The API supports this as follows:

- Parse a piece of Slovene text to get a sequence of tokens using /process-string-to-tokens.

  The API runs the standard CLASSLA parser with default parameters and returns a list of tokens in CoNLL-U format, with each token including a lemma (e.g., `"miza"`), MSD (`"Sozmm"`) and form (e.g., `"mizah"`). Thus this API call does not interact with the database, but serves as a handy wrapper for CLASSLA, so the user does not need to install it themselves.
- Search for a word form in the database using /search/form/, by providing a type (e.g., `"orthography"`) and a string (e.g., `"mizah"`). The output is a list of matching forms, along with basic associated data such as lexeme ID (e.g., `123`), lemma (e.g., `"miza"`) and JOS-

system MSD (e.g., `"Sozdm"`). Associated pronunciations for all matching forms are included if requested (e.g., `"extra-data":["forms-pronunciation"]`).

- Get all the forms of a given lexeme using /retrieve/lexeme/, using the lexeme's ID as input. To get all the orthography and pronunciation forms of the lexeme, specify in the input (e.g., `"extra-data":["forms-orthography", "forms-pronunciation"]`).
- Create (if it does not yet exist) a lexeme in the database using /get-or-create/lexeme/. Input consists of a lemma (e.g., `"miza"`) and MSD (e.g., `"Sozmm"`). The MSD can be any appropriate MSD for the lexeme, not necessarily the MSD of the lemma itself, since only the lexeme-level parts of the lemma (e.g., `"Soz"`) will be considered. The API calls the Inflector tool, which generates full paradigms of different kinds of forms (orthography, accentuation, pronunciation), and then saves the new lexeme with its forms in the database. However, if a lexeme already exists in the database which matches the database, no duplicate lexeme (along with forms) is created and the existing lexeme is returned.

Povejmo

One of the goals of the project is to enable a large language model to learn Slovene grammar. The training dataset will be largely based on data obtained from the Digital Dictionary Database, and the goal is to extract and incorporate as much data as possible, so that the model can also learn specific nuances.

To that end, the API's various `/export` and `/retrieve-batch` routes are particularly relevant, as they enable first finding all the relevant (single-word or multi-word) unit IDs and then different kinds of data for all the words. For example:

- Find the IDs of the first chunk of single-word lexical units in the database, using /export/lexical-units/ with `"type"="single_lexeme_unit"`.
- Accumulate all the IDs by repeatedly following the `"next"` links until next=null.
- Break the lexical unit IDs into batches that can be used in /retrieve-batch calls.
- For each batch of lexical unit IDs, retrieve associated data using:
    - /retrieve-batch/lexical-unit-lexemes/ with `"extra-data"=["forms-orthography"]` to get each lexical unit's lexeme along with all of its morphological forms.
    - /retrieve-batch/lexical-unit-definitions/ to get all definitions of different kinds for the lexical units' senses.
    - /retrieve-batch/lexical-unit-sense-relations/ with `"type"="synonym"` to get all the synonyms for the lexical units' senses.
    - /retrieve-batch/lexical-unit-collocations/ to get collocations.
- To get corpus examples, use the separate /export API calls (and following pagination, as above) rather than `/retrieve-batch/`, for performance reasons:
    - /export/lexical-unit-sense-examples/ with `"type"="single_lexeme_unit"` for the single-word lexical units.

- /export/lexical-unit-sense-examples/ with `"type"="collocation"` for the collocations.