ACADEMIC SOFTWARE PROJECT

Dictionary Assistant

Designing an API-powered English-Bulgarian lexical reference tool

What this case study demonstrates: Java application design, external API integration, English-language data processing, console UX, file operations, and critical technical evaluation.

A university project combining language technology and practical Java engineering. The application accepts an English word and assembles definitions, part of speech, examples, synonyms, antonyms, pronunciation, and a Bulgarian translation in a single structured output.

Project contributors: Aylin Kadirova and Elena Piperkova, as credited in the original academic presentation. Format: Java console application. Context: university IT practicum.

Portfolio note: This case study is an English-language adaptation of the supplied academic presentation and documentation. Any synthetic console outputs or illustrative metrics are explicitly labelled; they are not represented as measured results.

01 / CONTEXT

Executive overview

The problem

Looking up an unfamiliar English word commonly involves several separate actions: finding a definition, checking pronunciation, verifying its part of speech, exploring synonyms, and translating the term into Bulgarian. Switching between resources interrupts reading and slows down self-directed vocabulary study.

The proposed solution

Dictionary Assistant centralises this workflow behind one command-line search. It queries external dictionary and thesaurus services, requests a Bulgarian translation, maps responses into a WordEntry object, and prints the information in a consistent format.

Audience	Primary use
Students and independent learners	Review vocabulary, meanings and example usage
Teachers and linguistics students	Demonstrate lexical structure and API-based language resources
Translators	Perform an initial context-sensitive lexical lookup
Junior Java developers	Explore HTTP, parsing, object modelling, files and CLI input

Scope boundary: This is a working educational console-tool design, not a production dictionary service, CEFR-certified course, or user-tested learning platform.

02 / CAPABILITIES

Product experience

One query, multiple lexical fields

Information returned	Why it matters for language learning
English headword + part of speech	Distinguishes meaning and grammatical function
Definition + usage example	Supplies meaning and contextual usage
Pronunciation representation	Offers a cue for spoken-language recognition
Synonyms + antonyms	Supports semantic relationships and active vocabulary
Bulgarian translation	Connects English input to the learner’s first language

Beyond a single word lookup

- history: recalls recent searches (up to ten entries, with duplicate handling).
- export: saves the most recently found entry to a plain-text file.
- batch <file>: processes a newline-delimited word list sequentially.
- help / exit: offers command discoverability and predictable session management.

03 / SYSTEM FLOW

User journey and workflow

01 / INPUT

User enters a word or a recognised command in the console.

02 / VALIDATION

The application checks for empty, overly long or unsupported input.

03 / LOOKUP

Dictionary and thesaurus GET requests retrieve relevant English-language data.

04 / TRANSLATION

A translation service returns a Bulgarian-language equivalent.

05 / NORMALISATION

The parser constructs a WordEntry record from external JSON payloads.

06 / OUTPUT

The console displays the entry and updates the search history.

04/ ENGINEERING

Technical architecture

The documented Eclipse project follows a simple responsibility-based structure, with separate model, service and utility packages. This is a lightweight layered design, not a full MVC implementation.

Component	Responsibility
Main.java	Command loop; orchestration and input validation
model / WordEntry	Lexical data: meaning, example, translation and semantic relationships
service / MerriamWebsterService	Dictionary and thesaurus HTTP requests and status handling
service / TranslationService	Translation request and response extraction
service / JsonParserService	Maps raw response strings to the WordEntry model
util / ConsolePrinter	Readable command-line formatting and feedback
util / FileWordReader	Validates and loads batch word inputs
util / SearchHistory	Recent-word persistence and text export

The original technical documentation identifies Java HttpClient, an Eclipse project layout, JSON processing, two Merriam-Webster endpoints, and a separate translation API. It describes the primary workspace project as IT_Practice_25_26.

05 / DATA SOURCES

Integrating external services

What each service contributes

- Merriam-Webster Collegiate Dictionary API: definition, part of speech, pronunciation and example data when provided.
- Merriam-Webster Thesaurus API: synonym and antonym relationships.
- Translation API: Bulgarian translation of the queried English term.

Request and error-handling design

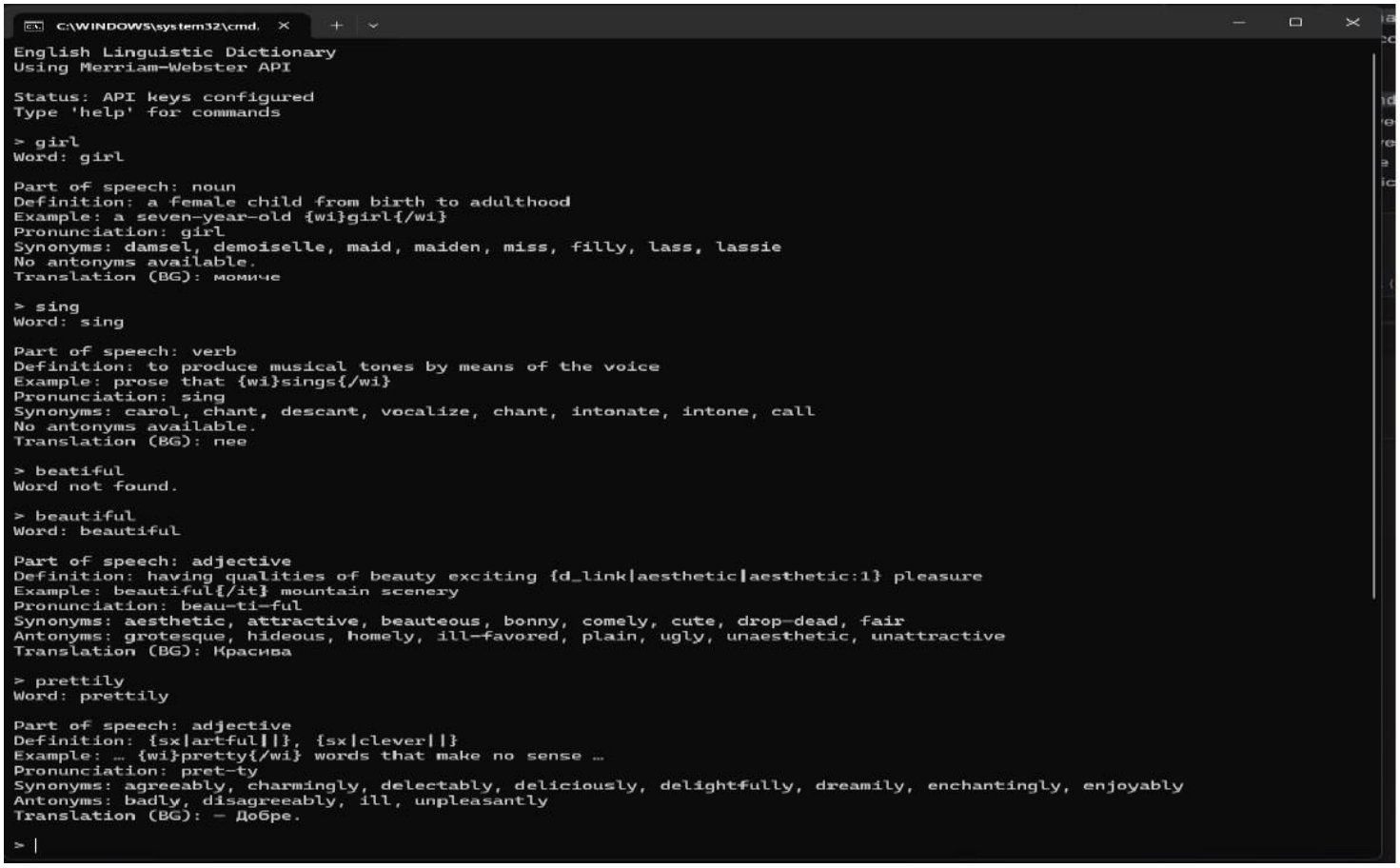
The dictionary service normalises search text, URL-encodes the query and makes HTTP GET requests. The documented implementation distinguishes successful responses from HTTP 404, 401 and other errors. The translation endpoint uses POST with a JSON payload and parses the translated_text value.

Engineering lesson: A feature that depends on three remote requests must account for partial failures, latency, service limits, missing fields and changing response formats. The documented application handles a subset of these concerns; production hardening remains a follow-up task.

06 / PRESENTATION SCREENSHOT

Actual application evidence

The following is an authentic console screenshot reproduced from the supplied university presentation. It is source evidence, not a newly executed program session.



Visible sample searches include girl, sing, beautiful and prettily. The screenshot also reveals an important quality issue: some example strings retain Merriam-Webster formatting tokens such as {wi3} and certain definitions contain encoded markup. This is a useful finding for subsequent parser improvements.

07 / SYNTHETIC DEMO

Illustrative console output

The following formatted output is fictional example data designed for portfolio illustration. It is not a transcript of an actual lookup, nor an authenticated API response.

```
> lookup journey

Word: journey

Part of speech: noun

Definition: An act of travelling from one place to another.

Example: They planned a long journey by train.

Pronunciation: /ˈdʒɜːr.ni/

Synonyms: trip, voyage, expedition

Antonyms: not available

Translation (BG): пътуване

> export

Saved: journey_definition.txt

> history

1. journey
```

All values above are mock data. Pronunciation conventions and actual API outputs can differ; do not use this synthetic entry as a benchmark or verified linguistic record.

08 / CONSOLE DESIGN

Handling user commands

Command	Expected behaviour
<word>	Query lexical information and display a formatted record
batch <file>	Read and process a text list of words, one after another
history	List recently searched words, deduplicated
export	Write the latest successful entry to a text file
help	Display supported commands
exit	Close the session

Edge cases to design for

- Empty input, digits, unusual punctuation or abnormally long search text.
- Invalid API credentials, 404 responses, unavailable services or an offline machine.
- Partial entries without examples, phonetic transcriptions or antonyms.
- Missing or invalid batch files, duplicate lines and non-UTF-8 input.
- Exports requested before any successful word lookup.

09 / API VS. OFFLINE

Architecture choices

The accompanying university paper compares dynamic API-based lookups with a hypothetical local-only dictionary. The offline alternative was an architectural comparison, not a second implemented application.

Dimension	API-supported lookup	Local-only dictionary
Data coverage	Broad provider-managed catalogue	Limited to locally stored entries
Updates	External provider may update content	Manual local curation required
Connectivity	Requires service access	Works without an internet connection
Availability	Exposed to network and API outages	Dependent on local data integrity
Maintenance	Manage API changes and credentials	Maintain content catalogue locally
Latency	Includes request overhead	Potentially faster local lookup

Recommended next iteration: hybrid access

The most useful evolution is an API-first model with a local cache. The application checks local entries before making a remote request, stores successful lookups, and remains useful for words already cached during temporary connectivity loss. This was proposed in the original comparison document and should be described as future work.

10 / TECHNICAL REVIEW

Quality, security and reliability

What the documented prototype does well

- Clear separation of console presentation, lexical model, API adapters and file utilities.
- Use of the Java standard HTTP client instead of adding a dependency for simple requests.
- Useful learner features beyond basic lookup: history, batch mode and export.
- Input validation and basic HTTP status handling.

Technical debt identified in the original documentation

- Manual JSON string extraction: vulnerable to nesting, escaping and response format changes; use Jackson or Gson.
- Credentials in source: move secrets into environment variables or protected configuration; rotate keys before public release.
- Build reproducibility: migrate Eclipse-only configuration to Maven or Gradle.
- Incomplete linguistic cleaning: normalise Merriam-Webster formatting tags before output.
- Resilience: add request timeouts, bounded retries, rate limiting and test doubles.

11 / HYPOTHETICAL TEST PLAN

Demonstration scenarios

This is a proposed QA matrix, not a record of tests executed against a running binary. The expected outcomes illustrate how the prototype could be verified and extended.

Scenario	Expected acceptance criterion
Known word	Populated headword, definition and available related fields
Unknown word	Helpful not-found message without app termination
Invalid word	Rejected before remote request
Translation service down	English entry remains usable; warning displayed
Malformed dictionary JSON	Parsing fails safely without unhandled exception
Repeated history entry	Recent-word list contains one latest instance
Batch list, mixed inputs	Valid words processed; errors reported per item
Export before search	Informative no-entry-to-export message

No test coverage percentage, pass rate, latency improvement, user satisfaction score or production usage claim is asserted in this case study: these were not evidenced by the uploaded materials.

12 / DEMONSTRATION ONLY

Synthetic product evaluation

To show how a future evaluation dashboard could look, the figures below are invented illustrative values, not measured results. They describe a hypothetical controlled benchmark and must never be cited as genuine performance evidence.

Mock benchmark measure	Fictional outcome	Interpretation
Sample lookup attempts	40	Illustrative dataset size
Lookup records displayed	35 / 40	Hypothetical 87.5% result-display rate
Cases requiring content cleanup	6 / 40	Hypothetical markup/formatting defects
Commands with expected behaviour	18 / 20	Hypothetical command-level checks
Median response time	0.82 s	Fictional end-to-end latency

How real numbers would be collected

Freeze a dataset with known words, missing words and malformed responses; record program version and API conditions; replay the same corpus; capture result validity, response time, failures and export correctness; then independently inspect definitions, translations, and linguistic formatting.

13 / NEXT ITERATIONS

Future roadmap

Priority	Engineering step	Product benefit
P0	Move secrets out of source control and rotate exposed keys	Safe portfolio and repository sharing
P0	Replace string-based JSON parsing with structured mapping	Resilient lexical data extraction
P1	Add automated tests and stubbed API fixtures	Reliable regression checks without live API dependence
P1	Introduce local caching and persistent favourites	Faster repeat search and basic offline access
P2	Add JavaFX or web user interface	Greater accessibility for nontechnical learners
P2	Support text/PDF vocabulary extraction	Turn reading material into lexical study lists
P3	Export flashcards and study sets	Bridge vocabulary lookup and active recall

The technical notes mention separate workspace experiments for JavaFX, jsoup and Apache PDFBox. These are research and environment checks, not proof that graphical UI, web scraping or PDF vocabulary extraction are already integrated in the main application.

14 / CONCLUSION

Reflection and portfolio value

From linguistic concepts to practical code

Dictionary Assistant translates familiar language-learning needs into executable software behaviours: a lexical record becomes a Java data model; reference services become HTTP integrations; vocabulary revision becomes history and export; and exceptions become user-facing fallback conditions.

What this project demonstrates

The strongest evidence is the breadth of the implemented and documented learning exercise: API use, object-oriented decomposition, file input/output, CLI design and English–Bulgarian information presentation. Its constraints are equally informative. The next engineering milestone is not another feature; it is correctness, secret management, deterministic tests and resilient data parsing.

Disclosure: English case-study narrative created from the supplied materials. Genuine presentation screenshot reproduced for context. Separate console and metric examples are fictional and explicitly labelled. No connection to Merriam-Webster beyond the described API integration is implied.

Source materials reviewed

- Java Dictionary App - university presentation (12 slides).
- Technical Documentation: Dictionary Assistant - university report (15 pages).
- API Advantages and the Dictionary Assistant With/Without API - comparison paper (10 pages).

Prepared for portfolio presentation. API endpoints, exact source behaviour and runtime results should be verified against the original codebase before external technical claims are expanded.