



TypeTaxonScript: sugarifying and enhancing data structures in biological systematics and biodiversity research

Lucas Sá Barreto Jordão ^{1*}, Marli Pires Morim², José Fernando A. Baumgratz², Marcelo Fragomeni Simon ³, André L. C. Eppinghaus¹ and Vicente A. Calfo¹

¹Centro Nacional de Conservação da Flora—CNCFlora, Instituto de Pesquisas Jardim Botânico do Rio de Janeiro, Rio de Janeiro, 22460-030, Brazil

²Diretoria de Pesquisa Científica—DIPEQ, Instituto de Pesquisas Jardim Botânico do Rio de Janeiro, Rio de Janeiro, 22460-030, Brazil

³Embrapa Recursos Genéticos e Biotecnologia, Parque Estação Biológica—PqEB, Brasília, 70770-901, Brazil

*Correspondence address. Centro Nacional de Conservação da Flora—CNCFlora, Instituto de Pesquisas Jardim Botânico do Rio de Janeiro, Rio de Janeiro, 22460-030, Brazil. E-mail: tucarj@gmail.com

Abstract

Object-oriented programming (OOP) embodies a software development paradigm grounded in representing real-world entities as objects, facilitating a more efficient and structured modelling approach. In this article, we explore the synergy between OOP principles and the TypeScript (TS) programming language to create a JSON-formatted database designed for storing arrays of biological features. This fusion of technologies fosters a controlled and modular code script, streamlining the integration, manipulation, expansion, and analysis of biological data, all while enhancing syntax for improved human readability, such as through the use of dot notation. We advocate for biologists to embrace Git technology, akin to the practices of programmers and coders, for initiating versioned and collaborative projects. Leveraging the widely accessible and acclaimed IDE, Visual Studio Code, provides an additional advantage. Not only does it support running a Node.js environment, which is essential for running TS, but it also efficiently manages GitHub versioning. We provide a use case involving taxonomic data structure, focusing on angiosperm legume plants. This method is characterized by its simplicity, as the tools employed are both fully accessible and free of charge, and it is widely adopted by communities of professional programmers. Moreover, we are dedicated to facilitating practical implementation and comprehension through a comprehensive tutorial, a readily available pre-built database at GitHub, and a new package at npm.

Keywords: JavaScript; TypeScript; JSON; Mimosa; Node.js; taxonomy; morphology; Leguminosae; Fabaceae; Visual Studio Code; plant

Introduction

The endeavour to describe and catalogue organisms spans generations, contributing significantly to the foundations of biological knowledge and classification. Rooted in historical scientific literature, the practice of representing organisms through textual descriptions acts as a bridge connecting past and present scientific communities [1, 2]. As the digital age dawns, traditional methods merge with contemporary technology [2, 3].

In the present day, taxonomists and systematists often resort to familiar text editors, like Microsoft (MS) Word, to meticulously craft their descriptions. While some practitioners venture into spreadsheets for structured data [4], rapid technological advancements unveil new avenues for documentation and data organization.

Amidst this evolving landscape, untapped potential arises through cutting-edge methodologies. While digital tools have significantly streamlined numerous research tasks, a notable gap persists between these contemporary solutions and their widespread acceptance within the scientific community. In this context, our exploration delves into the symbiotic relationship between object-oriented programming (OOP) and document-oriented databases (DOD). Through this lens, we foresee a

paradigm shift propelling biodiversity research into an era of efficiency, collaboration, and innovation.

TypeScript (TS), an extension of JavaScript (JS), is a robust choice for intricate and organized systems [5]. Combining OOP principles with TS creates a powerful development ecosystem, facilitating the building of a JS Object Notation (JSON) database. Moreover, it permits to incorporate multiple layers of data validation to establish a highly reliable database.

The JSON format emerges as crucial within DOD, standing out for data structuring [6]. Diverging from spreadsheets, JSON's versatility and hierarchy accommodate varied data types, ideal for housing diverse biological data and annotations. This aligns with complex domains like systematics, chemistry, ecology, reproduction, genomics, and proteomics, often better represented through nested hierarchies.

In parallel, another approach for biological data management involves ontologies like Gene Ontology [7, 8] and Plant Ontology [9, 10]. These structured vocabularies connect biological concepts intricately. Yet, not only their complexity, but also their representation demands specialized expertise, making JSON simplicity's appealing to a wider community.

Received: 28 December 2023. **Revised:** 19 February 2024. **Editorial decision:** 26 February 2024. **Accepted:** 12 March 2024

© The Author(s) 2024. Published by Oxford University Press.

This is an Open Access article distributed under the terms of the Creative Commons Attribution-NonCommercial License (<https://creativecommons.org/licenses/by-nc/4.0/>), which permits non-commercial re-use, distribution, and reproduction in any medium, provided the original work is properly cited. For commercial re-use, please contact journals.permissions@oup.com

JSON shines in integrating biodiversity databases, offering an accessible alternative. While ontologies require expertise and are reliant on specific software, JSON directly empowers biologists as it lies at the core of web evolution, enabling the democratization of data-driven research. Furthermore, embracing GitHub technology fosters collaboration by enabling the seamless sharing of code, data, and insights [11]. Concurrently, Visual Studio Code (VS Code) adeptly manages GitHub versioning. This combined toolset significantly supports researchers in diverse fields, facilitating multidisciplinary endeavours aimed at advancing our comprehension of biodiversity and ecological systems.

As capabilities in biodiversity research continue to expand, the role of effective data integration mechanisms and collaborative tools becomes increasingly pivotal. In this context, the utilization of JSON to represent data structures in biodiversity databases, alongside code versioning, not only bridges the gap between complex data structures and the broader scientific community but also propels collaborative solutions in the field of biodiversity research.

Here, we present TypeTaxonScript (TTS), a JS/TS package designed to enrich taxonomic descriptions and advance systematics, while also offering a versatile data structure capable of seamlessly accommodating ecological, reproductive, and other pertinent information of biological entities. We advocate starting the use of text and spreadsheet editors, such as MS Word and Excel, for taxonomic descriptions, due to their lack of robust validation processes during data entry. Instead, we aim to establish a resilient, user-friendly database ensuring data integrity, stability, and compliance with standardized scientific practices in textual taxonomic descriptions, as well as character and interaction matrices.

Background

JS and Node.js environment

JS [12] serves as the foundation for a wide range of modern programming endeavours, including the management of biological data [13–15]. Its versatility and widespread adoption have catalysed the development of tools and platforms that harness its capabilities.

Node.js, a runtime environment built on Chrome's V8 JS engine, extends the potential of JS beyond the confines of web browsers [16]. It enables the execution of JS code outside of browsers, facilitating server-side scripting. This is particularly advantageous for tasks involving data processing, handling API requests, and managing databases [15]. Moreover, Node.js offers access to a wide array of libraries and packages, expediting the development of databases while enhancing its overall functionality.

JS and Node.js are powerful tools in the field of biological data management. Their capabilities contribute to the development of efficient, dynamic, and scalable databases, facilitating advancements in biodiversity research.

OOP and TS

In the ever-evolving realm of biological data management, the fusion of OOP principles with TS, a powerful programming language extension, marks a significant leap forward. OOP, a paradigm in software development, revolves around representing real-world entities as objects, each imbued with distinct attributes and methods [17, 18]. At its essence, OOP depends on the foundation of classes and instances, where each object represents an instance of a class. This paradigm promotes code reusability, modularity, and streamlined maintenance, offering

indispensable tools for navigating the intricate landscape of biological data. The integration of OOP principles into information systems for biological data has been leveraged to combine diverse data sources and interactions [18–21].

TS, as an extension of JS closely tied to OOP, elevates the language's capabilities by introducing essential features like static typing and interfaces [5]. This upgrade makes TS a powerful tool for building complex, robust, and well-organized systems. A noteworthy aspect of TS is its ability to capture errors at compile-time, in contrast to the runtime error detection found in traditional JS [5]. This attribute enhances the identification and resolution of potential issues within the code prior to execution. In the construction and management of biological databases, this early error detection ensures data integrity and improves the overall dependability of the database. This becomes especially crucial when grappling with intricate relationships and complex hierarchies intrinsic to biological data.

Furthermore, within this proposed framework, methods play a pivotal role. These methods, which represent behaviours associated to classes, act as a layer of validation during runtime. They enforce conditions and rules, guaranteeing the accurate and consistent input of data. For example, when a particular structure's absence is declared, such as a plant trichome type, assigning a size to that character becomes illogical. Methods function as conditional safeguards, preventing erroneous or incompatible data from infiltrating the database. As a result, data quality and reliability are significantly enhanced.

The amalgamation of OOP principles with TS's capabilities spawns a potent toolkit for structuring and managing biological data. This dynamic fusion facilitates the establishment of an orderly and dependable ecosystem, ensuring the precision and integrity of biodiversity information.

In a similar vein, TS introduces the concept of syntactic sugar, heightening code readability and expressiveness. This feature incorporates elements that streamline code comprehension without altering its core behaviour. For instance, consider the contrast between dot notation and JSON-like indentation. Dot notation is significantly more intuitive and easier to understand. This refinement in syntax enhances human readability, making it accessible to both experienced developers and researchers, as well as those with limited programming backgrounds. The integration of syntactic sugar harmonizes seamlessly with the overarching framework objectives.

Another compelling instance of the syntactic sugar principle lies in the adoption of linguistic conventions from the domain of biological sciences. Instead of relying solely on primitive types like `true` or `false`, we advocate for the use of contextually appropriate terms like `yes` or `no`, `present` or `absent`. This strategic approach resonates with the syntax employed in descriptions of comparative analyses within the biological sciences. The deep alignment between code semantics and domain-specific concepts adds an additional layer of cohesion to the framework's objectives, nurturing a user-oriented and friendly environment meticulously designed to propel advancements in biodiversity research.

Document-oriented database and JSON

The paradigm shift initiated by DOD and the adoption of the JSON format has had a profound impact on the landscape of biological data management. In contrast to traditional relational databases, DOD store data as self-contained documents, embracing complexity while retaining flexibility [18, 22–27]. In a non-relational DOD, each row in a spreadsheet represents a

document in the database, while each column corresponds to a property of these documents.

At the core of this transformation lies the JSON format, a cornerstone of DOD, offering a versatile and hierarchical structure capable of accommodating diverse data types within a schema-free environment. This schema flexibility allows for the storage of semi-structured data without the constraints of predefined tables, as seen in traditional relational databases [27]. This capability enables capturing nuances that rigid frameworks might overlook. The inherent compatibility between the JSON format and the intricate characteristics of biological information leads to a more comprehensive and precise representation of biological features within the JSON's data structure. This starkly contrasts with the challenges posed by relational databases in capturing complex relationships without resorting to convoluted structures and joins.

While relational databases have found widespread use in structuring data, even within the field of biology, they lack the inherent readability that human thinking naturally craves. The shift from two-dimensional spreadsheet thinking to embracing the hierarchical notation of JSON is clearly advantageous, as it allows for a more effective handling of complexity, leading to the development of a more readable and manageable data structure.

A fundamental characteristic of a DOD is its structural foundation, comprised of an array of objects, each of which is treated as a document. This array serves as the database and requires robust querying capabilities. Each individual object within this array is assigned a specific index. When working with DOD, our primary objective is to navigate through all levels of a JSON structure. This involves not only identifying the index where our query is situated but also tracing the JSON path that leads to this discovery. This dual process of information retrieval offers us a comprehensive methodology for precisely pinpointing and accessing the desired data. Consequently, within a DOD, the document index within the database and the associated JSON path emerges as the two critical tools for effective navigation and data retrieval.

Within JSON, an object comprises a key-value pair. The key serves as a label for a JSON object property. It uniquely identifies a specific piece of data or information stored in the object. Each key is associated with a value, which represents the actual information or data assigned to that key. Values can encompass various data types, including strings, numbers, booleans, arrays, nested JSON objects, or null. This key-value structure in JSON allows for the organization, representation, and access of data in a structured manner, ensuring clarity and ease of retrieval when working with JSON objects.

Equally significant is JSON's contribution to simplifying data querying. By employing JSON notation, researchers can efficiently retrieve data, harnessing its inherent structure for effective information organization and retrieval. Prominent databases like MongoDB (<https://www.mongodb.com/>) leverage the capabilities of JSON, facilitating efficient querying and manipulation, but other non-opinionated tools can also be employed as query languages. These resources empower researchers by offering a data structure that facilitates a highly integrative approach and provides a more intuitive means of retrieving information.

Documentation and TSDoc

Documentation plays a vital role in storing information about objects, highlighting their attributes and values. This is particularly crucial when constructing a robust database with efficient metadata organization, as emphasized by Spinellis [28] and Rai et al. [29],

especially in the context of biological data, as highlighted by Warren et al. [30].

Creating detailed documentation establishes a comprehensive reference guide for all objects by accurately describing each class and its attributes, providing a clear overview of their fundamental characteristics.

Here, TSDoc (<https://tsdoc.org/>) functions as a valuable tool, introducing standardized annotations that clarify attribute details, including type and purpose. These annotations establish a consistent framework, helping contributors understand the purpose and usage of each element.

Solid documentation is essential for developing a sturdy database. It serves as the foundation for effective data organization, ensuring accurate inputs and maintaining database integrity. Furthermore, detailed documentation facilitates team collaboration by offering easy access to object information, reducing errors, and enhancing data accuracy.

Addressing documentation within metadata and database contexts constructs a framework that supports a comprehensive understanding of objects, attributes, and methods. Documentation not only describes but also organizes and clarifies stored values. Incorporating TSDoc annotations enhances this clarity by standardizing descriptions and improving accessibility.

Integrated development environment and Visual Studio Code

An Integrated Development Environment (IDE) stands at the heart of modern software development, offering a comprehensive toolkit to streamline coding, debugging, and collaborative efforts. Notably, VS Code (<https://code.visualstudio.com/>), a widely acclaimed IDE renowned for its adaptability and efficacy, exemplifies this role. Developed by MS, VS Code caters to a diverse range of programming languages and has become a go-to choice for developers and researchers alike.

The core strength of VS Code lies in its user-friendly interface, accommodating developers from various backgrounds. Yet, its true power emerges from an expansive extension marketplace and robust capabilities, which can be customized to meet the demands of diverse coding tasks, spanning TS projects and beyond.

Within our specific project context, VS Code becomes a pivotal asset for manipulating and managing JSON-formatted biological data. Seamlessly integrating TS in Node.js, this IDE ensures the maintainability, adaptability, and responsiveness of the underlying codebase. This proves especially crucial in meeting the evolving demands of biodiversity research. However, it is worth noting that other IDEs are also capable of performing the same tasks.

An added advantage of VS Code is its built-in version control functionality, hinging on Git and GitHub. This integration empowers collaborative development by enabling effortless sharing, reviewing, and tracking of code modifications. Beyond just expediting the development process, these features foster transparency, accountability, and effective teamwork within the realm of biological data management.

By embarking on an exploration of the integration of OOP, TS, and JSON databases within VS Code environment, we unlock novel possibilities to improve the processing, analysis, and utilization of biological data.

We prioritize the simplicity and effectiveness of a programming environment's functionality, intending to utilize it as a platform for constructing a controlled vocabulary database with comprehensive documentation. We have observed that this development environment is equally effective for creating biological databases, rather than attempting to create a new opinionated software. By

proposing this framework, our goal is to provide as few opinions as possible in building a robust biological database.

Git versioning and GitHub

The strategic utilization of version control significantly contributes to maintaining a coherent and collaborative coding environment. GitHub, a widely utilized platform for version control and collaborative development, seamlessly integrates with VS Code, thereby enhancing efficiency and transparency within the development process [31–33]. However, alternative Git versioning platforms are also available for use.

By capitalizing on GitHub's versioning capabilities embedded within VS Code, researchers, biologists, and systematists collaborate on shared projects with ease, effectively tracking changes and managing contributions. The integration guarantees timely synchronization between local repositories and remote repositories hosted on GitHub, ensuring that all team members have access to the most current code.

GitHub's versioning features, encompassing branching, pull requests, and merging, expedite effective collaboration by furnishing clear avenues for reviewing and incorporating code alterations. This structured approach to development workflow mitigates the potential for errors, ensuring thorough testing and approval of code changes before they are integrated into the primary codebase.

Branching plays a pivotal role in the development of codebases for biological data representation. It allows developers to work on distinct features or aspects of biological character representation independently, preventing conflicts and ensuring code integrity. By creating branches, researchers can experiment with different approaches or modifications, fostering innovation while maintaining the stability of the main codebase. This practice facilitates efficient collaboration, as team members can concurrently work on diverse aspects of biological representation.

Pull requests, an essential aspect of collaborative software development within Git versioning, serve as a mechanism for proposing and discussing changes to a repository's codebase. When a developer or contributor wants to suggest modifications or additions, they create a pull request, allowing others to review the proposed changes. This process promotes transparency, peer review, and collaboration within the development community. Pull requests also facilitate meaningful discussions and help maintain the quality and integrity of the codebase, making them an integral part of the open-source and collaborative development workflow on Git platforms.

Merging is a crucial aspect of maintaining codebase integrity in the context of biological data representation. It enables the seamless integration of changes made in separate branches back into the main codebase. When developers or researchers have completed their work on specific biological character representations, merging ensures that these modifications are smoothly incorporated into the larger project. Through merging, the collaborative efforts of the team are harmonized, and conflicts are resolved to produce a cohesive and comprehensive codebase, facilitating progress, and ensuring the accurate representation of biological data.

Issues in the representation of biological characters in the proposed classes can be effectively addressed through the use of issue tracking systems, such as Git issues. These platforms provide a structured framework for documenting and discussing challenges associated with accurately representing biological concepts within the codebase. Researchers and collaborators can openly engage in dialogues, assign tasks, categorize issues, and

link discussions to code modifications or pull requests. This approach promotes collaboration and transparency while enabling efficient progress tracking and maintaining a comprehensive record of decision-making. Consequently, leveraging issue tracking systems enhances the quality and integrity of biological character representations within the code, contributing significantly to the success of scientific endeavours.

The combination of GitHub versioning and VS Code engenders an environment that augments code management while fostering the open exchange of ideas and expertise. This collaborative approach empowers researchers and developers to collectively contribute to the advancement of biodiversity research, harnessing the power of version control to guarantee accuracy, traceability, and a unified endeavour towards comprehending and preserving the intricacies of biological systems.

The use of Conventional Commits (<https://www.conventionalcommits.org/>) in collaborative GitHub projects can yield significant benefits. By adhering to the standardized commit message format, developers can clearly and succinctly communicate the changes made to the codebase. This aids in comprehending modifications by fellow collaborators, as commit messages follow a consistent pattern. Additionally, automated tools can leverage these messages to automatically generate a detailed change history (changelog), assisting the team in maintaining a track record of alterations over time. With uniform naming and well-organized information, the code review process becomes more efficient as reviewers can swiftly grasp the purpose of each change.

This endeavour has the potential to make significant advancements in biodiversity research and systematics, which may lead to scaling up and innovation. This, in turn, enhances transparency, collaboration, and overall effectiveness in collaborative database development within the Git versioning environment.

Exploring the database

In the domain of DOD, exploring new avenues is of paramount importance. These databases provide a flexible schema approach, capable of accommodating diverse data formats within the same database—a particularly advantageous trait for navigating data characterized by evolving or unpredictable structures. The seamless management of unstructured and semi-structured data aligns harmoniously with the principles of DOD, fostering efficient storage, management, exploration, and analysis without imposing rigid structures. Additionally, these databases excel in managing complex data relationships found in real-world scenarios, directly representing hierarchical data structures, arrays, and nested documents. Moreover, robust aggregation frameworks play a pivotal role, empowering researchers to conduct advanced analyses within the database and extract valuable insights from extensive datasets.

In the context of data exploration, DOD employ sophisticated querying methods to efficiently retrieve and manipulate data. These methods are tailored to the adaptable data representation, enabling researchers to execute complex queries across diverse data structures. For instance, MongoDB (<https://www.mongodb.com/>), a prominent DOD, seamlessly integrates these querying techniques. MongoDB's capacity to handle adaptive schema, support unstructured data, and manage intricate relationships aligns seamlessly with the querying methodologies intrinsic to the nature of DOD. By leveraging MongoDB's querying capabilities, researchers can fully leverage these databases to explore and analyse diverse datasets, including the intricate domain of biological data analysis.

The popularity of JSON has led to the development of various packages, each offering unique ways to query JSON data. These querying packages, like `JsonPath` (<https://github.com/dchester/jsonpath>), `Underscore` (<https://underscorejs.org/>), and `Lodash` (<https://lodash.com/>), are similar to tools that help us find specific information within JSON structures. However, because each tool has its distinct approach, there is no universal method for querying JSON. This diversity can be complex, like having different tools for different tasks. Learning all these tools requires time, and comparing their performance can be challenging. Despite the absence of a standardized query language, the variety of approaches offers a range of possibilities for querying JSON data effectively. As the field evolves, researchers and developers continue to explore and refine methods to improve the efficiency and user-friendliness of querying JSON data.

Numerous querying techniques are available in DOD like MongoDB. However, the standard approach does not typically involve searching for specific keys or executing queries without providing the complete JSON path of keys. Nevertheless, we find ourselves in need of this particular capability. For instance, we aim to locate all keys such as `{"trichomes.setiform.are": "present"}` or `{"obtainingMethod": "SEM"}`, and retrieve database documents that match these conditions, irrespective of their structural placement. This requirement essentially involves conducting an exhaustive nested search, a functionality not readily provided by MongoDB or similar databases. Fulfilling this task requires the development of a recursive function with the ability to traverse all levels of a JSON configuration, a function provided within TTS via the `findProperty` command.

Standard guidelines

Standard recommendations serve as the foundation upon which we establish a cohesive and structured development environment. By diligently adhering to these standards and recommendations, we lay the groundwork for a development ecosystem that not only facilitates clear communication, collaboration, and intuitive coding practices, but also aligns with the intricate domains of TS development and biodiversity data management.

In this database, each piece of data must be organized within a predefined class. For example, `taxa`, `characters`, and `sources` are all depicted as classes, with each class residing within a dedicated `.ts` file.

The underlying structure of organisms, as delineated by taxonomists and biologists, is abstracted into a hierarchical tree of characters. This hierarchical structure is mirrored by a nested arrangement of directories, each containing `.ts` files. Biological characters are encapsulated as classes. If a character-describing class relies on other classes, it manifests as a directory with the class name, which includes an `index.ts` file inside (e.g. the class `Flower`, which imports other classes, is defined as `Flower/index.ts`). This `index.ts` file is imported and exported using the directory's name, implicitly referencing the `index.ts` filename. However, if a character-describing class stands alone without dependencies, it is represented by a standalone `.ts` file with the class's name (e.g. `Calyx.ts`). The interconnections between these components are established through module import and export mechanisms.

For instance, within our pre-built database, the *Mimosa* L. genus is exemplified by a class named `Mimosa`. This class is accompanied by an `index.ts` file within the `Mimosa` directory, providing a comprehensive depiction of the `Mimosa` class.

Similarly, the attributes of a leaf are described within a single `index.ts` file residing within the corresponding directory.

The process of inserting data occurs within the corresponding species file. Each piece of information is accommodated within an object that aligns with a specific class. These objects are tailored to interact seamlessly with their corresponding classes, facilitating the organized storage of data within the database. The initial step involves the instantiation of an object, followed by the population of that object with data. This process is pivotal for accessing object attributes and methods associated with the class instantiated during compilation. It also supports the utilization of autocompletion tools. Within the codebase, a new class instance is created in the following manner: `new ClassName()`.

The act of modelling classes constitutes a foundational task within this database. It entails the definition of attributes and methods. Although this necessitates coding effort, it is paramount for establishing the data types accommodated by each attribute (or property).

Sources are represented as instances of the `Sources` class and can be associated with specific taxa or characters. The instantiation of `Sources`, however, is achieved through the use of the `extends` function in TS. This design decision ensures that every class within this database has the capability to incorporate a source. This class functions as a repository for establishing connections between bibliographic sources, encompassing elements such as images or alternative data formats, and the distinctive structures of organisms.

Directory and file naming conventions

At the project's root, we find the `taxa` and `characters` directories, serving as repositories for our database of genus and species, along with their features, represented as a tree of characters.

Within the `characters` directory, we establish a nested hierarchy of features related to the organism's body, or the tree of characters. Characters are structured into directories according to their hierarchy. Each level of hierarchy is encapsulated within a directory bearing the class name, along with an accompanying `index.ts` file. Within this `index.ts` file, classes utilized within its scope are both imported and exported. On the highest level of the `characters` directory, we encounter initial characters such as 'stems', 'leaves', 'trichomes', 'prickles', 'inflorescence', 'flowers', 'fruits', and 'seeds'. In situations where a class does not rely on any other class, the filename corresponds to the class name with the initial letter capitalized (e.g. `Calyx.ts`).

Within the `taxa` directory, we generate subdirectories designated by the generic epithet (e.g. `Mimosa`). Within each of these subdirectories, an `index.ts` file is generated. This file is responsible for importing all character classes utilized for articulating the attributes of a species belonging to the particular genus, thereby contributing to the comprehensive structure of our database. Enhancing the systematic arrangement, each individual species is detailed within a distinct `.ts` file, situated within its corresponding genus directory (e.g. `./Taxa/Mimosa/Mimosa_sevilhae.ts`).

Character tree

A pre-built database, which houses the character tree located within the `characters` directory, has been created based on taxonomic research involving *Mimosa* (Leguminosae, Caesalpinoideae) [34–37]. As per tradition, it is initially released as version 1. Subsequent projects adhering to these standards can use this project as a starting point.

Describing genus

Within the genus directory, the `index.ts` file should encompass the importation of all characters found at the first depth of the tree of characters, subsequently declaring each of them as attributes. Note that the characters we imported, namely `Stems` and `Leaf`, are described in dedicated `.ts` files within the `character` directory.

The initial lines of code entail the import of all characters defined at the initial depth of the tree of characters, along with the annotation classes. Subsequent to this, the imperative task is the creation of the `Mimosa` class. In extending the `Sources` class to it, the capacity to associate a source with a species is enabled, distinct from attaching it to any of its constituent parts, such as the characters within the tree of characters. Notably, the exportation of the `Mimosa` class should not be overlooked. Refer to the following:

```
// Import characters
import {
  Stems,
  Leaf,
  //other classes of first depth in tree of characters
} from './characters/v1'

// Import annotation classes
import {Sources} from './characters/v1/Sources'
import {DescriptionAuthorship} from './characters/v1/descriptionAuthorship'

export class Mimosa extends Sources {
  specificEpithet: string
  habit: 'tree' | 'shrub' | 'subshrub' | 'herb'
  stems: Stems
  leaf: Leaf
  //other attributes

  constructor() {
    super()
  }
}
```

In the provided code snippet, it is worth emphasizing that, as is conventionally understood, lines starting with `//` are exclusively utilized for commenting within the codebase.

Describing species

Comprehensive information pertaining to a species is meticulously preserved within an individual `.ts` file inside its corresponding genus directory. To describe a species, we need to import the `Mimosa` class representing the *Mimosa* genus, import the characters classes, describe the species using the imported characters, and finally export the class representing this described species. This structure can also accommodate other infraspecific taxa within the database. The `Mimosa` class functions as the central module for the construction of all *Mimosa* species, encompassing the assembly of their respective tree of characters.

Importing modules

The initial lines of code involve importing taxa, characters, and annotation classes. To import these, use the following syntax:

```
// Import genus Mimosa
import {Mimosa} from '.'

// Import characters
import {Stems, Trichomes} from './characters/v1'
import {Capitate} from './characters/v1/Trichomes'
import {CapitateFiliform} from './characters/v1/Trichomes/Capitate'

// Import annotation classes
import {Source, DescriptionAuthorship} from './characters/v1'
```

For a streamlined import process, the technique of object destructuring can be employed, allowing for the efficient acquisition of classes from other interconnected modules.

Description

When describing a species, it is recommended to create a constant utilizing the following syntax: `Mimosa_osmarii`. While variables in JS are commonly named using camelCase, when developing a biodiversity database, it is suggested to use snake-case for variables that store descriptions. Additionally, to adhere to biological nomenclature rules, the genus name should be capitalized—a `Snake_case`.

Use dot notation to create objects within objects. This notation enhances code readability and is akin to a syntactic sugar concept.

To instantiate a class as an object in the description, we need to couple the instantiation within an object that accepts the instantiated class. In the example below, the code functions because the `Mimosa` class has the `specificEpithet` attribute, which receives a value of the primitive type `string`:

```
// Description of Mimosa leptantha
const Mimosa_leptantha = new Mimosa()
Mimosa_leptantha.specificEpithet = 'leptantha'
```

It is important to instantiate each nested class before state an attribute. Effectively describing the presence or absence of specific features requires adherence to precise conventions. The terms `is` and `are` serve as synonyms, facilitating the indication of feature presence or absence based on singular or plural object names. It is advisable to articulate the presence or absence immediately after initializing a new class instance:

```
Mimosa_leptantha.stems = new Stems()
Mimosa_leptantha.stems.trichomes = new Trichomes()
Mimosa_leptantha.stems.trichomes.capitate = new Capitate()
Mimosa_leptantha.stems.trichomes.capitate.filiform = new CapitateFiliform()
Mimosa_leptantha.stems.trichomes.capitate.filiform.are = 'present'
```

Based on the provided example, we are asserting the presence of capitate-filiform trichomes on the stems of *Mimosa leptantha* Benth., at the database.

Annotations

It is strongly recommended that annotations should be declared for last, but of course, before the exportation of the species description. After description. It is important to add the author of the description with the attribute `descriptionAuthorship` that

receive name and a date timestamp. The timestamp will adhere to the Unix timestamp format, representing the count of seconds since 00:00:00 Coordinated Universal Time on 01 January 1970.

```
// Description authorship
Mimosa_leptantha.descriptionAuthorship
= new DescriptionAuthorship()
Mimosa_leptantha.descriptionAuthorship.addAuthor({
  name: 'June Doe',
  date: 1692107172
})
```

A Source class is modelled to store bibtex-like citations, but we add the obtainingMethod attribute, that can be one of these values: nakedEyes, stereoscope, opticalMicroscope, scanningElectronMicroscope, transmissionElectronMicroscope, photo, drawing.

To add a source and link the publication and the plant body, first we need to create a constant that we recommend to store the class Source and we use the method addSource().

```
// Sources

/// Trichomes
const source1 = new Source()
source1.sourceType = 'article'
source1.authorship = 'Jordão, L.S.B. & Morim, M.P. &
Baumgratz, J.F.A.'
source1.year = '2020'
source1.title
= 'Trichomes in *Mimosa* (Leguminosae): Towards a
characterization and a terminology standardization'
source1.journal = 'Flora'
source1.number = '272'
source1.pages = '151702'
source1.figure = '9I, J, K, L'
source1.obtainingMethod
= 'scanningElectronMicroscope'
Mimosa_leptantha.stems.trichomes.capitate
.filiform.addSource(source1)
```

Exporting modules

While there are various ways to export modules in JS and TS, a standard practice is to ensure that the species is fully described, and the last line of the file exports the variable. This approach improves code readability and maintainability.

```
// Export Mimosa leptantha
export {Mimosa_leptantha}
```

Modelling character classes

In TS, character classes are structured as classes themselves. The class name adheres to the PascalCase convention and is defined using the syntax: `class Fruit {}`.

When importing a class for character description, the prevention of duplicate names is crucial. Consequently, addressing the occurrence of repeated character names becomes imperative.

In cases where a structure is named with hyphens, such as 'setiform-capitate' or 'stellate-lepidote', the class name initiates with the first word. In these instances, it transforms into SetiformCapitate and StellateLepidote.

For classes bearing generic names like 'abaxial', 'adaxial', or 'margin', the recommended practice is to lead with the common name, followed by the corresponding anatomical structure. For instance, AbaxialLeaflet and MarginBracteole.

Attribute names within the classes conform to the camelCase pattern and are defined using the syntax: `numberOfPairs: number`. This mirrors the key-value pairs characteristic of JSON objects.

Attributes can encompass a range of data types, including strings and numbers. Here are a few illustrative examples:

Defining a specificEpithet attribute of the Mimosa class that holds a string value:

```
class Mimosa {
  specificEpithet: string
}
```

Establishing a numberOfPairs attribute for the Leaflet class, accommodating a number value:

```
class Pinnae {
  numberOfPairs: number
}
```

Presenting alternative types for an attribute by employing the vertical bar symbol (|):

```
class Replum {
  shape: 'straight' | 'undulate'
}
```

Offering versatility to a multistate character by incorporating (value | value | value) []:

```
class Leaflet {
  shape:
    'linear' |
    'lanceolate' |
    'elliptic' |
    'oval' |
    (
      'linear' |
      'lanceolate' |
      'elliptic' |
      'oval'
    ) []
}
```

After describing the attributes of the class, it is necessary to invoke the constructor() {} function to instantiate each attribute as soon as the module is imported elsewhere.

The methods of the class are listed after the constructor. When it comes to naming methods for characters and documenting their functionality, maintaining a consistent and informative approach is crucial. We've chosen to avoid using native language functions to enhance intuitiveness. Method names follow a standardized format, such as using add for methods like addSource. When specifying data, set is employed, as seen in setLength, to ensure uniformity and clarity.

In the context of this codebase, a set of methods has been developed to manage measurements of different dimensions, specifically length, height, and width. These methods enable the precise definition and manipulation of these measurements for various objects. By accepting numeric values, the setLength, setHeight, and setWidth functions individually establish the primary dimensions. Furthermore, the _setLengthMinMax, setHeightMinMax, and setWidthMinMax functions accommodate the specification of ranges, considering minimum and maximum values. Additionally, the setLengthRarelyMin and setLengthRarelyMax functions allow for the input of occasionally used minimum and maximum values; the same for height and width. These methods collectively

contribute to a comprehensive framework for managing dimensional attributes in accordance with specific conditions and constraints that can be verified at the execution time.

Finally, during export, if a class has attributes that are classes with their own attributes, forming a nested pattern, it is necessary to export not only the created class but also each imported attribute. This is to ensure the coherence of imports and exports for module dependencies.

TS documentation

This document provides guidelines for documenting taxa descriptions, covering both mandatory and optional tags. Mandatory fields include the author's name and date, which attribute proper attribution to the description.

The documentation process employs TSDoc. The initial line should serve as a class title. To enhance class descriptions, the `@remarks` tag can be utilized. Several Markdown markups are functional within TSDoc, and topics can be added using a hyphen at the line's beginning.

An essential tag to consider is `@source`, which enables the addition of bibliographic references. Within the `@source` tag, Markdown notation for HTML references, `[label](link)`, can be employed effectively. Consequently, DOI URLs can be cited to establish direct links from the database.

Taxon documentation

When documenting the attributes of a genus, it is recommended to include the genus name in the initial line of each TSDoc's frame.

Before the `constructor() {}` section, it is advisable to declare 'Creates an instance of [class].' This practice reinforces the principles of OOP. Our database exclusively contains a singular taxon class, `Mimosa`, which represents a genus. For documenting this class, a specialized syntax is employed:

```
export class Mimosa extends Sources {
  /**
   * Species of *Mimosa*
   */
  specificEpithet: string

  /**
   * Habit of *Mimosa*
   */
  habit: 'tree' | 'shrub' | 'subshrub' | 'herb'

  /**
   * Stem of *Mimosa*.
   */
  stems: Stems

  /**
   * Prickles of *Mimosa*.
   */
  prickles: Prickles

  /**
   * Leaf of *Mimosa*: bipinnate
   */
  leaf: Leaf

  /**
   * Creates an instance of *Mimosa* species.

```

```
*/
  constructor() {
    super()
  }
}
```

For the genus description, pertinent documentation regarding main characters can be stored:

```
export class Mimosa extends Sources {
  /**
   * Trichomes of *Mimosa*.
   *
   * @remarks
   * Trichomes in *Mimosa* can be classified in the
   * following types:
   * - **filiform**
   * - **setiform**
   * - **stellate**
   * - **dendritic**
   * - **porrect**
   * - **fasciculate**
   * - **verruciform**
   * - **lepidote**
   * - **granular**
   * - **capitate-filiform**
   * - **capitate-setiform**
   *
   * @source [Jordão et al. (2020)] (https://doi.org/10.1016/j.flora.2020.151702)
   * @source [Santos-Silva et al. (2013)] (https://doi.org/10.11646/phytotaxa.119.1.1)
   */
  trichomes: Trichomes

  /**
   * Creates an instance of *Mimosa* species.
   */
  constructor() {
    super()
  }
}
```

Character documentation

To document a character class, it is recommended to include the name of the parent class in each attribute. Within `@remarks`, consider listing the attribute types as separate topics and boldening them using the `**` markup. After, provide a descriptive explanation. This approach ensures clear and comprehensive type descriptions.

As illustrated in the example below, the attribute name is reiterated and linked to its associated class. This practice ensures the cohesion of assembling objects within the character tree:

```
export class Epicarp extends Sources {
  /**
   * Type of epicarp.
   */
  type: 'monospermic' | 'undivided'

  /**
   * Margin of epicarp.
   *
   * @remarks
   * The epicarp margin can be: 'straight' or 'undulate':

```



```

* - **straight**: The epicarp margin is straight and
not undulated.
* - **undulate**: The epicarp margin is undulated
and not straight.
*/
margin: 'straight' | 'undulate'

/**
* Creates an instance of Epicarp.
*/
constructor () {
  super ()
}
}

```

Exporting the database

In order to export the JSON database, we present a method that involves the retrieval of all species files from the genus directory and their organization into an array of objects. Additionally, we have implemented a layer of validation that checks for duplicates within these arrays and removes them, but other validations can be implemented.

The `Sources` class, represented as an array of objects, can be found at all depths within the taxa description and the tree of characters. To facilitate data retrieval and querying, especially for source-related data, we have included a method to export a source database. This database allows us to retrieve positional information (such as index and JSON path) within the main database. This information is particularly useful for pinpointing the exact character associated with a specific source.

Git versioning

In the context of Git versioning, commit messages can adhere to a standardized semantics for clear communication, and even certain automation on Git system.

Adding a new taxon

In scenarios where a new taxonomic entry is introduced to the database, the suggested commit message format is exemplified as follows:

```

feat (taxon) : Add taxon "Mimosa osmarii"
Add taxon to the database.

```

By encapsulating the action ('Add'), the affected entity ('taxon'), and the specific entity's name ('Mimosa osmarii'), this format concisely communicates the essence of the commit. The accompanying description provides further context, ensuring that fellow developers understand the nature of the addition.

Updating data of taxon

When updates are made to the data associated with an existing taxon, the commit message adheres to the following structure:

```

feat (taxon) : Update taxon "Mimosa osmarii"
Update data of taxon in the database.

```

This message structure harmoniously conveys the alteration made ("Update taxon"), specifies the taxon being modified ("Mimosa osmarii"), and offers a brief description of the change itself.

Add new character

Introducing new characters into the database follows a similar semantic framework:

```

feat (character) : Add character "Leaf", "Petiole"
Add character (s) to the database.

```

The "feat(character)" identifier indicates the addition of a character, followed by the character names. The accompanying description provides clarity on the action taken.

Updating character

Updating existing character information is likewise captured within this structure:

```

feat (character) : Update character "Leaf", "Petiole"
Update character (s) in the database.

```

The consistency in structure enables swift comprehension of the change ('Update character'), identifies the specific characters modified ('Leaf' and 'Petiole'), and offers a concise summary of the update itself.

Tutorial of TTS

Install Node.js

Before you begin, ensure that Node.js is installed on your system. Node.js is essential for running JS applications on your machine. You can download and install it from the official Node.js website (<https://nodejs.org/>).

Install VS Code

VS Code is a versatile code editor that provides a user-friendly interface and a plethora of extensions for enhanced development. Download and install VS Code from its official website (<https://code.visualstudio.com/>) to utilize its features for your project.

Clone the repository from GitHub in VS Code

To clone the *Mimosa* project repository for TTS from GitHub, follow these steps:

1. In VS Code, access the Command Palette by pressing `Ctrl + Shift + P` (Windows/Linux) or `Cmd + Shift + P` (macOS).
2. Type `Git: Clone` and select the option that appears.
3. A text field will appear at the top of the window. Enter the URL of the repository you want to clone. In this case, use <https://github.com/lbjordao/TTS-Mimosa>.
4. Choose a local directory where you want to clone the repository to.

We highly recommend using a path for cloning the repository that excludes spaces () or any other unconventional text characters. This precaution ensures that files can be easily opened by simply pressing `Ctrl +` clicking on the file path within the IDE's console.

Open the TTS project directory in VS Code

To open the TTS project directory in VS Code:

1. Click on `File` in the top menu.
2. Select `Open Folder` from the dropdown menu.
3. Navigate to the location where your TTS project (e.g. TTS-Mimosa) directory is stored.
4. Click on the TTS project directory to select it.
5. Click the `Open` button.

Installing TS globally

Within VS Code, open your terminal and execute the command below, following these steps:

1. Navigate to the top menu and select Terminal.
2. From the dropdown menu, choose New Terminal.
3. In the terminal, type and execute the following command:

```
npm install -g typescript
```

Installing TTS package globally

Within a terminal in VS Code, type and execute the following command:

```
npm install -g @lsbjordao/type-taxon-script
```

Install it globally using `-g` to prevent unnecessary dependencies from being installed within the TTS project directory. If one do not include `-g` argument, the `./node_modules` directory and `package.json` file will be inconveniently created in the TTS project directory.

To verify the installation of the TTS, use the following command to check the current version:

```
tts -version
```

For comprehensive guidance on available commands and functionalities, access the help documentation using:

```
tts -help
```

Uninstalling TTS package

To uninstall TTS package, open your terminal and execute the command at the root, where is the `package.json`:

```
npm uninstall -g @lsbjordao/type-taxon-script
```

Initializing a TTS project

To initiate the use of a TTS project, execute the following command:

```
tts init
```

This command will verify the presence of an existing TTS project within the directory. Additionally, it will generate two mandatory directories, `./input` and `./output`, but only if the `characters` and `taxon` directories already exist. These newly created directories are essential for the project functioning.

Describing a new taxon

To generate a new `.ts` file containing a comprehensive script outlining the entire hierarchy of characters, serving as the foundational template to initiate the description of a species from scratch, utilize the command `tts` followed by the `-new` argument, specifying the genus name and the specific epithet as shown below:

```
tts new -genus Mimosa -species epithet
```

After the process, a new file named `Mimosa_epithet.ts` will be created in the `./output` directory. To access this script file, simply hold down the `Ctrl` key and click on the file path displayed in the console. However, before you begin editing the script, it is important to relocate this file to the `./taxon` directory, as the script specifically functions within that directory. Outside this directory, the script will not work properly. Opening the script outside of this directory will trigger multiple dependency errors.

Importing from .csv file

It is also possible to import data of multiple taxa from a `.csv` file with a header in the following manner:

```
tts import -genus Mimosa
```

The `.csv` file is formatted to be compatible with MS Excel, utilizing the separator; and `"` as the string delimiter. The only required field is `specificEpithet`. Each column should be named according to the complete JSON path of the corresponding attribute. All values are imported automatically.

To indicate multiple states within a cell, utilize this syntax: `['4-merous', '5-merous']`, as demonstrated in the `./input/importTaxa.csv` file.

If we want to describe a specific characteristic, which is a key object, we need to fill the column name with its JSON path and enter `yes` in the cell where that characteristic needs to be automatically instantiated. For example, if we have inflorescence types `'capitate'` and `'spicate'`, to instantiate the respective class within the file, in the `.csv` table, we create columns `inflorescence.capitate` and `inflorescence.spicate` and enter `yes` in the cells of the respective taxon. Of course, only one of them is possible in the plant body, and we should not instantiate both concurrently. See the example provided in the `./input/importTaxa.csv` file.

The generated `.ts` files will be located in the `./output` directory. Since the script operates exclusively within the `./taxon` directory, it is necessary to relocate all these files to that specific directory for the script to function properly.

Documentation

Every element within the code is accompanied by metadata (Fig. 1). Simply hover your cursor over an element, and its metadata will promptly appear.

Taxon edition

To edit a species `.ts` file, open it and utilize the `'` key after the `=` sign to access attribute options (Fig. 2). After that, press `Enter`. The autocompletion feature will assist in completing the entry:

Cross-referencing

Every class is interconnected through cross-referencing. By holding down `Ctrl` and clicking on a class, the associated `.ts` file containing the class description will open automatically. This feature allows us to seamlessly navigate through the character tree hierarchy.

Furthermore, we have the capability to track down instances where a class is employed. For example, when we seek to identify occurrences of a character class being used, we can easily inspect the class name. As illustrated in the given example, a `Gall` is mentioned in the description of `Mimosa gemmulata` Barneby, and by clicking on it, we can promptly open its respective file (Fig. 3).

Multi-line edition

Use the shortcut `Ctrl + Shift + L` for efficient multi-line editing (Fig. 4). Press `Esc` to end the multi-line edition.

Automatic code formatting

When you right-click on any content in a file and select `Format Document` in VS Code, the code is automatically adjusted for indentation, spacing, and more. This feature simplifies code maintenance and helps maintain a consistent coding style throughout your code.

Git versioning

Within VS Code, a quick click on a file listed in the `Git` panel allows you to instantly inspect code changes (Fig. 5). As you open the file, a split-screen emerges, delineating

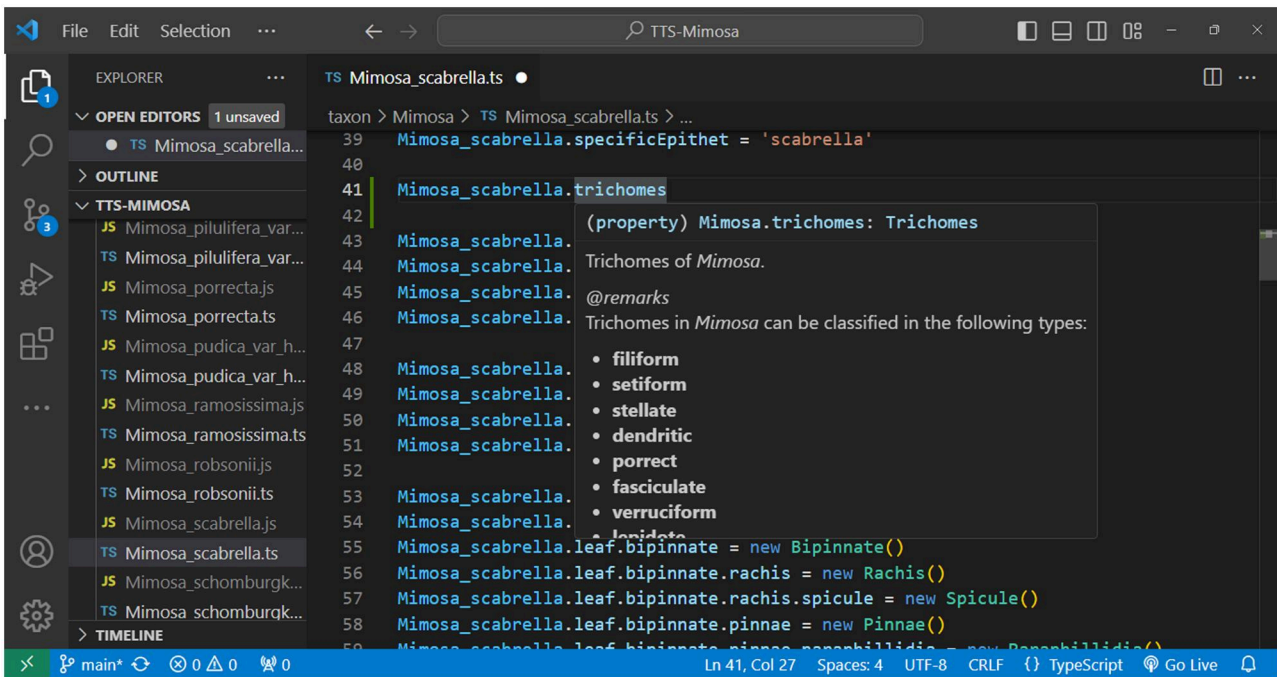


Figure 1 When hovering the cursor over a text in an IDE like VS Code, a popup will display its metadata

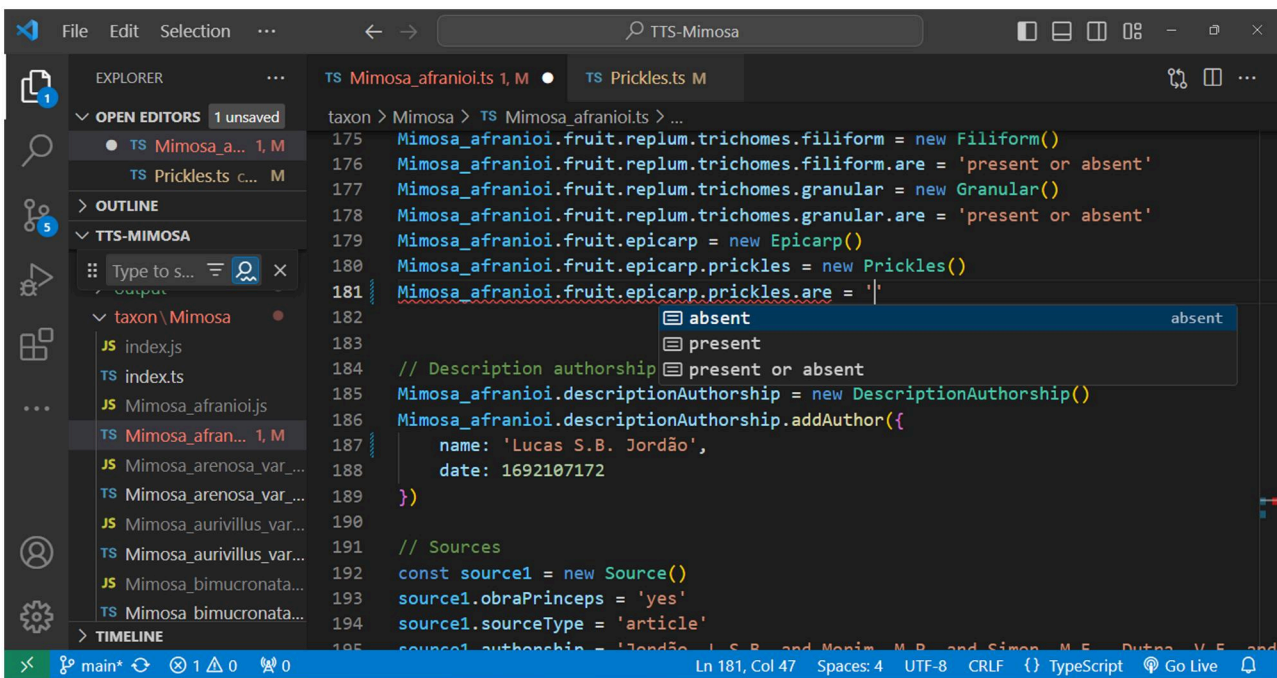


Figure 2 VS Code assists with its autocomplete tool, displaying all allowed states for each property

alterations in green (edits) and red (revisions) in contrast to the previous version of the code. This functionality streamlines the review process, providing an intuitive and efficient means to track modifications in your development environment.

VS Code offers a range of features and extensions to streamline conflict resolution. These include interactive merge tools, side-by-side file comparison, and even built-in three-way merge support. We can manage the Git versioning process using simple clicks of a button.

Export .json database

To export all taxa inside `./taxon/Mimosa`, type:

```
tts export -genus Mimosa
```

If you intend to generate a database containing a specific list of taxa from the directory `./taxon/Mimosa`, edit the `./input/taxonToImport.csv` file accordingly. After making the necessary edits, execute the following command:

```
tts export -genus Mimosa -load csv
```

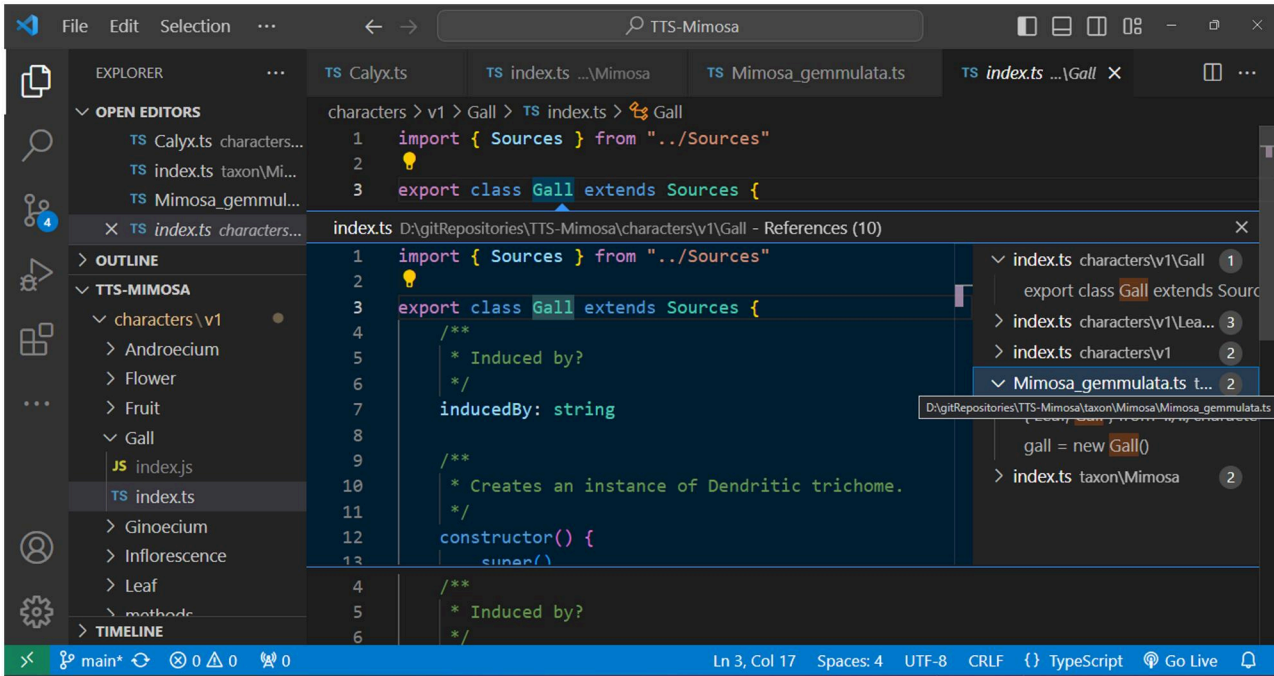


Figure 3 Module dependency cross-referencing aids in quickly identifying where a character is invoked, while VS Code efficiently tracks and displays the usage of each character from the character tree

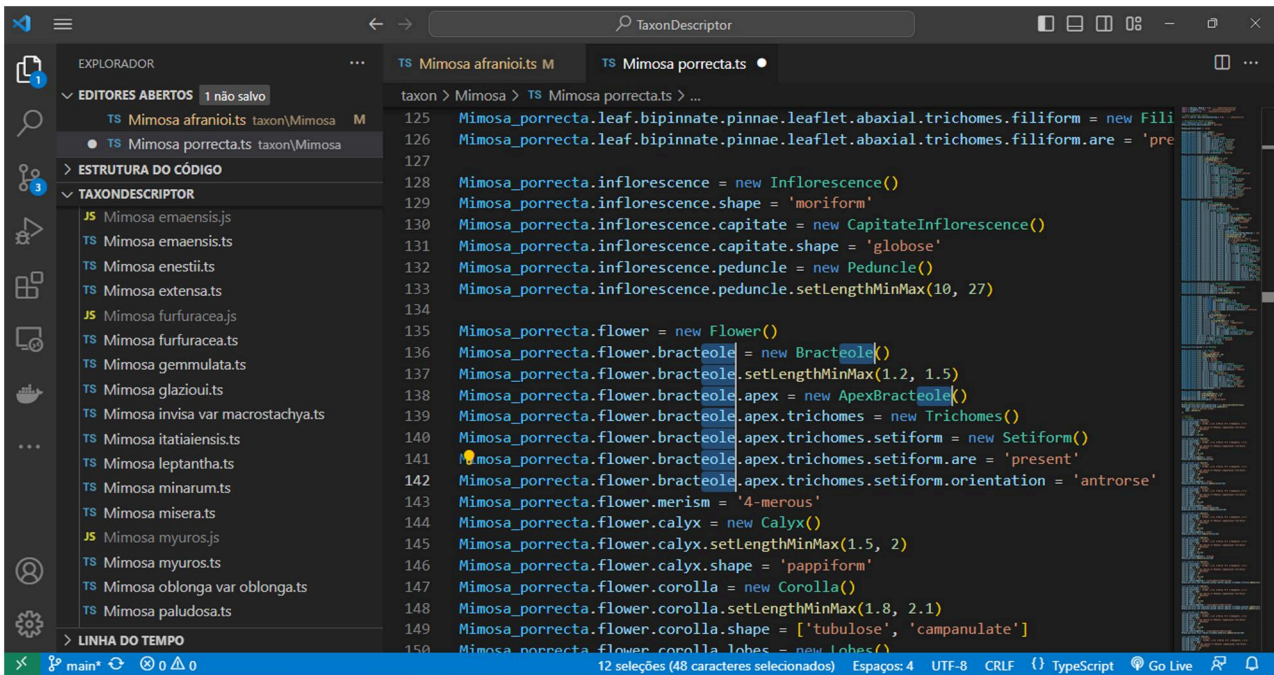


Figure 4 Code editing can sometimes be challenging, but VS Code provides helpful tools to make it more agile, such as multi-line editing

The resulting JSON database $\{\text{genus}\}\text{DB.json}$ file will be generated and stored in the directory `./output/`.

Errors may arise twice in the export process: once during the compilation (TS) phase and again during the execution (JS) stage.

Regarding compilation errors, for instance, two issues were encountered in files `Mimosa_test.ts` and `Mimosa_test2.ts` while attempting to export the *Mimosa* database (Fig. 6). In the `Mimosa_test.ts` script, an undeclared property for the adaxial surface of the leaflet was caught. In the `Mimosa_test2.ts`

script, the class `ractoole` was listed as a property of `flower`, but the error message suggests the correction to `bracteole`. See below:

And errors can be caught during the execution phase. In the case below, a stipule length was set with its minimum value as 5 and its maximum as 3 using the `.setHeightMinMax()` method (Fig. 7). Such an error will not be caught during compilation as the type is correct (number), but during execution, a message in the terminal indicates that the

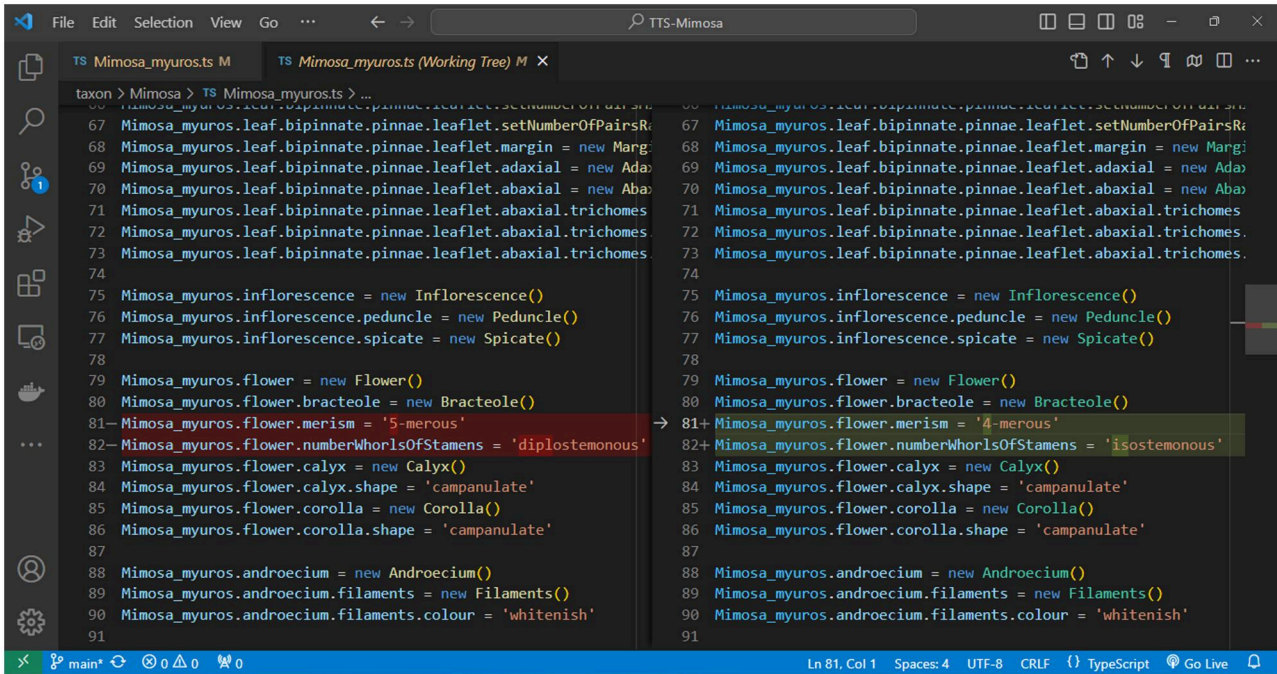


Figure 5 VS Code seamlessly integrates with GitHub, making it user-friendly even for individuals with minimal expertise, requiring only a small learning curve

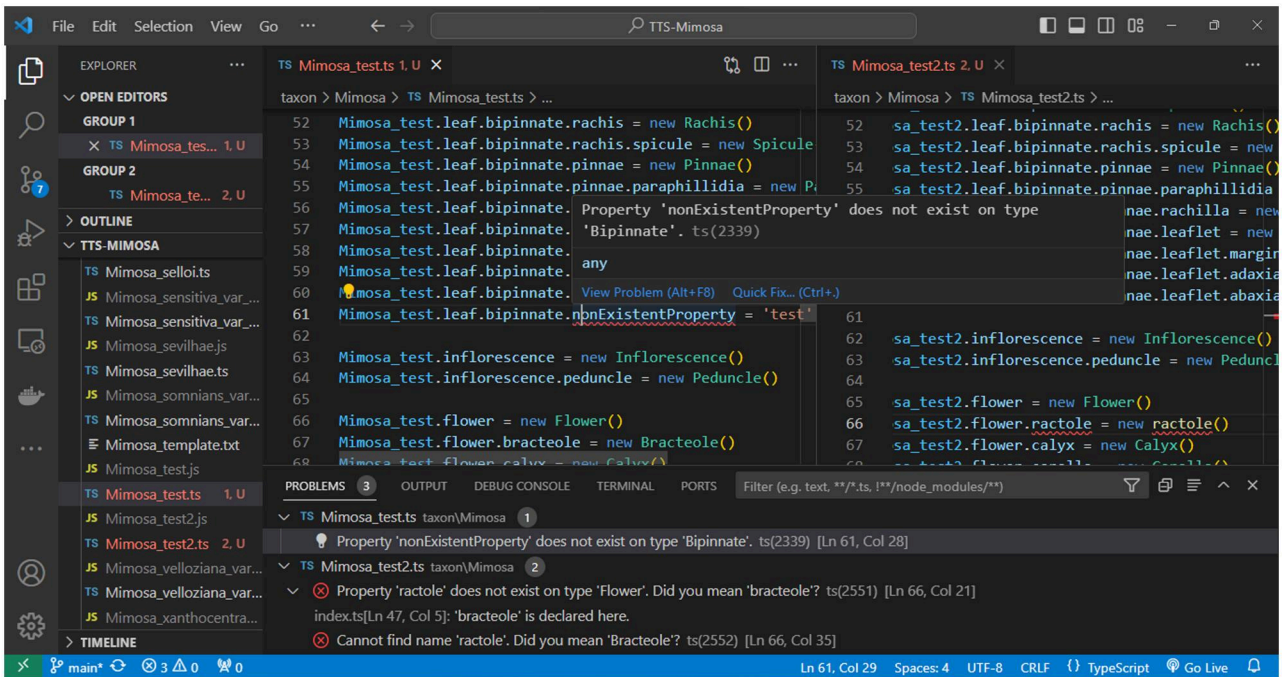


Figure 6 Errors occurring during the transpilation process from .ts to .js files result from violations of established syntax rules or type inconsistencies within the TypeScript code. These violations encompass syntax errors, incorrect typing, or improper use of TypeScript language features. Each error message provides specific details aiding in the identification and resolution of the precise issue encountered

“minimum height must be less than the maximum height.”
See below:

Sources dataset

We can create a consolidated dataset that compiles all sources into a flatter JSON structure, enabling simpler query access. To generate a database solely containing sources related to the taxa, execute the following command:

```
tts exportSources -genus Mimosa
```

This dataset includes an index that relates to the main database and provides the complete key path where each source is located:

```
{
  index: 7,
  path: 'flower.corolla.trichomes.stellate.lepidote',
```

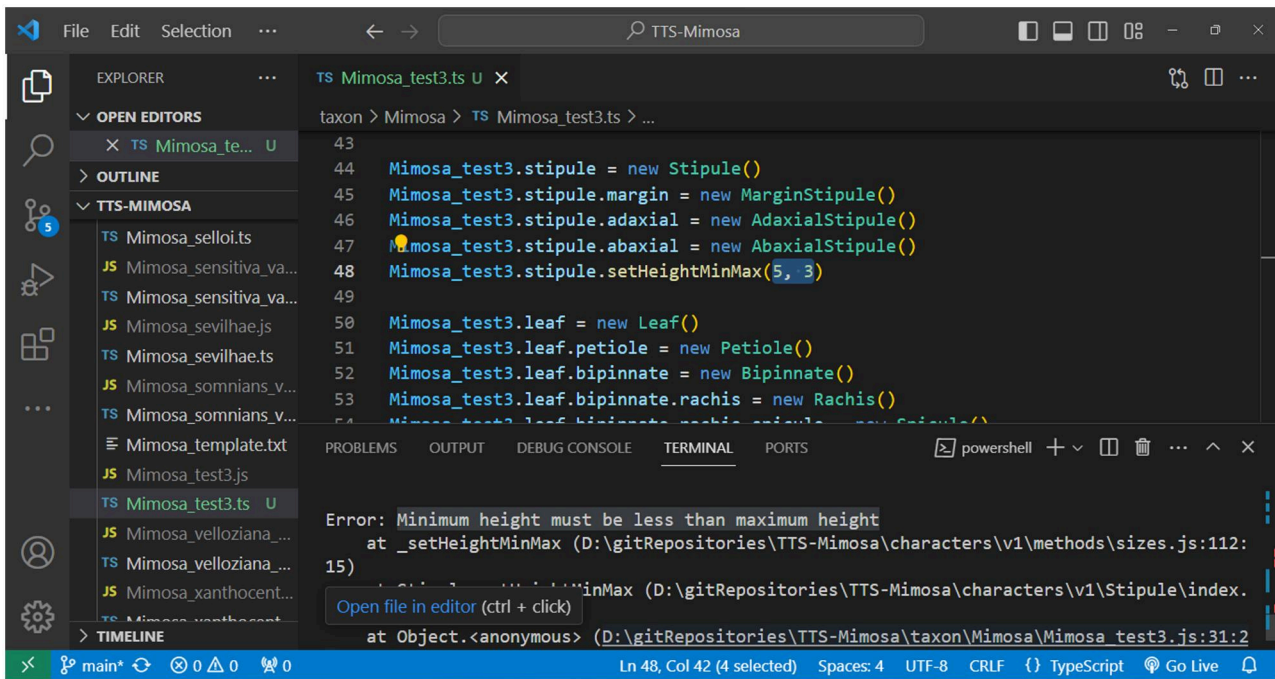


Figure 7 Errors stemming from the compilation of .js files preempt runtime errors, executing the business rules defined within each class method. In this example, despite no data type violation, as the function accepts two numbers, there exists a business rule to validate whether the minimum value is truly less than the described maximum value. This process prevents data insertion errors, ensuring adherence to predefined business rules

```

source: {
  sourceType: 'article',
  authorship: 'Jordão, L.S.B. & Morim, M.P.
& Baumgratz, J.F.A.',
  year: '2020',
  title: 'Trichomes in *Mimosa* (Leguminosae): Towards a
characterization and a terminology standardization',
  journal: 'Flora',
  number: '272',
  pages: '151702',
  figure: '4I',
  obtainingMethod: 'scanningElectronMicroscope'
}
}]

```

Export .csv database

In TTS, there is a convenient method for exporting a converted JSON database to a CSV format using the 'exportToCsv' command. Similar to the 'export' command, it generates a CSV output. Indeed, opening a CSV file in a spreadsheet is undoubtedly helpful. However, it is important to note that this conversion result in a loss of data structure. The first line of the CSV will be comprised by all JSON paths, preserving the nested hierarchy, but arrays of elements will be treated as strings with the same JSON syntax. For more advanced transformations, one can explore additional methods in Json2csv (<https://mircozeiss.com/json2csv/>).

Navigating the database

Utilizing the JSON Grid Viewer extension (<https://github.com/dutchigor/json-grid-viewer>), which is readily accessible on the Visual Studio Marketplace (<https://marketplace.visualstudio.com/>), we can effortlessly delve into the intricate structure of JSON configurations (Fig. 8).

Querying methods

Data querying techniques encompass a range of methods tailored to diverse needs. Basic querying relies on key-value pairs for precise data retrieval, while range queries are optimal for numerical, or date-based data, allowing data extraction within specified value ranges.

Another type of query method involves the aggregation approach, which provides advanced data manipulation capabilities, enabling chain operations such as grouping and filtering within the database. This is made possible because the result of a query always returns the complete document within the database. Thus, additional queries can be chained to perform multiple filter aggregations.

Character path querying

An essential aspect of querying is to identify a JSON path that represents nested properties within an array of documents in a JSON database. In this particular scenario, our objective is to navigate the character tree to retrieve taxa properties.

Let us define a 'property' as a JSON path of keys within the character tree. When we need to retrieve a property from the database, we search for its corresponding JSON path, such as `trichomes.stellate`. This search yields the indices of the documents where the property was found and the paths where it was located, achieved using the `findProperty` command:

```
tts findProperty -property trichomes.stellate
-genus Mimosa
```

The result should be similar to:

```
// Indices and paths of objects with the property:
// "trichomes.stellate":
[
  {specificEpithet: 'furfuraceae', index: 5, paths:
  ['flower.corolla']},
```

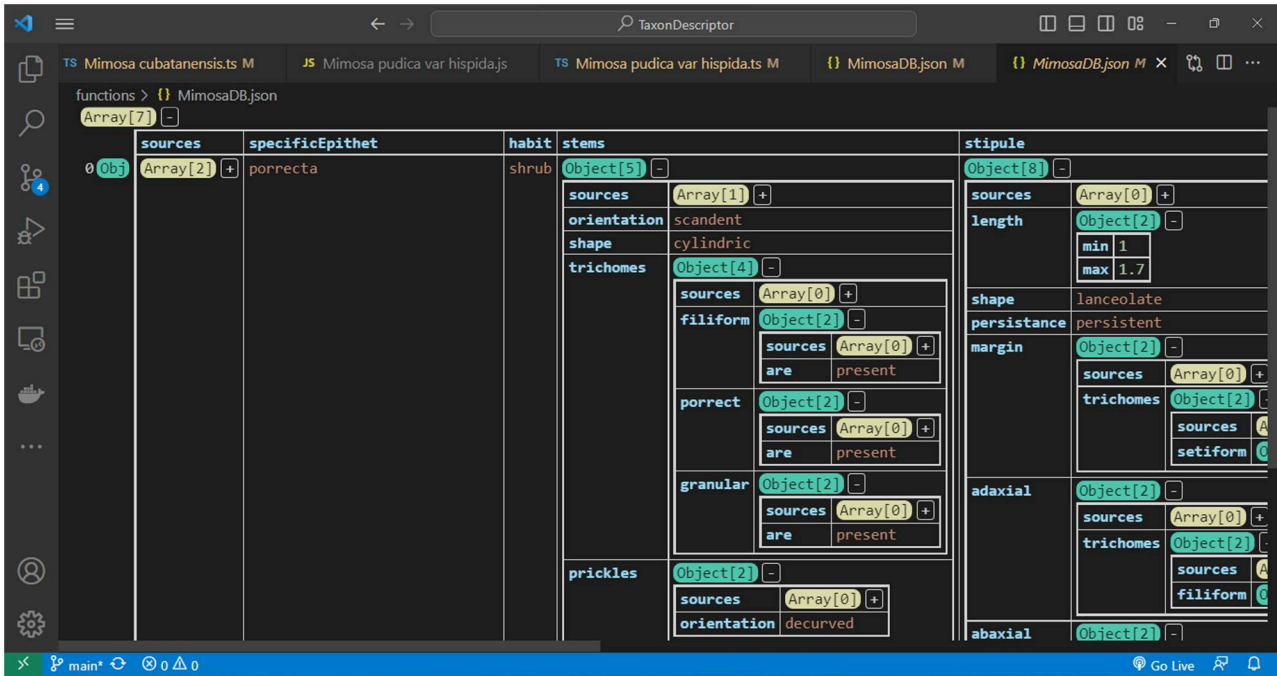


Figure 8 The JSON-grid-viewer package is highly useful as it provides a user-friendly visualization of JSON files. Its primary focus is on visualizing objects and arrays, which are typical components of a JSON structure

```
{specificEpithet: 'myuros', index: 6, paths: ['stems']},
{
  specificEpithet: 'schomburgkii',
  index: 7,
  paths: [
    'leaf.bipinnate.pinnae.leaflet.abaxial',
    'flower.corolla'
  ]
}
```

In the preceding example, stellate trichomes were identified within the corolla of *M. furfuraceae* Benth., the stems of *M. myuros* Barneby, and both the abaxial surface of leaflet and corolla in *M. schomburgkii* Benth.

Flexible key-value querying

Another querying approach for property querying involves flexible key-value querying. This method enables searching within a JSON path using a specific value that can meet defined conditions.

To initiate queries within a TTS project export, perform these operations outside the project's directory. Begin by creating a separate directory for a new project, naming it as desired (e.g. `flex-json-searches`). Open this directory using an IDE like VS Code.

For flexible JSON searching, installation of the `flex-json-searcher` (<https://github.com/vicentecalf/flex-json-searcher>) and `fs` modules is necessary. While the `fs` module is intrinsic to basic file processing in Node.js, the `flex-json-searcher` module offers comprehensive functionality tailored for diverse JSON file queries. To install these modules, open a new terminal and execute the following commands:

```
npm install fs
```

```
npm install flex-json-searcher
```

Next, create a JS file (e.g. `script.js`) within the project root directory and use the following code snippet as a reference to perform flexible JSON searches:

```
// script.js
const fs = require('fs')
const {FJS} = require('flex-json-searcher')
const filePath = './output/MimosaDB.json'
fs.readFile(filePath, 'utf8', async (err, data) => {
  if (err) {
    console.error('Error reading the file:', err)
    return
  }

  try {
    const mimosaDB = JSON.parse(data)
    const fjs = new FJS(mimosaDB)
    const query = {'flower.merism': {'$eq': '3-merous'}}
    const output = await fjs.search(query)
    const specificEpithets = output.result.
      map(item => item.specificEpithet)
    console.log('Species found:', specificEpithets)
  } catch (error) {
    console.error('Error during processing:', error)
  }
})
```

After saved, run the following line in the terminal:

```
node script
```

The result should be similar to:

```
Species found: [
  'afranoi',
```

```
'caesalpinifolia',
'ceratonia var pseudo-obovata',
'robsonii'
]
```

During the search, `*` can be employed to locate a particular JSON path associated with a value determined by specific conditions.

Range querying

Range querying involves searching for and retrieving data within a specific range of values or criteria, such as a range of dates, numerical values, or any other defined attributes.

To perform range querying, we rely on the `fs` and `flex-json-searcher` modules, both of which need to be installed. To do this, within the VS Code terminal of a new project directory, execute the following command:

```
npm install fs
npm install flex-json-searcher
```

Next, create a `script2.js` file within the project root directory with the code below:

```
// script2.js
const fs = require('fs')
const {FJS} = require('flex-json-searcher')

const filePath = './output/MimosaDB.json'

fs.readFile(filePath, 'utf8', async (err, data) => {
  if (err) {
    console.error('Error reading the file:', err)
    return
  }

  try {
    const mimosaDB = JSON.parse(data)
    const fjs = new FJS(mimosaDB)
    const query = {'leaf.bipinnate.pinnae.leaflet.numberOfPairs.min': {'$gt': '15'}}
    const output = await fjs.search(query)

    const specificEpithets = output.result.map(item => item.specificEpithet)
    console.log('Species found:', specificEpithets)

  } catch (error) {
    console.error('Error during processing:', error)
  }
})
```

In terminal, run:

```
node script2
```

The result should be similar to:

```
Species found: [
  'bimucronata',
  'bocainae',
  'dryandroides var. dryandroides',
  'elliptica',
  'invisa var. macrostachya',
  'itatiaiensis',
  'pilulifera var. pseudincana'
```

```
]
```

In the preceding example, we are conducting a query to find species with a minimum leaflet pairs number greater than 15.

We leverage the output `.result` to chain queries or perform query aggregations, allowing us to achieve multiple filtering operations within the database. To perform a dual conditional query using 'greater than' and 'less than' conditions, try the code below by creating a `script3.js` file:

```
// script3.js
const fs = require('fs')
const {FJS} = require('flex-json-searcher')

const filePath = './output/MimosaDB.json'

fs.readFile(filePath, 'utf8', async (err, data) => {
  if (err) {
    console.error('Error reading the file:', err)
    return
  }

  try {
    const mimosaDB = JSON.parse(data)
    const fjs = new FJS(mimosaDB)

    //First query with criteria greater than 15
    const gt15Query = {'leaf.bipinnate.pinnae.leaflet.numberOfPairs.min': {'$gt': '15'}}
    const gt15Output = await fjs.search(gt15Query)

    const gt15SpecificEpithets = gt15Output.result.map(item => item.specificEpithet)
    console.log('Species with more than 15 leaflet pairs found: \n', gt15SpecificEpithets)

    //Second query using the results of the first search
    const fjs2 = new FJS(gt15Output.result)
    const lt20Query = {'leaf.bipinnate.pinnae.leaflet.numberOfPairs.min': {'$lt': '20'}}
    const lt20Output = await fjs2.search(lt20Query)

    const lt20SpecificEpithets = lt20Output.result.map(item => item.specificEpithet)
    console.log('Species with less than 20 leaflet pairs found: \n', lt20SpecificEpithets)

  } catch (error) {
    console.error('Error during processing:', error)
  }
})
```

In terminal, run:

```
node script3
```

The result should be similar to:

```
Species with more than 15 leaflet pairs found:
[
  'bimucronata',
  'bocainae',
  'dryandroides var. dryandroides',
  'elliptica',
  'invisa var. macrostachya',
```



```
'itatiaiensis',
'pilulifera var. pseudincana'
]
Species with less than 20 leaflet pairs found:
['bimucronata', 'itatiaiensis',
'pilulifera var. pseudincana']
```

Source querying

In the exported sources database, we have the capability to perform queries and retrieve specific information. For instance, we can query the database to obtain all images captured using a scanning electron microscope. To accomplish this, create a `script4.js` file and insert the following code:

```
// script4.js
const fs = require('fs')
const {FJS} = require('flex-json-searcher')

const filePath = './output/MimosaSourcesDB.json'

fs.readFile(filePath, 'utf8', async (err, data) => {
  if (err) {
    console.error('Error reading the file:', err)
    return
  }

  try {
    const mimosaSourcesDB = JSON.parse(data)
    const fjs = new FJS(mimosaSourcesDB)
    const query = {'source.obtainingMethod':
      : {$eq: 'photo'}}
    const output = await fjs.search(query)
    console.log(output.result)
  } catch (error) {
    console.error('Error during processing:', error)
  }
})
```

In terminal, run:

```
node script4
```

The result should be similar to:

```
[
  {
    index: '0',
    path: '',
    specificEpithet: 'afranioi',
    source: {
      obraPrinceps: 'yes',
      sourceType: 'article',
      authorship: 'Jordão, L.S.B. and Morim, M.P. and
Simon, M.F., Dutra, V.F. and Baumgratz, J.F.A.',
      year: '2021',
      title: 'New Species of *Mimosa* (Leguminosae)
from Brazil',
      journal: 'Systematic Botany',
      volume: '46',
      number: '2',
      pages: '339-351',
      figure: '3',
      obtainingMethod: 'photo'
    }
  },
  {
    index: '17',
    path: '',
    specificEpithet: 'emaensis',
    source: {
      obraPrinceps: 'yes',
      sourceType: 'article',
      authorship: 'Jordão, L.S.B. and Morim, M.P. and
Simon, M.F., Dutra, V.F. and Baumgratz, J.F.A.',
      year: '2021',
      title: 'New Species of *Mimosa*
(Leguminosae) from Brazil',
      journal: 'Systematic Botany',
      volume: '46',
      number: '2',
      pages: '339-351',
      figure: '5',
      obtainingMethod: 'photo'
    }
  },
  {
    index: '21',
    path: 'leaf.bipinnate.pinnae.gall',
    specificEpithet: 'gemmulata',
    source: {
      sourceType: 'article',
      authorship: 'Vieira, L.G. & Nogueiro, R.M. & Costa,
E.C. & Carvalho-Fernandes, S.P. & Santos-Silva, J.',
      year: '2018',
      title: 'Insect galls in Rupestrian field and
Cerrado stricto sensu vegetation in Caetité,
Bahia, Brazil',
      journal: 'Biota Neotrop.',
      number: '18',
      volume: '2',
      figure: '2P, Q',
      obtainingMethod: 'photo',
      doi: 'https://doi.org/10.1590/1676-0611-BN-2017-0402'
    }
  }
]
// ...
]
```

```
{
  index: '17',
  path: '',
  specificEpithet: 'emaensis',
  source: {
    obraPrinceps: 'yes',
    sourceType: 'article',
    authorship: 'Jordão, L.S.B. and Morim, M.P. and
Simon, M.F., Dutra, V.F. and Baumgratz, J.F.A.',
    year: '2021',
    title: 'New Species of *Mimosa*
(Leguminosae) from Brazil',
    journal: 'Systematic Botany',
    volume: '46',
    number: '2',
    pages: '339-351',
    figure: '5',
    obtainingMethod: 'photo'
  }
},
{
  index: '21',
  path: 'leaf.bipinnate.pinnae.gall',
  specificEpithet: 'gemmulata',
  source: {
    sourceType: 'article',
    authorship: 'Vieira, L.G. & Nogueiro, R.M. & Costa,
E.C. & Carvalho-Fernandes, S.P. & Santos-Silva, J.',
    year: '2018',
    title: 'Insect galls in Rupestrian field and
Cerrado stricto sensu vegetation in Caetité,
Bahia, Brazil',
    journal: 'Biota Neotrop.',
    number: '18',
    volume: '2',
    figure: '2P, Q',
    obtainingMethod: 'photo',
    doi: 'https://doi.org/10.1590/1676-0611-BN-2017-0402'
  }
}
// ...
]
```

The complete information for each source is readily accessible, such as the `sourceType`, `journal`, `figure`, `authorship`.

Other querying applications

MongoDB and its companion tool, MongoDB Compass, offer advanced querying capabilities (Fig. 9). MongoDB's query language, empowered by methods like `find()` and a rich set of comparison operators such as `$lt` (less than), `$gt` (greater than), and `$eq` (equal to), allows precise document filtration based on specific criteria. MongoDB Compass, a graphical interface for MongoDB, provides an intuitive platform to visually construct and execute queries. It simplifies query creation, data visualization, and optimization by offering a user-friendly graphical representation of data structures. Leveraging MongoDB's querying prowess along with Compass's interactive interface enables users to proficiently explore, retrieve, and manipulate data within MongoDB databases.

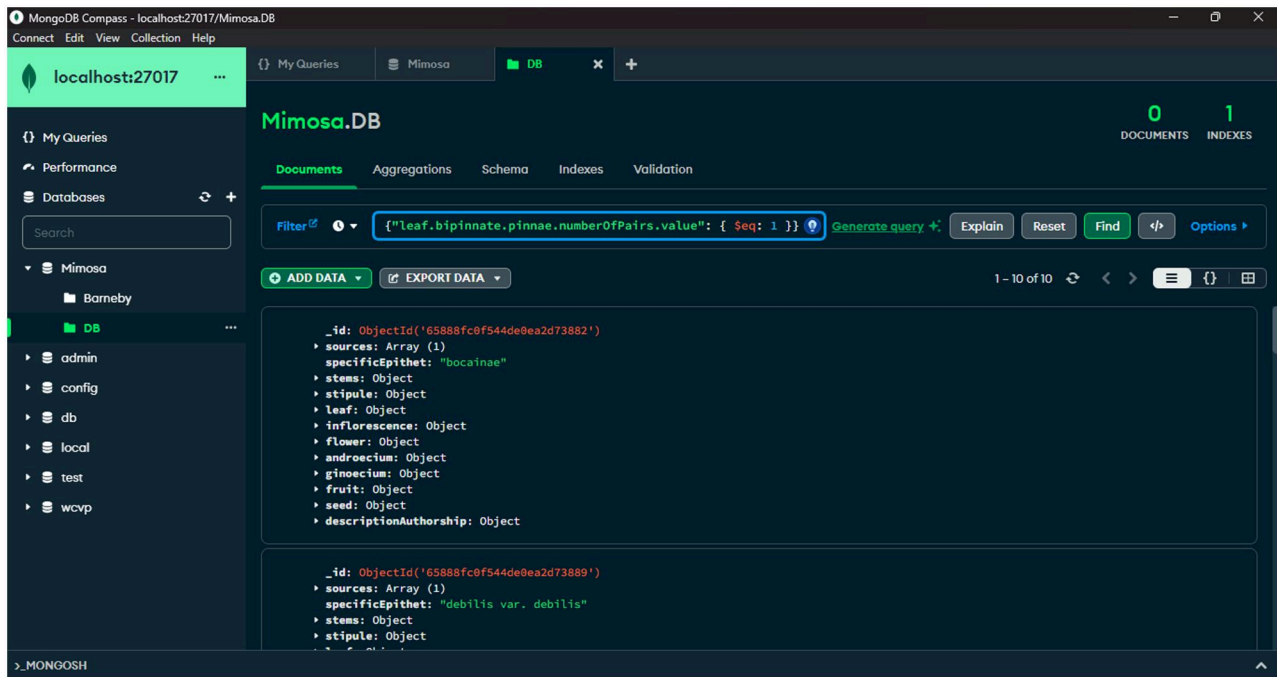


Figure 9 MongoDB Compass offers a robust framework for visualizing and investigating DOD, enabling powerful query capabilities

Call to action

Not Word or Excel, but TTS. We stand at the threshold of a new era in biological taxonomy descriptions. In this methodology, software engineering methods using TS are employed to build a robust data structure, fostering enduring, non-redundant collaborative efforts through the application of a kind of taxonomic engineering of biological bodies. This innovative programming introduces a new approach for precise and resilient documentation of taxa and characters, transcending the limitations of traditional text and spreadsheet editors. TTS streamlines and optimizes this process, enabling meticulous and efficient descriptions of diverse organisms including annotations, propelling the science of taxonomy and systematics to elevate levels of collaboration, precision, and effectiveness.

In the realm of collaborative work, even amid challenges, lies a critical opportunity. Our proposal seeks to galvanize a broader adoption by the collective community of biologists, taxonomists, and systematists. This endeavor entails providing an open-access infrastructure that facilitates collaborative efforts, aligning with the creation of object classes. This unity ensures the safeguarding of data integrity, eliminating the need for repetitive descriptions of fundamental components like the 'leaf' class across various initiatives. By structuring attributes of classes, we establish a streamlined approach that simplifies the work for subsequent contributors. Each description necessitates the instantiation of these classes as objects, fostering a taxonomic engineering-like approach to create virtual representations of biological bodies.

Within this collaborative framework, roles are clearly delineated—some focus on database modelling, while others engage in data population. The process of database modelling involves shaping the object classes that form the foundation of the database. To achieve consensus in this dynamic task, community engagement is pivotal. In cases where conflicts arise, resolution will be effectively managed by the repository overseer. Additionally, even conflicting hypotheses can find their place within the same

database. Each hypothesis can be accommodated as a distinct class, allowing for diverse perspectives and coexistence of varying viewpoints. This inclusive approach embodies the essence of collaborative research, where differences are valued and contribute to the richness of the database.

In our pre-built database [38], we add data from the following publications: [34, 35, 36, 39], and [37]. Furthermore, we incorporate initial data from a project to conduct text mining on [40] (<https://github.com/lbjordao/PDF-scraping-Barneby>), aiming to compile a table of all characters derived from the meticulous descriptions.

Discussion

The JSON format occupies a unique position between the simplicity of a CSV spreadsheet and the formal structure of an ontology. However, its potential remains largely untapped in the realm of biodiversity research. One of the contributing factors to this underutilization is the preference of many systematic biologists for languages like R when delving into programming tasks, leading to an attachment to storing and managing data using familiar spreadsheet-like tools, such as dataframes.

While languages like R are popular choices due to their specialized statistical capabilities, they lack native support for JSON manipulation. Additionally, tools like RStudio, often favoured by biologists, might not offer the same level of efficiency and versatility found in IDEs like VS Code. This attachment to storing data in spreadsheet formats might stem from their familiarity and ease of use, even though they can limit the representation of complex data structures and relationships that JSON excels at.

The integration of JSON into the ecosystem of OOP and TS brings forth an opportunity to bridge this gap. By showcasing how JSON databases can effectively represent complex relationships and structures inherent in biological data, researchers can be encouraged to explore alternative data storage methods. This can facilitate data sharing, interoperability, and the development

of innovative analytical methods, while still addressing the attachment that some biologists may have to spreadsheets.

As the field of biodiversity research continues to expand and integrate technological advancements, embracing JSON as a fundamental data format can lead to enhanced data integrity, efficient workflows, and accelerated discoveries. By providing biologists with the tools to manage and manipulate JSON data smoothly, this integrated framework holds the potential to shift the tide from traditional spreadsheet-based data storage to more powerful, schema-free, document-based, and collaborative approaches, ultimately propelling the field forward into new realms of understanding and exploration.

Before addressing concerns regarding the necessity of implementing a database for specimen (individual) descriptions, it is crucial to clarify that one of our milestones is the establishment of a database capable of accommodating detailed specimen descriptions. However, the process of implementation requires careful consideration, due to many other complexities that arise regarding data consistency, such as the lack of definition of data types in legacy data associated with databases of biological collections. Given the intricate and varied nature of information linked to each specimen, encompassing physical traits, geographical origins, and collection histories, the development of a flexible and scalable database structure is paramount. Analysing the project's specific requirements, selecting suitable technologies, and defining a robust data schema are pivotal aspects for the success of this endeavor. Moreover, while storing specimen descriptions within TTS may not be optimal, its principles could inspire the creation of a dedicated package, such as TTS-specimens.

TTS presents an innovative approach towards the collaborative creation and development of databases. Its versatile framework not only facilitates the collective construction of robust data repositories but also invites users, including readers, to explore and pioneer new applications. The adaptability of this method transcends conventional boundaries, empowering individuals and communities to envision and implement diverse uses that resonate with their specific needs. As we delve deeper into the realm of collaborative data curation, the potential for novel, impactful applications remain vast, awaiting the creativity of each participant to shape its future trajectories.

Author contributions

Lucas Sá Barreto Jordão (Conceptualization [lead], Data curation [lead], Formal analysis [lead], Methodology [lead], Software [lead], Writing—original draft [lead], Writing—review & editing [lead]), Marli Pires Morim (Investigation [equal], Validation [lead], Visualization [equal]), José Fernando Andrade Baumgratz (Investigation [equal], Validation [lead], Visualization [equal]), Marcelo Fragomeni Simon (Investigation [equal], Validation [lead], Visualization [equal]), André Eppinghaus (Software [equal], Supervision [equal], Validation [lead], Visualization [equal]), and Vicente Calfo (Software [equal], Supervision [lead], Validation [lead], Visualization [equal])

Funding

None declared.

Conflict of interest statement. None declared.

References

1. Winston JE. *Describing Species: Practical Taxonomic Procedure for Biologists*. New York, NY: Columbia University Press, 1999, 512.
2. Morim MP, Lughadha EMN. Flora of Brazil online: can Brazil's botanists achieve their 2020 vision? *Rodriguesia* 2015;**66**: 1115–35. doi:10.1590/2175-7860201566412.
3. da Silva TSR. Species descriptions and digital environments: alternatives for accessibility of morphological data. *Rev Bras Entomol* 2017;**61**:277–81. doi:10.1016/j.rbe.2017.06.005.
4. Sarkar I, Schenk R, Norton CN. Exploring historical trends using taxonomic name metadata. *BMC Evol Biol* 2008;**8**:144. doi:10.1186/1471-2148-8-144.
5. Microsoft Corporation. *TypeScript*. 2024. <https://www.typescriptlang.org/> (29 February 2024, date last accessed).
6. JSON Schema Community. *JSON Schema*. 2024. <https://json-schema.org/> (29 February 2024, date last accessed).
7. Ashburner M, Ball CA, Blake JA et al. Gene ontology: tool for the unification of biology. *Nat Genet* 2000;**25**:25–9. doi:10.1038/75556.
8. Aleksander SA, Balhoff J, Carbon S et al. The gene ontology knowledgebase in 2023. *Genetics* 2023;**224**:10.
9. Cooper L, Walls RL, Elser J et al. The plant ontology as a tool for comparative plant anatomy and genomic analyses. *Plant Cell Physiol* 2012;**54**:e1. doi:10.1093/pcp/pcs163.
10. Walls RL, Athreya B, Cooper L et al. Ontologies as integrative tools for plant science. *Am J Bot* 2012;**99**:1263–75. doi:10.3732/ajb.1200222.
11. Perkel J. Democratic databases: science on GitHub. *Nature* 2016; **538**:127–8. doi:10.1038/538127a.
12. ECMA International. *ECMAScript Language Specification* 2023. ECMA International, 2024. <https://www.ecma-international.org/publications/standards/Ecma-262.htm> (29 February 2024, date last accessed).
13. Janicki J, Narula N, Ziegler M et al. Visualizing and interacting with large-volume biodiversity data using client-server web-mapping applications: the design and implementation of antmaps.org. *Ecol Inform* 2016;**32**:185–93. doi:10.1016/j.ecoinf.2016.02.006.
14. Lin J, Gebaly KE. The future of big data is ... JavaScript? *IEEE Internet Comput* 2016;**20**:82–8. doi:10.1109/mic.2016.109.
15. DiPierro M. The rise of JavaScript. *Comput Sci Eng* 2018;**20**:9–10. doi:10.1109/MCSE.2018.01111120.
16. Node.js Contributors. *Node.js; Node.js Foundation*. 2023. <https://nodejs.org/> (29 February 2024, date last accessed).
17. Wegner P. Concepts and paradigms of object-oriented programming. *Sigplan OopS Mess* 1990;**1**:7–87. doi:10.1145/382192.383004.
18. Sequeira RA, Olson RL, McKinion JM. Implementing generic, object-oriented models in biology. *Ecol Model* 1997;**94**:17–31. doi:10.1016/S0304-3800(96)01925-4.
19. Bedathur SJ, Haritsa JR, Sen US. The building of BODHI, a biodiversity database system. *Inf Syst* 2003;**28**:347–67. doi:10.1016/S0306-4379(02)00073-X.
20. Onkov K. Object oriented modelling in information systems based on related text data. *IFIP Adv Inf Commun Technol* 2011; **364**:212–8. doi:10.1007/978-3-642-23960-1_26.
21. Tylman W. Computer science and philosophy: did Plato foresee object-oriented programming? *Found Sci* 2016;**23**:159–72. doi:10.1007/s10699-016-9506-7.
22. Chai H, Wu G, Zhao Y. *A Document-Based Data Warehousing Approach for Large Scale Data Mining, in Pervasive Computing and the Networked World*. Berlin Heidelberg, Germany: Springer, 2013, 69–81. doi:10.1007/978-3-642-37015-1_7.

23. Karnitis G, Arnicans G. Migration of relational database to document-oriented database: structure denormalization and data transformation. *7th International Conference on Computational Intelligence, Communication Systems and Networks*. Piscataway, NJ: IEEE, 2015, 113–8. doi:[10.1109/CICSyN.2015.30](https://doi.org/10.1109/CICSyN.2015.30).
24. Chickerur S, Goudar A, Kinnerkar A. Comparison of relational database with document-oriented database (MongoDB) for big data applications. *8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*. Piscataway, NJ: IEEE, 2015, 41–7. doi:[10.1109/ASEA.2015.19](https://doi.org/10.1109/ASEA.2015.19).
25. Olivera HV, de Holanda MT, Guimarães V et al. Data modeling for NoSQL document-oriented databases. *Symposium on information management and big data*. 2015. <https://api.semanticscholar.org/CorpusID:15589232>.
26. Mason RT. NoSQL databases and data modeling for a document-oriented NoSQL database. In: *Proceedings of Informing Science & IT Education Conference (InSITE)*. Denver, CO: College of Computer & Information Sciences, Regis University, 2015, 259–68. doi:[10.28945/2245](https://doi.org/10.28945/2245).
27. Baazizi MA, Colazzo D, Ghelli G et al. Schemas and types for JSON data: from theory to practice. In: *Proceedings of the 2019 International Conference on Management of Data*. New York, NY: ACM, 2019, 2060–3. doi:[10.1145/3299869.3314032](https://doi.org/10.1145/3299869.3314032).
28. Spinellis D. Code documentation. *IEEE Softw* 2010;**27**:18–9. doi:[10.1109/ms.2010.95](https://doi.org/10.1109/ms.2010.95).
29. Rai S, Belwal RC, Gupta A. A review on source code documentation. *ACM Trans Intell Syst Technol* 2022;**13**:1–44. doi:[10.1145/3519312](https://doi.org/10.1145/3519312).
30. Warren A, Patterson DJ, Dunthorn M et al. Beyond the “Code”: a guide to the description and documentation of biodiversity in ciliated protists (Alveolata, Ciliophora). *J Eukaryot Microbiol* 2017;**64**:539–54. doi:[10.1111/jeu.12391](https://doi.org/10.1111/jeu.12391).
31. Blischak JD, Davenport ER, Wilson G. A quick introduction to version control with git and GitHub. *PLoS Comput Biol* 2016;**12**: e1004668. doi:[10.1371/journal.pcbi.1004668](https://doi.org/10.1371/journal.pcbi.1004668).
32. Perez-Riverol Y, Gatto L, Wang R et al. Ten simple rules for taking advantage of git and GitHub. *PLoS Comput Biol* 2016;**12**: e1004947. doi:[10.1371/journal.pcbi.1004947](https://doi.org/10.1371/journal.pcbi.1004947).
33. Crystal-Ornelas R, Varadharajan C, Bond-Lamberty B et al. A guide to using GitHub for developing and versioning data standards and reporting formats. *Earth Space Sci* 2021;**8**: e2021EA001797. doi:[10.1029/2021ea001797](https://doi.org/10.1029/2021ea001797).
34. Jordão LSB, Morim MP, Baumgratz JFA. A new species of *Mimosa* (Leguminosae) from Brazil. *Phytotaxa* 2014;**184**:131. doi:[10.11646/phytotaxa.184.3.2](https://doi.org/10.11646/phytotaxa.184.3.2).
35. Jordão LSB, Morim MP, Baumgratz JFA et al. A new species of *Mimosa* (Leguminosae) endemic to the Brazilian cerrado. *Phytotaxa* 2017;**312**:237. doi:[10.11646/phytotaxa.312.2.6](https://doi.org/10.11646/phytotaxa.312.2.6).
36. Jordão LSB, Morim MP, Baumgratz JFA. Toward a census of *Mimosa* (Leguminosae) in the Atlantic domain, southeastern Brazil. *Syst Bot* 2018;**43**:162–97. doi:[10.1600/036364418x696905](https://doi.org/10.1600/036364418x696905).
37. Jordão LSB, Morim MP, Baumgratz JFA. Trichomes in *Mimosa* (Leguminosae): towards a characterization and a terminology standardization. *Flora* 2020;**272**:151702. doi:[10.1016/j.flora.2020.151702](https://doi.org/10.1016/j.flora.2020.151702).
38. Jordão LSB. *lsbjordao/TTS-Mimosa: first release (version v1)*. Zenodo. 2024. doi:[10.5281/zenodo.10671076](https://doi.org/10.5281/zenodo.10671076).
39. Vieira LG, Nogueira RM, Costa EC et al. Insect galls in rupestrian field and cerrado stricto sensu vegetation in Caetité, Bahia, Brazil. *Biota Neotrop* 2018;**18**:e20170402. doi:[10.1590/1676-0611-bn-2017-0402](https://doi.org/10.1590/1676-0611-bn-2017-0402).
40. Barneby RC. *Sensitivae Censitae: A Description of the Genus Mimosa Linnaeus (Mimosaceae) in the New World*. Bronx, NY: New York Botanical Garden, 1991, 835.