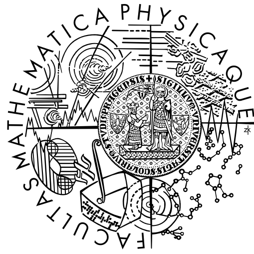


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Kseniia Popova

Type Provider for the UniProt Knowledge Base

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D
Study programme: Computer Science with specialisation in Programming and Software Development(IPP2)

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, Mgr. Tomáš Petříček, Ph.D., for his guidance throughout the development of this thesis and for sharing his unique expertise.

I am deeply grateful to Carlin Bowman Kerbage, whose support and encouragement carried me through the most challenging phases of this journey.

Title: Type Provider for the UniProt Knowledge Base

Author: Kseniia Popova

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D, Department of Distributed and Dependable Systems

Abstract: The UniProt knowledge base is an essential, expertly curated protein database. UniProt can be accessed programmatically using clients in langauges such as Java and Python, but it lacks a suitable statically-typed interface for exploratory programming. This thesis presents a domain-specific F# type provider that integrates UniProt's schema into the type system, enabling compile-time validation and auto-completion for intuitive data exploration. By abstracting UniProt's REST API into a fluent, type-safe interface, the solution reduces boilerplate and emphasises domain semantics. It supports efficient exploration of protein entries through on-demand data fetching. The type provider simplifies access for researchers, combining static typing with interactive exploration to accelerate data-driven discovery in the field of bioinformatics.

Keywords: type providers, uniport, bioinformatics

Název práce: Type Provider for the UniProt Knowledge Base

Autor: Kseniia Popova

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D, Katedra distribuovaných a spolehlivých systémů

Abstrakt: UniProt knowledge base je široce užívána, odborně zpracovaná databáze proteinů. K databázi UniProt lze přistupovat programově pomocí klientů v jazycích, jako jsou Java a Python, ale chybí jí vhodné staticky typované rozhraní pro programování zaměřené na průzkum dat. Tato práce představuje doménově specifický type provider v jazyce F#, který integruje schéma UniProt do typového systému a umožňuje validaci při kompilaci a automatické doplňování kódu pro intuitivní průzkum dat. Abstrahováním rozhraní REST API UniProt do fluentního, typově bezpečného rozhraní klade prezentované řešení důraz na doménovou sémantiku. Podporuje efektivní průzkum proteinových záznamů prostřednictvím načítání dat na vyžádání. Type provider zjednodušuje přístup pro výzkumné pracovníky a kombinuje statické typování s interaktivním průzkumem s cílem urychlit objevování dat v oblasti bioinformatiky.

Klíčová slova: type providers, uniport, bioinformatics

Contents

Introduction **7**

1 Background **9**
1.1 Type Provider in F# . . . 9
1.2 UniProt . . . 10
1.3 Utilisation in Bioinformatics . . . 12
1.3.1 Advantages of a UniProt-specific Type Provider . . . 12
1.4 Requirements . . . 12

2 Tools Overview **14**
2.1 F# . . . 14
2.2 Type Provider SDK . . . 16
2.2.1 Type Provider Architecture . . . 16
2.2.2 Usage of the Type Provider SDK . . . 16
2.3 Ionide . . . 17

3 Work Description **18**
3.1 Provided Types . . . 18
3.2 Types Generation . . . 19
3.2.1 Delayed Members . . . 20
3.2.2 Recursive Definition . . . 20
3.3 Query Configuration . . . 20
3.4 Results Caching . . . 22
3.5 Design Considerations . . . 22
3.5.1 Types . . . 23
3.5.2 Schema Change . . . 24
3.5.3 Schema Variability . . . 24
3.5.4 Lazy Generation . . . 24
3.5.5 Asynchrony . . . 25
3.5.6 Large Results Set . . . 25
3.6 Encountered Obstacles . . . 26
3.6.1 Troubleshooting Techniques . . . 27

4 Usage **28**
4.1 Getting Started . . . 28
4.1.1 Installation From Sources . . . 28
4.1.2 NuGet . . . 28
4.2 Interface . . . 28
4.3 Code Samples . . . 29
4.4 Interactive Notebooks . . . 34

5 Evaluation **35**
5.1 JSON Type Provider . . . 35
5.2 REST API Clients . . . 36
5.3 UniProtJAPI . . . 36

5.4 BioServices . . . 37
5.5 Biopython . . . 37
5.6 Summary . . . 38

Conclusion **39**

Bibliography **40**

List of Figures **42**

A Attachments **43**
A.1 Overview of the UniProtProvider Implementation . . . 43
A.2 Protein Entry JSON Example . . . 44

Introduction

Working with structured data is an essential part of software development. Many applications require using data from different sources, such as databases and web APIs. Handling this data typically requires mapping it from its external format to the type system of the programming language being used. This means ensuring that data from external sources aligns with, and therefore ought to be implemented, according to the language's type structure.

One of the solutions to this challenge is a mechanism called Type Provider in the F# programming language. [1] Type providers dynamically create types at compile (design) time. In addition, types can be generated on demand, only when required by the user-written code, allowing integration of Internet-scale data sources. This makes it easier to connect to external data and use it in code, improving workflow and providing better tools for development. This approach can improve the way developers manage and interact with external data in a wide range of applications.

Let us demonstrate the differences in approaches between a proposed type provider (called UniProtProvider) and one of the available libraries for access to the UniProt data. Suppose that the user is interested in information about human insulin, using the UniProt BioPython library they can write the following:

Listing 1 BioPython example

```
from Bio import UniProt

query = "insulin AND (reviewed:true) AND (organism_name:human)"
results =  UniProt.search(query, batch_size=50)[:50]
for result in results:
    print(f"ID: {result['primaryAccession']}, \
        Name: {result['proteinDescription'] \
        ['recommendedName']['fullName']['value']}")
```

The first thing to note is that the query must be constructed manually, requiring prior knowledge of the parameters and their syntax, which must be sourced from external documentation. Next, the result is returned as a list of dictionaries, forcing the user to reference keys through string literals. Discovering available keys requires executing *print(dir(result))* on individual results. The information about the retrieved entries is not presented to the user until it is directly accessed and the program is executed, printing the values.

In contrast, using the UniProtProvider, the user can write the following:

Listing 2 UniProtProvider example

```
UniProtProvider.ByKeyWord<"insulin">()
    .ByOrganism<"human">()
    .Reviewed().``Insulin-degrading enzyme (IDE_HUMAN)``
```

Here, the user benefits from a declarative, autocompletion-driven interface. Typing "." triggers the type provider to dynamically generate a list of results, containing information about the name and accession. This helps to quickly navigate through the results and find the specific protein. After selecting the result from the list, the user can obtain the corresponding statically typed protein entry. Its hierarchy can be further explored and traversed through property access.

This workflow does not require any knowledge of query parameters and property names.

Building on this approach, this thesis presents the design and implementation of a type provider for the UniProt knowledge base. The developed solution enables mapping from the online knowledge base to the F# type system, exposing individual protein entries through member access and providing various filtering options for result refinement.

Thesis Outline

The thesis is organised into the following chapters:

- Chapter 1 presents the problem domain and motivations behind this work, alongside the core requirements guiding the implementation.
- Chapter 2 overviews the tools used for the development.
- Chapter 3 provides a detailed guide through the implementation. It includes an explanation of the type provider's internals, explores the type generation mechanism in depth, outlines the rationale behind key design decisions and discusses challenges encountered during the type provider's development.
- Chapter 4 provides a practical guide to using the UniProtProvider, including code samples.
- Chapter 5 evaluates existing alternatives and assesses the feasibility and advantages of the proposed type provider.

1 Background

This chapter explains the motivation behind this thesis and provides an overview of the concepts used in this work. It further details the specific scenarios that the proposed type provider is designed to support and concludes with a structured list of formal requirements.

1.1 Type Provider in F#

An F# type provider is a mechanism that makes it possible to provide a statically-typed interface for accessing external data sources, resources and services from the F# programming language. It supports providing types, properties and methods for use in a program. [1]

Provided types are dynamically generated at design-time, a phase that includes both interactive code editing, where integrated development environments execute background compiler instances to validate code incrementally, and explicit compilation invocations. During this phase, type providers are activated by the compiler to dynamically check types based on developer interactions and external data sources.

Types generated by a type provider are determined by input parameters. This input can be, for example, a sample schema or a direct URL to an external service. The type provider dynamically generates these types when they are explicitly referenced in the program. This approach allows on-demand, direct integration of Internet-scale data sources while maintaining a strongly typed representation in the code.

As an illustrative example, let us explore an F# Type Provider for JSON that generates types inferred from a JSON schema. [2]

Listing 1.1 JSON type provider

```
type UniProtKB = JsonProvider<"""
                             {
                               "primaryAccession": "P14735",
                               "uniProtkbId": "IDE_HUMAN"
                             }
                              """>
let insulin = UniProtKB.Parse("""
                             {
                               "primaryAccession": "P06213",
                               "uniProtkbId": "INSR_HUMAN"
                             }
                               """)
insulin.PrimaryAccession
insulin.UniProtkbId
```

The *JsonProvider<...>* accepts a static parameter containing a representative JSON sample. During compilation, this sample is analysed to infer a schema, which the type provider uses to generate corresponding types and properties. In the provided example, the UniProtKB type is created with properties like *PrimaryAccession* and *UniProtkbId*, which become immediately accessible in the editor.

These types, however, are not present in the compiled code — a characteristic of erased types, which are replaced with simpler runtime representations. This leads us to a key distinction in type provider implementations:

Generated type providers generate actual .NET types that are compiled into the assembly. These types persist at runtime and are available for reflection and API exposure. *Erased* type providers provide design-time types for IntelliSense and type checking, but erase them at compile-time. It is possible to replace the erased types with runtime representations, which affects how they are represented in compiled .NET code.

In most cases, including the provided example, type providers make use of erased types when working with large information spaces, as it reduces the size of generated code. [1]

In our example, the generated type *UniProt* is erased to a runtime type *JsonValue*. The property access is erased to method calls of the JSON type provider runtime. [3] After the *Parse(...)* is executed, *insulin.PrimaryAccession* resolves to the 'P06213' string value.

1.2 UniProt

"UniProt is the world's leading high-quality, comprehensive and freely accessible resource of protein sequence and functional information". [4] It consists of UniProtKB - the protein knowledge base, Proteomes - the protein sets expressed by organisms, UniRef - the protein clusters, UniParc - the archive of sequences and several supporting datasets, such as Taxonomy, Diseases and Subcellular locations.

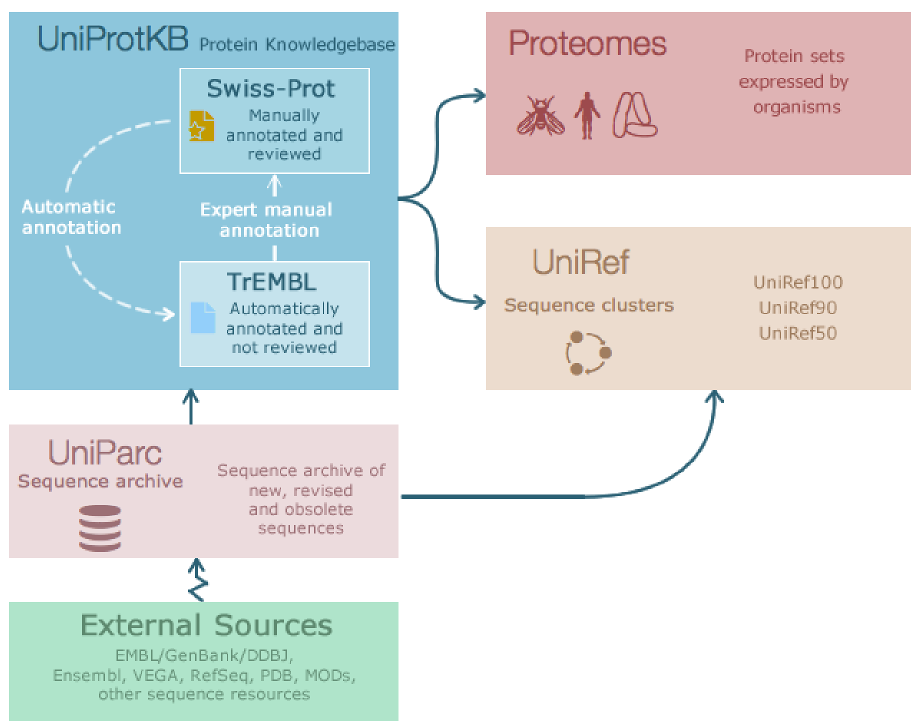


Figure 1.1 UniProt data overview

"The UniProt Knowledgebase, and in particular UniProtKB/Swiss-Prot, is used to access functional information on proteins. Every UniProtKB entry contains the amino acid sequence, protein name or description, taxonomic data and citation information." [5] In addition, the UniProtKB entry contains many other annotations, the full list of which is available in the UniProtKB manual. Individual entries may not contain all the properties defined in the schema. The Attachment A.2 contains an example of data from one of the protein entries. The following is an example of some of the protein entry information available on the UniProt website:

Figure 1.2 Protein entry

UniProt provides several APIs for programmatically querying and accessing its data. [6] Throughout the available formats are JSON and TSV. In this thesis, we focus on providing access to protein entries (253,206,170 results as of 05/04/2025) with the support of taxonomy data (2,998,013 results).

1.3 Utilisation in Bioinformatics

Given the amount and specificity of data, the UniProt knowledge base is a good candidate for implementing a type provider. It will enable intuitive exploration of UniProt data directly from a code editor and serve as a valuable extension to the ecosystem of existing bioinformatic type providers, such as BioProviders [7] - a package designed to simplify programmatic access to genomic resources like GenBank.

A user could begin typing *UniProtKB.ByKeyword<"human">* and receive auto-completion and suggestions at each step, guided by the type provider's compile-time generation. For non-programmers, this interactivity lowers the barrier to programmatic data access. The goal is to provide the ability to retrieve, examine and access individual properties of protein entries, complementing the existing interfaces provided by UniProt that offer advanced search options and data downloading in batches.

1.3.1 Advantages of a UniProt-specific Type Provider

Integration of the UniProt schema: Implementing a type provider that is compatible with the UniProt-specific data schema brings data types that reflect its structure. This integration facilitates direct access to fields and nested data with IntelliSense code completion, reducing the need for manual reference to external documentation.

Statically typed data access: The API enables direct compile-time resolution of biological data through static types, wherein developers can reference individual proteins or taxonomic entries from the database as first-class language elements. This feature remains unique to the type provider mechanism compared with alternatives.

Type safety: Compile-time validation of field names and data types, preventing runtime errors due to typos and parameter mismatches during data retrieval and processing.

Efficiency of exploration: For large queries (e.g. retrieving all human proteins), the type provider takes advantage of on-demand generation of results. In addition, local caching mechanisms minimise redundant network calls, improving the performance of repeated queries.

1.4 Requirements

To summarise, the proposed type provider implementation must meet the following requirements:

1. Provide static access to individual protein and taxonomy entries available from UniProt.

2. Support multiple chained filtering operations to iteratively narrow the result set.
3. Guide the user through available results and filtering options directly in the editor.

At the same time, the following are not considered requirements for this work:

1. Download and process large result sets.
2. Dynamic generation of end types (Protein and Taxonomy) based on available properties.

2 Tools Overview

This chapter introduces the technologies and tools enabling the development of a type provider for the UniProt knowledgebase.

2.1 F#

F# is a universal, strongly typed programming language in the .NET ecosystem. F# supports functional, object-oriented and imperative programming models. It is open-source, cross-platform and interoperable with C#, Python and other languages. [8] F# was suitable for introducing a type provider mechanism because of its core features:

Type inference and static typing

F# infers types at compile-time while maintaining strict static typing. This is critical for type providers that dynamically generate types based on external schemas while ensuring compile-time safety.

Listing 2.1 F# dynamic inference example

```
let getLength input =
    input |> string |> String.length

getLength 42
getLength "Hello"
```

The compiler infers a type of *input* as a generic type *'a* because the conversion to string can be applied to any type. Therefore, the type of the *getLength* function is *'a* $\rightarrow$ *int*. In the first example, 42 is converted to string and its length is returned.

Metaprogramming capabilities

F# code quotations provide a mechanism for generating code at runtime. The type provider leverages this capability to define an expression for getter methods for constructors and generated properties. Reflection is used to add member (methods, types, properties) metadata to a type definition.

Listing 2.2 F# quotation example

```
let expr = <@ 1 + 2 @>
```

In this example *expr* will be compiled into an object that represents an expression. It is represented as *Call (None, op_Addition, [Value (1), Value (2)])*, where the *Call* constructor represents a method call of a static (instance object is None) addition operation applied to two literals. Developers can programmatically traverse this expression tree to analyse its structure, modify components or convert it to alternative formats.

While the type provider implementation does not utilise this tool, the quotation can be evaluated using the *QuotationEvaluator*:

Listing 2.3 F# quotation evaluation example

```
open FSharp.Quotations.Evaluator
expr |> QuotationEvaluator.Evaluate
```

.NET ecosystem

F# integrates with existing .NET tooling, including development environments, package managers and .NET libraries.

Listing 2.4 F# .NET integration example

```
#r "nuget: Newtonsoft.Json, 13.0.1"

open System.Net.Http
open Newtonsoft.Json

let httpClient = new HttpClient()
httpClient.DefaultRequestHeaders
    .Add("User-Agent", "F# Demo/1.0")

type RepoStars = {
    stargazers_count: int
}

let getRepoStars owner repo = async {
    let! response =
       httpClient.GetStringAsync
           ($"https://api.github.com/repos/{owner}/{repo}")
       |> Async.AwaitTask
    return JsonConvert
        .DeserializeObject<RepoStars>(response)
}

[ ("dotnet", "runtime")
  ("fsharp", "fsharp") ]
|> List.map (fun (owner, repo) -> getRepoStars owner repo)
|> Async.Parallel
|> Async.RunSynchronously
|> Array.iter (fun stars ->
    printfn $"{stars.stargazers_count} stars")
```

This example demonstrates F#'s integration with the .NET ecosystem by combining cross-platform .NET tooling (via F# interactive), NuGet package management (Newtonsoft.Json) and core .NET libraries (HttpClient) to build an API client. In the type provider implementation, these libraries are useful for fetching JSON data and deserialising it into F#'s strongly typed representation.

Functional programming paradigm

F# emphasises immutability and declarative patterns, producing code that is concise and less prone to side effects:

Listing 2.5 F# declarative pipline example

```
let numbers = [1..10]

let evenSquaresSum =
    numbers
    |> List.filter (fun x -> x % 2 = 0)
    |> List.map (fun x -> x * x)
    |> List.sum
```

This code sample demonstrates F#'s functional capabilities through an immutable, declarative pipeline. By chaining with the '|>' operator, it transforms data without mutable variables or explicit loops, focusing on what to compute rather than how.

2.2 Type Provider SDK

The Type Provider Software Development Kit is essential for implementing a Type Provider. It helps to generate the dependencies and project structure required for development. [9] In this way, the Type Provider SDK streamlines development by abstracting the infrastructure, allowing the focus to be on domain-specific logic.

2.2.1 Type Provider Architecture

The core components of a Type Provider include:

- ProvidedTypes API: contains interfaces for generating types, methods and properties. The source files are obtained via the *Paket* dependency manager.
- Type Provider Design Time Component (TPDTC) is the DLL that is loaded into host tools, most notably the F# compiler and the service that provides information to F# editors. This component references the *ProvidedTypes.fs/fsi* files from the type provider SDK.
- Type Provider Runtime Component (TPRTC), points to the design-time DLL. This component is referenced by the user program the same way as any other library. TPRTC provides the runtime representation for erased types and the implementation of their functionality.
- Host tool: any tool that hosts *FSharp.Compiler.Service.dll* in the user application. For example, the *devenv* process in the Microsoft Visual Studio IDE or the *FsAutoComplete* service used by Visual Studio Code.

2.2.2 Usage of the Type Provider SDK

The creation of a functional type provider is easy using .NET tools:

Listing 2.6 Creating a type provider from a template

```
dotnet new -i FSharp.TypeProviders.Templates
dotnet new typeprovider -n LemonadeProvider -lang F#
```

This creates a template for a type provider implementation that can be further modified [10]. The template contains two example type providers: *BasicGenerativeProvider* and *BasicErasingProvider*. The type provider definition is annotated with the *[<TypeProvider>]* attribute. In addition, the assembly declares its type provider capability via an assembly attribute. This should be placed in a RunTime DLL and point to the DesignTime DLL if they are separated.

Listing 2.7 Type provider entry point

```
[<assembly: TypeProviderAssembly("DesignTime")>]
do()
```

The type provider must implement the *ITypeProvider* interface. The easiest way to do this is to inherit from the *TypeProviderForNamespaces* contained in *ProvidedTypes.fs*.

All defined types are erased by default; generated types should be explicitly marked as such by passing the *isErased = false* parameter to the type definition.

Next, the namespace for the provided types should be defined. The assembly itself should also be obtained. Types, their properties and members can be created using the TypeProvider interface. Finally, the types are added as a provided namespace.

Listing 2.8 Implementing a type provider

```
let ns = "MyNamespace"
let asm = Assembly.GetExecutingAssembly()
...
do this.AddNamespace(ns, createTypes())
```

2.3 Ionide

Ionide is a suite of Visual Studio Code extensions for cross-platform F# development. [11]

Features critical to type provider development include:

- IntelliSense for provided types: Autocompletion for generated UniProt properties (e.g., protein.sequence.length), enabled by the Ionide's support for the F# Compiler Service.
- Error highlighting: Early detection of wrong parameters and incompatible types.
- Interactive development (fsi): Allows exploratory coding with live UniProt data.
- Integration with .NET tools: Built-in support for Paket (dependency management) and FAKE (building tool).

3 Work Description

This chapter provides a detailed description of the design and implementation of the UniProt type provider, focusing on its core mechanisms for generating types, interfacing with an external API and overcoming technical challenges. The work prioritises type safety, usability and ensuring the seamless integration of UniProt data into F#'s static type system.

3.1 Provided Types

The UniProtProvider has two basic workflows to progressively refine queries:

Listing 3.1 By keyword

```
ByKeyWord<"keratin">().ByOrganism<"human">()
```

This invocation order is advantageous in scenarios where the user is interested in finding specific details about a particular protein within a provided organism. The workflow first retrieves all entries matching the keyword, then filters the results to entries that are associated with the specified organism, as shown in Figure 3.1.

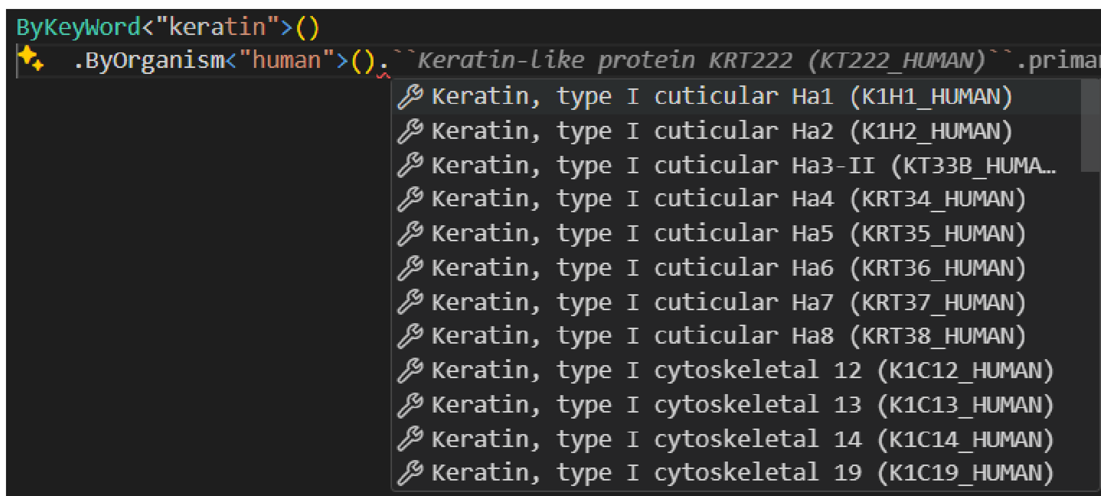


Figure 3.1 ByKeyword results

For use cases where all the proteins associated with a particular organism need to be identified, the reverse order can be beneficial:

Listing 3.2 By organism

```
ByOrganism<"human">().``Homo sapiens (9606)``.FindRelated()
```

First, the workflow queries UniProt for all entries matching the specified organism name. For the selected organism, it returns protein entries related to that organism, as shown in Figure 3.2.

In both workflows, if the result set exceeds 50 entries, the *"More..."* property is appended to facilitate navigation through the results. Accessing this property triggers the retrieval and display of the subsequent batch of results (Section 3.5.6).

```
ByOrganism<"human">()
    .``Homo sapiens (9606)``
    .FindRelated().``Clarin-2 (CLRN2_HUMAN)``
                   Aldo-keto reductase family 1 member B15 (AK1BF…
                   Alkyldihydroxyacetonephosphate synthase, perox…
                   Bridge-like lipid transfer protein family memb…
                   Ciliated left-right organizer metallopeptidase…
                   Clarin-2 (CLRN2_HUMAN)
                   Coiled-coil domain-containing protein 66 (CCD6…
                   Cysteine-rich tail protein 1 (CRTP1_HUMAN)
                   Deoxyribonuclease-2-alpha (DNS2A_HUMAN)
                   E3 ubiquitin-protein ligase RNF103 (RN103_HUMA…
                   Espin (ESPN_HUMAN)
                   FERM domain-containing protein 3 (FRMD3_HUMAN)
                   Galectin-8 (LEG8_HUMAN)
```

Figure 3.2 ByOrganism results

For protein entries, users can further refine their search by applying additional filters. An overview of filtering capabilities is provided in later sections.

3.2 Types Generation

The type generation mechanism employs a two-stage execution model [12], combining design-time code generation with runtime execution. To illustrate this process, we examine a representative example from the UniProtProvider implementation:

Listing 3.3 Properties generation

```
let getOrganismProps (props: array<TaxonomyIncomplete>)() =
    props
    |> Array.map (fun i ->
        let name = $"{i.scientificName} ({i.taxonId})"
        let value = i.taxonId
        ProvidedProperty(
            propertyName=name,
            propertyType = typeof<Taxonomy>,
            getterCode = (fun _ ->
                <@@ getOrganismById value
                    |> Async.RunSynchronously @@>)
        )
    )
    |> Array.toList
```

This function is responsible for the generation of properties for a set of taxonomy results. Given the array of taxonomy metadata, containing information about the name and accession, it returns a list of *ProvidedProperty* objects. The curried function signature – denoted by the empty parameter set '()' – enables partial function application, delaying computation until the second argument is supplied. This enables the type provider's ability to add members in a delayed

fashion, which is described in detail in the next subsection.

Property definitions are generated at design-time, upon accessing the *"More..."* property within the editor. The *getterCode* function, embedded within code quotations, is executed at runtime. This quotation invokes the *getOrganismById* function from the runtime component with the unique taxonomy ID as a parameter and retrieves the corresponding entry of a *Taxonomy* type.

3.2.1 Delayed Members

When processing result sets containing multiple entries, types are created lazily (on demand) to optimise performance and resource usage by avoiding unnecessary overhead for generating unused entries.

- On the first request, the type provider fetches a set of UniProt entries containing only name and accession information to present to a user.
- Properties containing complete protein or taxonomy entries are generated only when referenced in code.

The implementation leverages the type provider's native support for delayed member generation through the *AddMembersDelayed* method:

Listing 3.4 Delayed members

```
next.AddMembersDelayed (getOrganismProperties results)
```

In the UniProtProvider design, the logic for generating types, methods and properties is encapsulated within delayed functions, which are used as building blocks for composing a type provider. For each provided type, the relevant functions return members that are added to that type and are only executed at compile-time if explicitly accessed by a user. Internally, this process is performed as follows: member metadata is added to a type definition, but the computation is queued until it is required by the compilation context. [13]

3.2.2 Recursive Definition

A notable implication of the delayed execution model is the support for recursively defined members. For instance, the *"More..."* property is implemented in this manner: once accessed, the type provider triggers the generation of subsequent properties, along with an additional *"More..."* property to facilitate further iterations, as shown in Figure 3.3. This mechanism allows for a potentially infinite chain of invocations, although in practice the result set is limited by the number of entries in the UniProt dataset.

Filtering methods employ mutual recursion, propagating unapplied filters to subsequent type definitions. This permits arbitrary ordering of filters.

3.3 Query Configuration

The UniProtProvider encapsulates query construction logic, abstracting low-level implementation details from end users to streamline interactions with the UniProt API. It offers an interface for filtering results using criteria such as unique

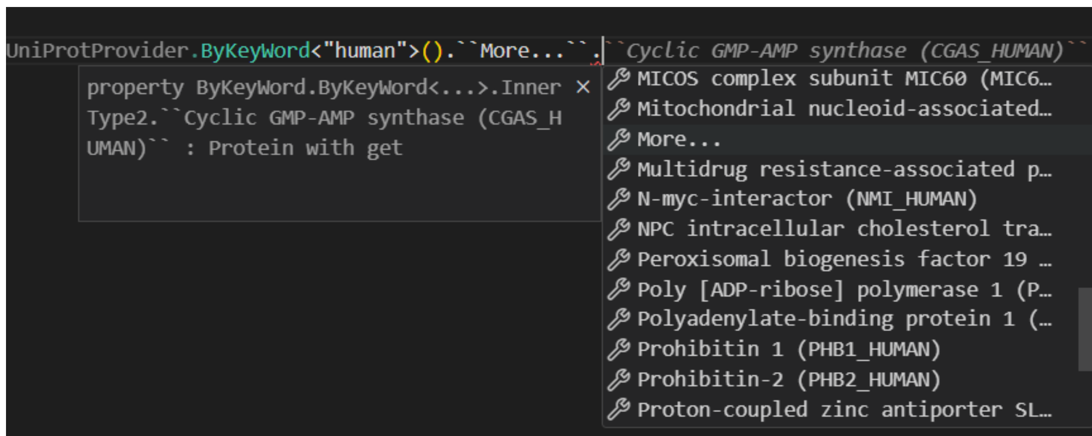


Figure 3.3 "More..." property

identifiers, organism names and protein names, with these filtering capabilities integrated into the type provider's design. This tight integration ensures type safety and autocompletion features while preserving the declarative workflow.

The Figure 3.4 illustrates the workflow for progressively constructing queries for the *ByKeyWord* type provider. Rounded rectangles denote function applications, parallelograms represent parameter inputs (e.g. filtering criteria) and rectangles correspond to property selections. The process begins by initialising the type provider with a mandatory string parameter representing a protein name. Applying a filter sets the corresponding query parameter, and the diagram highlights that filters can be applied in any order, with an updated result set displayed at each step. The *"More..."* property configures a parameter responsible for pagination. At every stage, users may apply any remaining unapplied filter, generate the next batch of results (if available), or select a specific Protein entry.

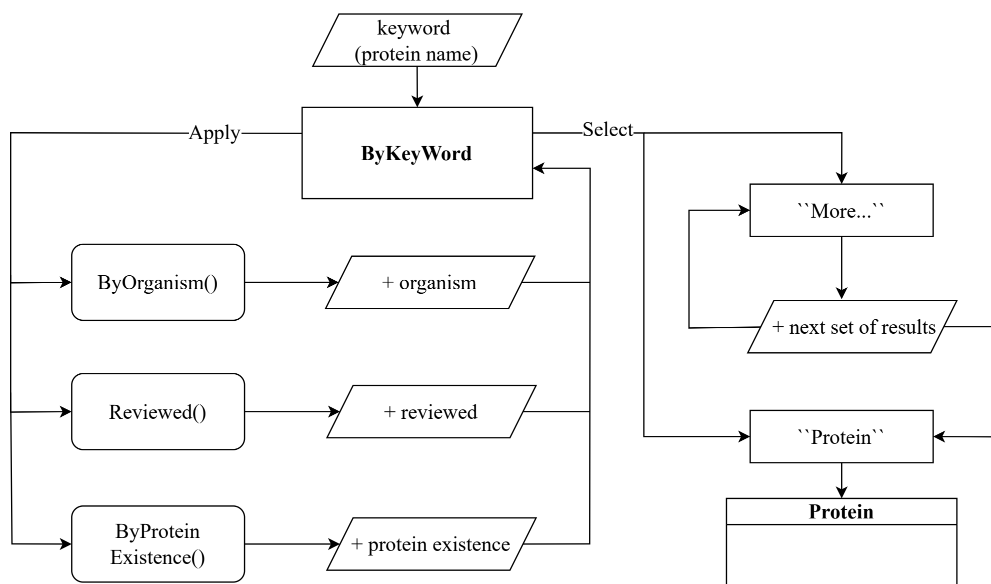


Figure 3.4 Parameters configuration

Once a query is configured, it is used to request and fetch results in JSON format. The JSON response is then mapped to type-safe data structures. By combining abstraction, pagination and on-demand generation, the UniProtProvider balances performance with usability for data exploration tasks.

3.4 Results Caching

To minimise redundant API requests and improve efficiency, the type provider implements a disk-based caching mechanism that persistently stores compressed JSON responses retrieved from UniProt in a temporary directory. This approach is efficient when working with the same entries repeatedly, saving API calls for already available data. In addition, all read/write operations use non-blocking asynchronous methods, allowing overlapping of disk I/O with other operations. The implementation follows these steps:

- Each URL of the API request is hashed using *System.Data.HashFunction.CityHash* to generate a unique filename. This ensures deterministic naming, collision-free storage and consistent retrieval of cached entries.
- The UniProt API supports GZip-compressed responses, which reduce file sizes by approximately 80% compared to uncompressed JSON. The type provider utilises this feature, directly storing the compressed data to minimise disk space consumption.
- Prior to initiating an API request, the provider checks the cache directory for an existing entry matching the hashed URL. If found, the compressed JSON is decompressed and returned without contacting UniProt's servers.

While this approach significantly reduces API traffic and accelerates repeated workflows, it introduces two limitations. First, cached data does not automatically reflect updates to UniProt entries post-retrieval. Users must manually delete the temporary directory to ensure subsequent requests fetch the latest revisions. Second, prolonged use without cache clearance may accumulate disk storage overhead.

To keep data up-to-date and reduce data accumulation, users are advised to delete the temporary directory containing cached results upon concluding their work with the type provider. Upon the next compilation, the provider will regenerate the required data by querying UniProt's API anew. Once retrieved, the data persists in a readily accessible state, enabling efficient manipulation with minimal computational overhead. This temporary performance cost is counterbalanced by the assurance of up-to-date records, as manual cache deletion remains the sole method to override locally stored data in favour of fresh revisions.

3.5 Design Considerations

Type providers mechanism inherently supports flexibility through its dual capacity to integrate both runtime execution and design-time code generation. This hybrid approach enables developers to strategically allocate components

between dynamic data retrieval and static schema definition. Such decisions are context-dependent, requiring careful evaluation of factors including data volatility, performance requirements, maintainability and usability. This section outlines architectural choices made during the development of the UniProtProvider, including their advantages and limitations.

3.5.1 Types

The UniProtProvider employs design-time type generation to construct result lists, represented as properties within the provider's interface. This approach extends to static methods such as *ByOrganism<...>()* and *FindRelated()*, which dynamically generate intermediate types to guide users through query refinement and exploration of nested data.

Figure 3.5 demonstrates the hierarchy of the generated types. The *ByKeyWord* type provider presents an initial list of up to fifty results alongside an *“More...“‘* property for loading subsequent entries, accompanied by filtering capabilities. Upon application of a filter, the new list of results is generated along with yet unused filters. The *ByOrganism* type provider generates a list of up to fifty organism results, similarly featuring the *“More...“* property. Following organism selection, users may either retrieve the corresponding Taxonomy instance by selecting the *“Homo sapiens (9606)”* property or obtain a list of the protein entries associated with this organism. For the list of proteins, the equivalent filtering options are available to those present in the *ByKeyWord* provider, with the exception of the *ByOrganism<...>()* filter as the organism is already specified.

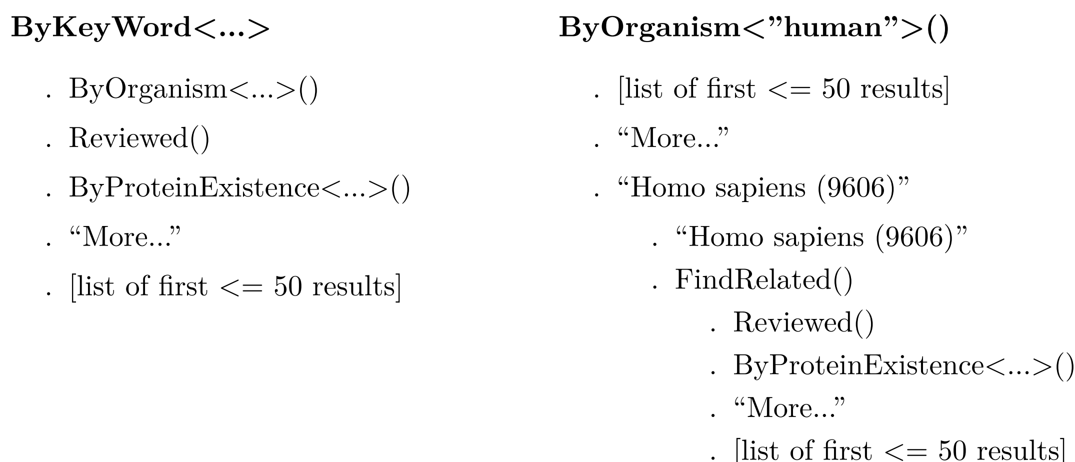


Figure 3.5 Overview of generated types

The target types — *Protein* and *Taxonomy* — are implemented as immutable F# record types based on the available UniProt schema. When a user selects a specific entry, the corresponding JSON payload is retrieved and deserialised into the appropriate record type.

The choice to employ static record types offers substantial benefits over dynamic or generative typing strategies that would represent every query as a separate type. Records enable compile-time validation of data transformations, comparisons and assignments. Values can be stored in standard collections (e.g., arrays, lists) or passed between functions, allowing integration with various applications.

3.5.2 Schema Change

The UniProtProvider models protein and taxonomy entries as F# record types, mirroring the structure defined by UniProt's JSON schema. This implementation introduces a dependency on UniProt's schema stability. Should the schema evolve (e.g., through field additions or renamings), type mismatches between the provider's static records and the updated JSON responses would result in errors and necessitate manual adjustments to the library.

This dependency, however, is mitigated by UniProt's role as a central hub for collecting the protein information. Schema modifications are inherently unlikely, as abrupt changes would disrupt the tools reliant on its consistency. The current schema's maturity further reduces this risk, having been optimised over years. However, it is not explicitly stated that such change is impossible.

Thus, while the schema dependency represents a theoretical limitation, the practical trade-off favours static typing advantages.

3.5.3 Schema Variability

Biological data exhibits inherent variability: entries within UniProt may omit certain schema fields. To mitigate deserialisation failures arising from missing data, the UniProtProvider designates all record fields — except critical mandatory identifiers — as optional. This ensures that partial or incomplete entries can still be parsed without errors. Optional fields are represented using F#'s *Option* type, requiring pattern matching or explicit handling for potential absences using *IsSome* property. This necessity stems from the constraints of a strongly typed system and static record type definitions.

In the following example, the presence of the *nect2_human_genes* value is safely handled via the *Option.toArray* conversion, and the individual *geneName* values are accessed using pattern matching:

Listing 3.5 Optional fields

```
let nect2_human_genes =
    UniProtProvider.ById<"Q92692">()
        .``Nectin-2 (NECT2_HUMAN)``.genes
nect2_human_genes
|> Option.toArray
|> Array.collect id
|> Array.iter (fun i ->
    match i.geneName with
    | None -> ()
    | Some name ->
        printfn "%s" name.value
)
```

3.5.4 Lazy Generation

The type provider interface supports on-demand generation of types and their members, which means that only types that are actually accessed in the code written by the user are generated. Types of members that are not accessed in the code written by the user are never needed and are thus not generated and do

not appear in the compiled code. This is particularly useful when working with large-scale data, as it minimises compilation time and the size of the generated code. The UniProtProvider leverages this capability in its implementation. A possible trade-off is a latency on the first access to a previously ungenerated member, as the provider dynamically constructs the required type. However, this delay is negligible compared to the cumulative savings in compilation speed and resource usage.

3.5.5 Asynchrony

F# provides a lightweight asynchronous programming model, enabling non-blocking operations through composable workflows [14]. The UniProtProvider leverages this paradigm to optimise performance in two key areas:

- API request handling: all interactions with the UniProt server are implemented as async methods (Section 3.3). This prevents thread starvation during network latency, allowing concurrent execution of other operations while awaiting responses.
- Caching: Disk reads and writes for cached JSON responses (Section 3.4) are executed asynchronously, allowing concurrent execution of other operations.

The UniProtProvider does not utilise parallel execution across type provider instances. This decision stems from two factors:

- Sequential query refinement: Users typically refine queries iteratively (e.g., filtering organisms before selecting proteins), a workflow naturally aligned with sequential execution.
- Lazy generation: Types are generated on-demand, eliminating the need to pre-generate results.

3.5.6 Large Results Set

Retrieving the full set of results (e.g., all human proteins) risks timeouts or significant delays, degrading the user experience. Even though initial queries request only necessary information, such as name and accession, the potential volume of results remains substantial.

To address this, the provider leverages the UniProt REST API's native pagination support, enabling efficient retrieval of results in manageable batches of 50 entries. This number is chosen to balance two objectives: providing a concise overview of the results without excessive scrolling through available properties and minimising perceived latency.

The API prioritises returning the most relevant matches first, thereby minimising the need for users to manually traverse entire datasets. [15] Internally, the type provider automates pagination by parsing the *Link* header included in HTTP responses, which contains URLs for subsequent batches. This header is stored and used for subsequent requests. Users navigate through result sets via "*More...*" property appended to each batch, abstracting HTTP mechanics.

3.6 Encountered Obstacles

Debugging a type provider during development presents unique challenges because it runs at both design-time and compile-time.

The runtime component - responsible for server communication, query construction and caching mechanisms - offers more straightforward debugging opportunities. By decoupling this logic from the design-time component, it can be tested independently using standard debugging techniques and sample inputs, much like conventional runtime code.

The situation differs significantly for design-time on-demand code generation. Typical debugging techniques cannot be directly applied to the process of dynamic type generation. While some issues manifest as explicit compilation errors, others might fail silently - for instance, cases where a member function is not added to the provided type despite apparently successful compilation. In addition, some of the error descriptions can be unclear, making it difficult to identify the problem.

This section describes frequent challenges and errors encountered when developing a type provider, as well as lessons learned and troubleshooting strategies.

"Object reference not set to an instance of an object"

This common .NET exception arises when accessing an uninitialised object. Within type provider development, such errors introduce diagnostic challenges, as they manifest in user code rather than within the provider implementation itself, and a stack trace is not available. The underlying issue often originates from within the generated type hierarchy – for example, incomplete or erroneous type-generation logic may produce null references where valid object instances were expected.

"GetMethodImpl. not support overloads"

This compilation error wrapped into a generic FS3021 "Unexpected exception from provided type" manifests when a type provider generates methods with identical signatures. A representative example appears as:

"error FS3021: Unexpected exception from provided type: The type provider 'UniProtProvider.DesignTime.ByKeyWord' reported an error: GetMethodImpl. not support overloads, name = 'get_Regenerating islet-derived protein 3-gamma', methods - '[|...|]'"

The verbose stack trace offers limited diagnostic value. The core issue stems from non-unique property definitions within generated types – here, multiple properties sharing the name "Regenerating islet-derived protein 3-gamma". In this case, appending unique identifiers to protein names resolved the problem.

Properties are added to an incorrect type

The similar issue arises when generated types share non-unique names. In this scenario, no compile error occurs, and the type provider remains functional but exhibits incorrect property aggregation. When multiple types receive the same name, they are coalesced into a single type definition, even if they were originally nested inside different types. Consequently, properties intended for distinct types

accumulate within one shared type definition, resulting in the exposure of incorrect members:

```
let mouseInulin : Protein = Inulin().ByOrganism<"mouse">().``Regenerating islet-derived protein 3-gamma (REG3G_MOUSE)``
let mutable humanInulin : Protein = Inulin().ByOrganism<"human">().
                         property ByKeyWord.ByKeyWord<...>.Inner ×   Beta-fructofuranosidase, insoluble isoenzyme CWIN…
                         Type.``Beta-fructofuranosidase, insolub     Beta-fructofuranosidase, insoluble isoenzyme CWIN…
                         le isoenzyme CWINV1 (INV1_ARATH)`` : Pr     Ig heavy chain V region AMPC1 (HVM34_MOUSE)
                         otein with get                              Ig heavy chain V-III region A4 (HVM27_MOUSE)
                                                                     Ig heavy chain V-III region ABE-47N (HVM30_MOUSE)
```

Figure 3.6 Wrong properties

To address this, UniProtProvider utilises a global variable with a counter within its type generation logic. Each generated type is assigned a unique numerical suffix.

”Type ’InnerType’ was not added as a member to a declaring type”

This compilation error arises when a generated nested type (e.g., InnerType) fails to register explicitly with its parent type during construction. While methods referencing InnerType may appear in the parent type’s definition, it is required to explicitly add nested types to a parent type using *AddMember* method.

”No value has been specified for ’DeclaringType’”

Similarly to the previous case, this compilation error may arise when generated methods lack explicit association with their parent type. While such methods may appear available during member inspection, they remain invalid for compilation.

3.6.1 Troubleshooting Techniques

To conclude, this section presents techniques employed in this work that could assist in addressing the implementation challenges arising during the development of type providers.

Separate the runtime component: Decouple runtime logic (data operations, caching) from design-time type generation. This separation enables independent validation of core functionality.

Develop tests: Implement granular test cases covering all the type provider functionalities. Such tests act as executable specifications, help with a better understanding of the user code, as well as ensure the expected functionality during development, validating all the subsequent changes.

Logging: Even rudimentary ’print-to-stdout’ logging may provide valuable insights into program behaviour.

4 Usage

This chapter describes different ways to install and use the UniProtProvider.

4.1 Getting Started

4.1.1 Installation From Sources

Clone the source repository:

```
https://github.com/Senya-P/UniProtProvider
```

Dependencies: .NET runtime 5.0+

Change the version inside the *UniProtProvider/global.json* file to the one installed on your machine.

Compile the project:

```
dotnet tool restore
dotnet paket update
dotnet build
```

Source files include tests. Running them requires .NET 6.0; newer versions are not supported.

Test execution

Tests are located at *UniProtProvider/tests/UniProtProvider.Tests/*

To run them, execute the following command inside the root directory:

```
dotnet test
```

4.1.2 NuGet

NuGet is an easy-to-use package manager for .NET ecosystem. The package is available at:

```
https://www.nuget.org/packages/UniProtProvider/#versions-body-tab
```

Add the package to your project by executing the following command:

```
dotnet add package UniProtProvider --version 1.0.10
```

To use the package, reference it from an .fsi (F# interactive) file:

```
#r "nuget: UniProtProvider, 1.0.10"
```

4.2 Interface

There are four type providers that work with the UniProt API, contained within the *UniProtProvider* namespace. These types are enriched with XML documentation, visible via IntelliSense, to help guide users without the need for external API references.

See *Code Samples* section for usage examples.

ById

Retrieves a single protein entry. Can be provided with *UniProtKBId* or a unique primary accession. [16]

ByKeyword

Retrieves all protein results matching the specified keyword - protein name or part of it. The result set can be further refined by calling the *ByOrganism<..>()* method, which accepts an organism name and returns a new subset of results that match both the protein name and the specified organism. To restrict results to manually reviewed proteins, the *Reviewed()* method may be invoked. Additionally, filtering by evidence type is supported through the *ByProteinExistence<..>()* method. The type provider returns results in batches of 50 entries. Subsequent batches are accessible via the *"More..."* property. Nevertheless, it is advised to refine search criteria as precisely as possible.

ByTaxonId

Retrieves a single taxonomy entry. Accepts unique identifier *TaxonId*.

ByOrganism

Retrieves all taxonomy results that match the provided organism name. Paging is also available. For the particular result, it is then possible to obtain all the protein entries related to the currently selected taxonomy entry by invoking the *FindRelated()* method.

4.3 Code Samples

This section provides code examples to get started with UniProtProvider. Note that individual properties are returned as an optional value, e.g. *string option* due to the variable schema. Each value should be checked for presence to avoid runtime errors.

Examine individual protein entries

```
UniProtProvider.ById<"A0AVG3">()
```

After the protein entry is retrieved, its properties can be examined via member access, as shown in Figure 4.1.

Examine individual taxonomy entries

```
UniProtProvider.ByTaxonId<9606>()
```

After the taxonomy entry is retrieved, its properties can be examined via member access, as shown in Figure 4.2.

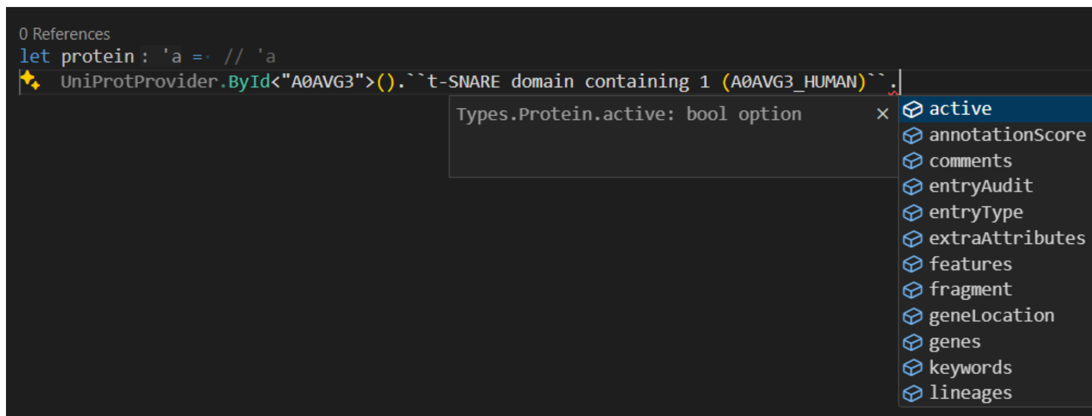


Figure 4.1 Individual protein entry properties

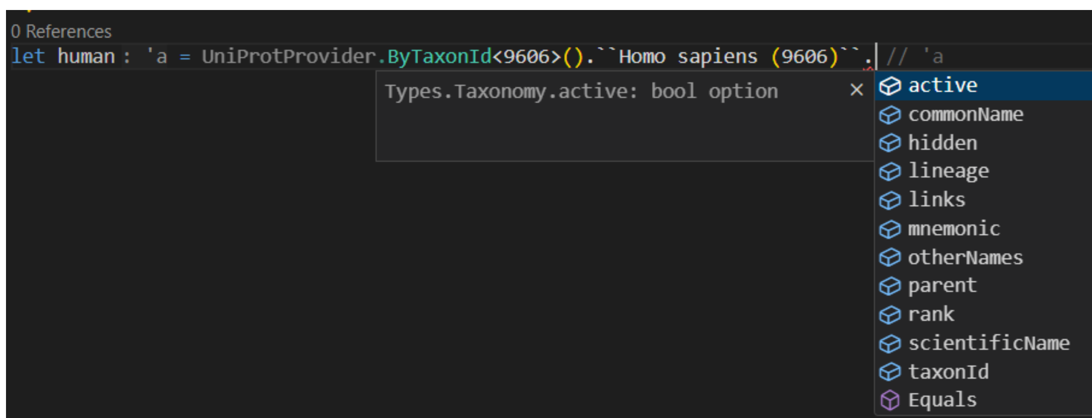


Figure 4.2 Individual taxonomy entry properties

Find protein entries by keyword

```
UniProtProvider.ByKeyWord<"inulin">()
    .``6(G)-fructosyltransferase (GFT_ASPOF)``
```

A list of the first fifty related entries is returned. To load more entries, *"More.."* property can be used.

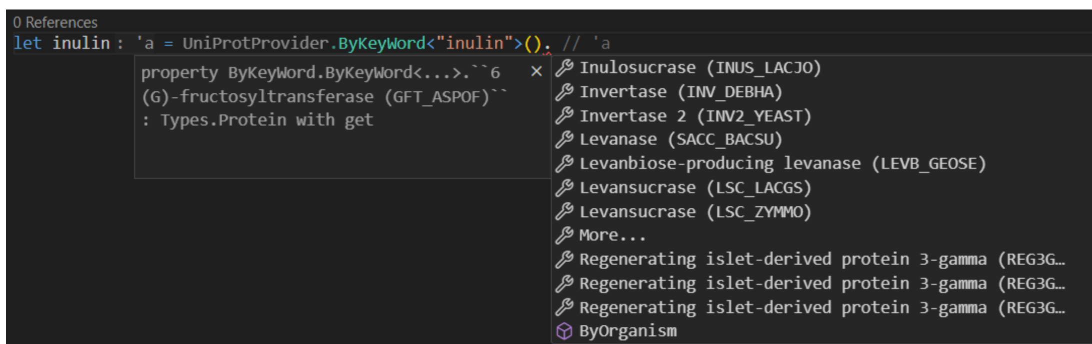


Figure 4.3 List of protein results

```
UniProtProvider.ByKeyWord<"kerain">().keratin
```

If no results are found, suggested keywords may be available.

```
0 References
let q : 'a = UniProtProvider.ByKeyWord<"kerain">(). // unit -> ById
        property ByKeyWord.ByKeyWord<...>.keral ×    kerala
        a: ByKeyWord.ByKeyWord<...>.InnerType19      keratin
        5 with get                                    keratini
                                                      kersin
                                                      kertain
```

Figure 4.4 List of suggested results

Filter protein entries by organism name

```
UniProtProvider.ByKeyWord<"inulin">()
    .ByOrganism<"mouse">()
    .``Ig heavy chain V region AMPC1 (HVM34_MOUSE)``
```

A list of results related to the given organism is returned.

```
0 References
let inulin : 'a = ByKeyWord<"inulin">().ByOrganism<"mouse">(). // 'a
    property ByKeyWord.ByKeyWord<...>.Inner ×   Ig heavy chain V-III region A4 (HVM27_MOUSE)
    Type257.``Beta-fructofuranosidase, inso      Ig heavy chain V-III region ABE-47N (HVM30_MOUSE)
    luble isoenzyme CWINV1 (INV1_ARATH)`` :      Ig heavy chain V-III region E109 (HVM29_MOUSE)
    Types.Protein with get                       Ig heavy chain V-III region J606 (HVM32_MOUSE)
                                                 Ig heavy chain V-III region T957 (HVM31_MOUSE)
                                                 Ig heavy chain V-III region U61 (HVM28_MOUSE)
                                                 Ig heavy chain V-III region W3082 (HVM33_MOUSE)
                                                 Ig kappa chain V-V region EPC 109 (KV5AI_MOUSE)
                                                 Ig kappa chain V-V region J606 (KV5AK_MOUSE)
                                                 Ig kappa chain V-V region W3082 (KV5AL_MOUSE)
                                                 Immunoglobulin kappa variable 11-125 (KV5AH_MOUSE)
                                                 Regenerating islet-derived protein 3-gamma (REG3G…
```

Figure 4.5 List of protein results filtered by organism

Find taxonomy entries by keyword

```
UniProtProvider.ByOrganism<"human">()
```

A list of the first fifty related entries is returned. To load more entries, *"More.."* property may be used.

Load more

For longer lists, property *"More..."* is available for loading the next result set on demand. It is recommended to narrow the result set as much as possible. Note that the type provider is primarily designed to work with individual entries, not for processing large amounts of data.

```
UniProtProvider.ByOrganism<"human"()
    .``More...``
    .``More...``
    .``More...``
```

...and so on. Sorting of protein search results is determined by a query score, which takes into account the annotation score, the importance of a search term,

Figure 4.6 List of taxonomy results

and how often the term appears in the results. Consequently, the most relevant results are likely to be found among the top few entries. [17]

List related protein entries

```
UniProtProvider.ByOrganism<"9606">()
    .``Homo sapiens (9606)``
    .FindRelated()
```

Protein entries associated with a specified organism can be returned.

Figure 4.7 List of related protein entries

List only reviewed protein entries

```
UniProtProvider.ByKeyWord<"inulin">()
    .ByOrganism<"human">()
    .Reviewed()
```

It is also possible to filter the protein results related to a specific organism.

```
UniProtProvider.ByOrganism<"human">()
    .``Homo sapiens (9606)``
    .FindRelated()
    .Reviewed()
```

Results are filtered to include only manually reviewed entries from UniProtKB.

Figure 4.8 List of reviewed protein entries

Filter protein entries by evidence type

```
UniProtProvider.ByKeyWord<"keratin">()
    .ByProteinExistence<ProteinExistence
        .InferredFromHomology>()
```

Protein existence indicates the type of evidence that supports the existence of the protein [18]. There are 5 types of evidence available in UniProtKB:

1. Experimental evidence at protein level
2. Experimental evidence at transcript level
3. Protein inferred from homology
4. Protein predicted
5. Protein uncertain

Internally they are represented as a *ProteinExistence* enum type.

Figure 4.9 Protein existence

Filtering options

Filtering of protein results can be applied in any order. Thus, all the following invocations are valid and return the same result.

```
UniProtProvider.ByKeyWord<"keratin">()
    .ByOrganism<"mouse">()
    .Reviewed()
    .ByProteinExistence<ProteinExistence
        .InferredFromHomology>()
```

```
UniProtProvider.ByKeyWord<"keratin">()
    .Reviewed()
    .ByOrganism<"mouse">()
    .ByProteinExistence<ProteinExistence
        .InferredFromHomology>()

UniProtProvider.ByKeyWord<"keratin">()
    .ByProteinExistence<ProteinExistence
        .InferredFromHomology>()
    .ByOrganism<"mouse">()
    .Reviewed()
```

4.4 Interactive Notebooks

It is possible to execute the type provider code using any notebook with .NET Interactive support. For example, Polyglot notebook [19], which is available as a Visual Studio Code extension. It supports referencing a NuGet package and using the type provider in an interactive environment, as shown in Figure 4.10.

Figure 4.10 Polyglot notebook

5 Evaluation

Implementing a completely new type provider is not the only way to integrate external data into a program. This chapter discusses the available alternatives.

5.1 JSON Type Provider

F# data is a library containing type providers for integrating common structured data types such as XML, CSV and JSON into the F# language type system. [3] While F#'s JSON type provider [2] can parse UniProt entries using a sample schema, it has limitations for bioinformatics applications:

Loss of semantic structure: JSON supports only predefined primitive types, such as numbers, strings, arrays and structures that combine them into key-value pairs, which do not provide insight into the complex structure of the protein entry. It would also allow one to examine individual entries or lists of entries, but there is no way to filter and refine queried results on the fly. The introduction of a type provider will wrap the members of the protein entry into a meaningful structure and allow us to extend it with useful member functions.

Lack of query abstraction: UniProt's REST API supports filtering via URL parameters (e.g. *?query=organism_id:9606&format=json*). The JSON type provider requires manual URL construction, exposing users to low-level HTTP syntax:

Listing 5.1 JSON Type Provider usage example

```
let humanProteins = JsonProvider<...>
    .Load("https://rest.uniprot.org/uniprotkb/search?query=
       keratin")
```

This approach becomes cumbersome for complex queries (e.g. combining taxonomy and keyword filters) and provides no compile-time validation of parameter names or values. A custom type provider can encapsulate this detail behind a fluent API:

Listing 5.2 UniProt Type Provider usage example

```
UniProtProvider.ByKeyWord<"keratin">()
               .ByOrganism<"human">()
```

Unstable schema: The JSON type provider requires a stable data schema: if the loaded file deviates from the structure of the sample, it may result in a runtime error. [2] This is the case with the UniProt schema, where individual entries may or may not contain some of the fields. In addition, the JSON type provider explicitly requires a sample to be provided from which the schema is inferred.

5.2 REST API Clients

Libraries such as HttpClient, available in most programming languages, allow developers to send raw HTTP requests directly to UniProt's REST API. While they are simple, they do require significant boilerplate:

Listing 5.3 UniProt REST API request example

```
let response = httpClient
    .GetStringAsync("https://rest.uniprot.org/uniprotkb/
       P42694")
    .Result
let protein = JsonConvert
    .DeserializeObject<Protein>(response)
```

Custom query builder: Similar to the JSON Type Provider, this approach requires manual query construction.

Manual pagination and caching: Handling large datasets (e.g. all protein entries) requires custom logic for caching and batch processing.

Manual type definition: Requires implementation of the type matching the JSON schema.

5.3 UniProtJAPI

A Java library from EMBL's European Bioinformatics Institute that provides a robust API for programmatic access to UniProt data. [20] It is suitable for integrating UniProt data into Java applications. UniProtJAPI represents types as Java interfaces, ensuring type-safe parsing of entries into structured objects. Its flexible and comprehensive query builder allows precise filtering of results.

Listing 5.4 UniProtJAPI usage example

```
try {
    ServiceFactory serviceFactoryInstance =
        Client.getServiceFactoryInstance();
    UniProtService uniProtService =
        serviceFactoryInstance.getUniProtQueryService();

    Set<String> accessions =
        new HashSet<>(Arrays.asList("P99999", "P99998"));
    Query query = UniProtQueryBuilder.accessions(accessions);
    QueryResult<UniProtEntry> entries =
        uniProtService.getEntries(query);
    entries.forEachRemaining((e) -> {
        System.out.println(
            e.getOrganism().getCommonName().getValue()
        );
    });
} catch (ServiceException ex) {...}
```

Runtime fragility: Missing fields or incorrect parameters result in runtime errors that require explicit exception handling and delay feedback cycles.

Java syntax verbosity: Requires complex setup, making this tool less accessible to people without Java development experience.

5.4 BioServices

BioServices is a Python library designed to simplify programmatic access to biological web services, including UniProt [21]. BioServices wraps the UniProt REST API and allows querying by accession number, protein name or advanced search terms. It also supports identifier mapping between different data sources (e.g. UniProt and KEGG).

Limited query customisation: Lack of support for UniProt REST parameters (e.g. field filtering), requiring manual URL construction.

No native object mapping: BioServices does not map UniProt data to domain-specific classes with intuitive attributes. Results are returned as Python dictionaries, Pandas data frames, or raw data such as XML.

Listing 5.5 BioServices usage example

```
from bioservices import UniProt
u = UniProt(verbose=False)
data = u.search("zap70+and+taxonomy_id:9606",
                frmt="tsv",
                limit=3,
                columns="id,length,accession, gene_names")
print(data)
```

5.5 Biopython

Biopython is an open-source Python library for biological computation. [22] This tool facilitates the conversion of bioinformatics files into usable data structures, including Swiss-Prot, a manually annotated and peer-reviewed subset of the UniProt knowledge base. In particular, it supports the parsing of plain text and XML formatted Swiss-Prot files into sequence records.

Listing 5.6 Biopython usage example

```
from Bio import ExPASy
from Bio import SwissProt
accessions = ["O23729", "O23730", "O23731"]
records = []
for accession in accessions:
    handle = ExPASy.get_sprot_raw(accession)
    record = SwissProt.read(handle)
    records.append(record)
```

Manually annotated data only: Biopython exclusively handles reviewed entries, omitting unreviewed TrEMBL entries and supporting datasets.

No integration with the UniProt REST API: Biopython relies on static data retrieval methods, requiring users to parse locally stored flat files or fetch entries via the ExPASy FTP server. These workflows return Swiss-Prot records in raw text or XML formats and do not support dynamic queries. In order to obtain accession numbers or to enable advanced search capabilities, the query construction must be handled separately:

Listing 5.7 Biopython query example

```
from Bio import UniProt

query = "(organism_id:2697049) AND (reviewed:true)"
results = list(UniProt.search(query))
```

5.6 Summary

While alternative approaches may provide a partial solution or be valuable for different scenarios, such as retrieving and processing the textual representation of a subset of protein entries, they may not inherently align with UniProt's complex schema and prioritise flexibility over domain specificity. A dedicated type provider bridges this gap by:

- Embedding domain semantics.
- Guaranteeing type safety.
- Abstracting API details.
- Providing user-friendly domain-specific methods.

This approach makes exploring UniProt data faster, less error-prone and more self-explanatory for programmers and biologists.

Conclusion

This work introduced a new tool for integrating structured biological data into a programming language's type system. As argued in Chapter 5 - Evaluation, none of the existing alternatives satisfy the requirements defined in the first chapter 1.4.

Future enhancements could extend the UniProtProvider's capabilities by incorporating additional filtering options, such as subcellular location, function, gene ontology or interactions, thereby replicating the full functionality of UniProt's advanced online search. Support for UniParc and UniRef databases could further broaden its utility, enabling direct exploration of protein sequence archive data and reference cluster datasets via the UniProtProvider. Integration of other supporting data types, such as proteomes or disease associations, represents another possible extension.

Improvements to the implementation might include introducing asynchronous retrieval of properties. This would eliminate redundant requests, taking advantage of an asynchronous model described in Section 3.5.5. Additionally, implementing an automatic cache expiration mechanism would remove the need for manual deletion of outdated files, as detailed in Section 3.4.

Finally, the type generation strategy could be re-evaluated. The current implementation, described in Section 3.5.1, employs static record types. This has several disadvantages, such as dependence on the UniProt schema. An alternative implementation would generate end types dynamically, using reflection and schema inference, similar to the approach used for the JSON type provider.

Overall, this thesis proposes a tool that advances bioinformatic data exploration with a user-friendly, domain-specific, and robust approach.

Bibliography

1. SYME, Don; BATTOCCHI, Keith; TAKEDA, Kenji; MALAYERI, Donna; FISHER, Jomo; HU, Jack; LIU, Tao; MCNAMARA, Brian; QUIRK, Daniel; TAVEGGIA, Matteo; CHAE, Wonseok; MATSVEYEU, Uladzimir; PETRICEK, Tomas. *F#3.0 – Strongly-Typed Language Support for Internet-Scale Information Sources*. Microsoft Research, 2012-09. Tech. rep., MSR-TR-2012-101. Microsoft Research, Microsoft Corporation, and University of Cambridge. Available also from: `https://www.microsoft.com/en-us/research/publication/f3-0-strongly-typed-language-support-for-internet-scale-information-sources/`.
2. FSHARP.DATA; PETRICEK, Tomas; GUERRA, Gustavo. *JSON Type Provider*. Available also from: `https://fsprojects.github.io/FSharp.Data/library/JsonProvider.html`.
3. PETRICEK, Tomas; GUERRA, Gustavo; SYME, Don. Types from data: making structured data first-class citizens in F#. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. ACM, 2016. Pp. 477–490. DOI: `10.1145/2908080.2908115`.
4. THE UNIPROT CONSORTIUM. UniProt: the Universal Protein Knowledgebase in 2025. *Nucleic Acids Research*. 2025, vol. 53, no. D1, pp. D609–D617. Available from DOI: `10.1093/nar/gkae1010`.
5. THE UNIPROT CONSORTIUM. *The Universal Protein Resource* [online]. [visited on 2025-04-05]. Available from: `https://github.com/ebi-uniprot/uniprot-manual/blob/main/pdfs/uniprot_flyer.pdf`.
6. THE UNIPROT CONSORTIUM. *Programmatic access*. Available also from: `https://www.uniprot.org/help/api_queries`.
7. KENNA, Alex; SMITH, Samuel; HOGAN, James. *BioProviders* [online]. [visited on 2025-04-05]. Available from: `https://fsprojects.github.io/BioProviders/`.
8. MICROSOFT. *F# programming language*. Available also from: `https://learn.microsoft.com/en-us/dotnet/fsharp`.
9. MICROSOFT. *Type Providers SDK* [online]. [visited on 2025-04-05]. Available from: `https://fsprojects.github.io/FSharp.TypeProviders.SDK`.
10. MICROSOFT. *F# Type Provider Tutorial*. Available also from: `https://learn.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/creating-a-type-provider`.
11. CIESLAK, Krzysztof; CONTRIBUTORS. *Ionide* [online]. [visited on 2025-04-05]. Available from: `https://ionide.io`.

12. Taha, Walid. A Gentle Introduction to Multi-stage Programming. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Lengauer, Christian; Batory, Don; Consel, Charles; Odersky, Martin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50. isbn 978-3-540-25935-0. Available from doi: 10.1007/978-3-540-25935-0_3.
13. FSharp.Data. *ProvidedTypeDefinition* [online]. [visited on 2025-04-05]. Available from: https://fsprojects.github.io/FSharp.Data.Toolbox/reference/providerimplementation-providedtypes-providedtypedefinition.html.
14. Syme, Don; Petricek, Tomas; Lomov, Dmitry. The F# Asynchronous Programming Model. *Proceedings of Practical Aspects of Declarative Languages*. PADL 2011. 2011.
15. The UniProt Consortium. *Programmatic Pagination* [online]. [visited on 2025-04-05]. Available from: https://www.uniprot.org/help/pagination.
16. The UniProt Consortium. *UniProt Accession*. Available also from: https://www.uniprot.org/help/accession_numbers.
17. The UniProt Consortium. *Query score* [online]. [visited on 2025-04-05]. Available from: https://www.uniprot.org/help/annotation_score.
18. The UniProt Consortium. *Protein existence* [online]. [visited on 2025-04-05]. Available from: https://www.uniprot.org/help/protein_existence.
19. .NET Foundation. *Polyglot Notebooks* [online]. [visited on 2025-04-05]. Available from: https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.dotnet-interactive-vscode.
20. The UniProt Consortium. *UniProtJAPI* [online]. [visited on 2025-04-05]. Available from: https://www.ebi.ac.uk/uniprot/japi/usage.html.
21. Cokelaer, Thomas; Pultz, Dennis; Harder, Lea M.; Serra-Musach, Jordi; Saez-Rodriguez, Julio. BioServices: A common Python package to access biological Web Services programmatically. *Bioinformatics*. 2013, vol. 29, no. 24, pp. 3241–3242. Available from doi: 10.1093/bioinformatics/btt547.
22. Cock, Peter J. A.; Antao, Tiago; Chang, Jeffrey T.; Chapman, Brad A.; Cox, Cymon J.; Dalke, Andrew; Friedberg, Iddo; Hamelryck, Thomas; Kauff, Frank; Wilczynski, Bartek; Hoon, Michiel J. L. de. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*. 2009, vol. 25, no. 11, pp. 1422–1423. Available from doi: 10.1093/bioinformatics/btp163.

List of Figures

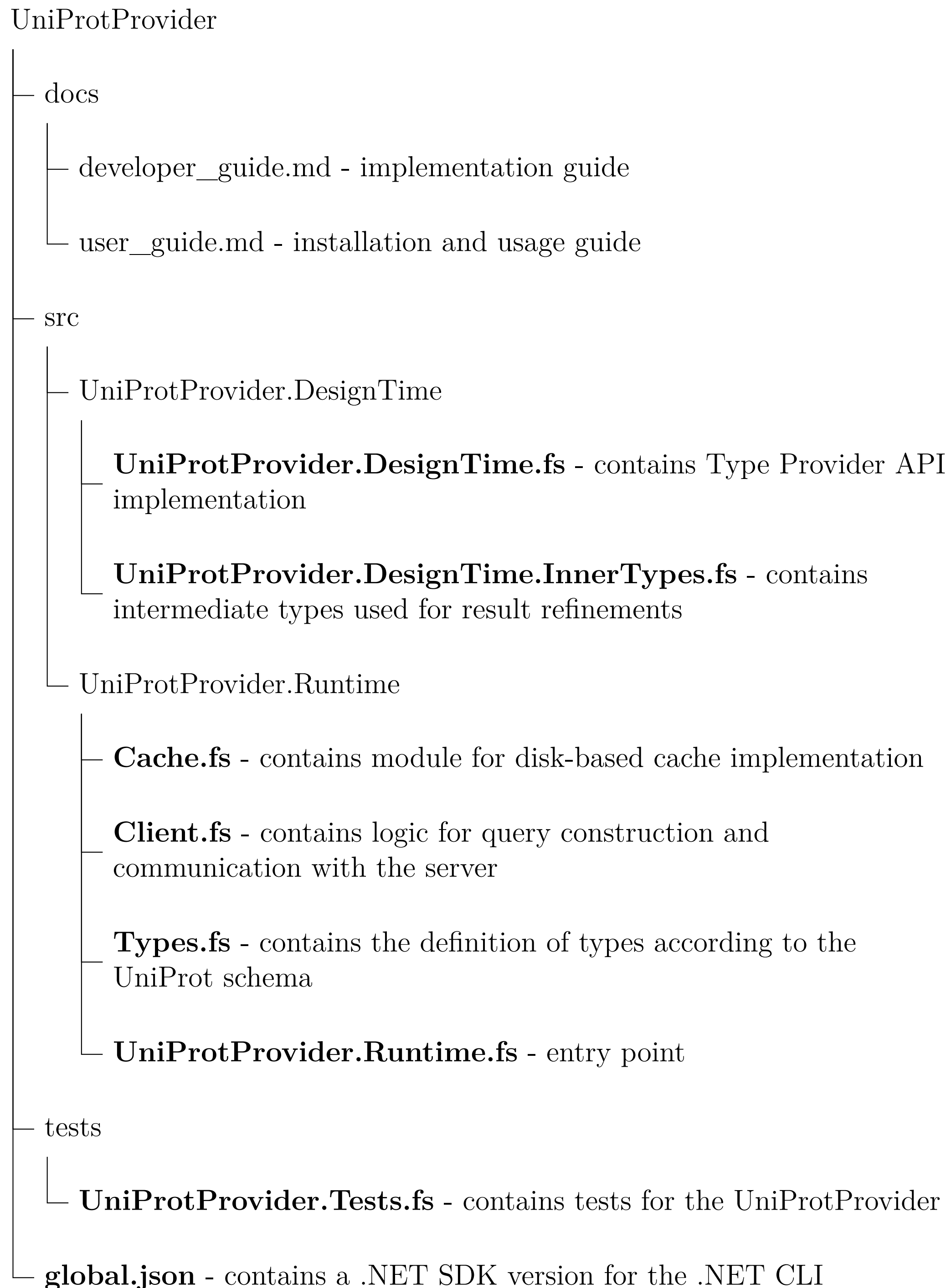
1.1 UniProt data overview . . . 10
1.2 Protein entry . . . 11

3.1 ByKeyword results . . . 18
3.2 ByOrganism results . . . 19
3.3 “More...“ property . . . 21
3.4 Parameters configuration . . . 21
3.5 Overview of generated types . . . 23
3.6 Wrong properties . . . 27

4.1 Individual protein entry properties . . . 30
4.2 Individual taxonomy entry properties . . . 30
4.3 List of protein results . . . 30
4.4 List of suggested results . . . 31
4.5 List of protein results filtered by organism . . . 31
4.6 List of taxonomy results . . . 32
4.7 List of related protein entries . . . 32
4.8 List of reviewed protein entries . . . 33
4.9 Protein existence . . . 33
4.10 Polyglot notebook . . . 34

A Attachments

A.1 Overview of the UniProtProvider Implementation



A.2 Protein Entry JSON Example


```
{
    "entryType": "UniProtKB reviewed (Swiss-Prot)",
    "primaryAccession": "P14735",
    "secondaryAccessions": [
        "B2R721",
        "B7ZAU2",
        "D3DR35",
        "Q5T5N2"
    ],
    "uniProtkbId": "IDE_HUMAN",
    "entryAudit": {
        "firstPublicDate": "1990-04-01",
        "lastAnnotationUpdateDate": "2025-04-09",
        "lastSequenceUpdateDate": "2008-11-25",
        "entryVersion": 231,
        "sequenceVersion": 4
    },
    "annotationScore": 5,
    "organism": {
        "scientificName": "Homo sapiens",
        "commonName": "Human",
        "taxonId": 9606,
        "lineage": [
            "Eukaryota",
            "Metazoa",
            "Chordata",
            "Craniata",
            "Vertebrata",
            "Euteleostomi",
            "Mammalia",
            "Eutheria",
            "Euarchontoglires",
            "Primates",
            "Haplorrhini",
            "Catarrhini",
            "Hominidae",
            "Homo"
        ]
    },
    "proteinExistence": "1: Evidence at protein level",
    "proteinDescription": {
        "recommendedName": {
        "fullName": {
            "evidences": [
                {
                    "evidenceCode": "ECO:0000303",
                    "source": "PubMed",
                    "id": "20364150"
                }
            ],
            "value": "Insulin-degrading enzyme"
        },
        "ecNumbers": [
            {
                "evidences": [
                    {
```

```
                    "evidenceCode": "ECO:0000269",
                    "source": "PubMed",
                    "id": "10684867"
                },
                {
                    "evidenceCode": "ECO:0000269",
                    "source": "PubMed",
                    "id": "17051221"
                },
                {
                    "evidenceCode": "ECO:0000269",
                    "source": "PubMed",
                    "id": "21098034"
                },
                {
                    "evidenceCode": "ECO:0000269",
                    "source": "PubMed",
                    "id": "2293021"
                }
            ],
            "value": "3.4.24.56"
        }
    ],
    "alternativeNames": [
        {
            "fullName": {
                "value": "Abeta-degrading protease"
            }
        },
        {
            "fullName": {
                "evidences": [
                    {
                        "evidenceCode": "ECO:0000303",
                        "source": "PubMed",
                        "id": "20364150"
                    }
                ],
                "value": "Insulin protease"
            },
            "shortNames": [
                {
                    "evidences": [
                        {
                            "evidenceCode": "ECO:0000303",
                            "source": "PubMed",
                            "id": "20364150"
                        }
                    ],
                    "value": "Insulinase"
                }
            ]
        },
        {
            "fullName": {
                "evidences": [
                    {
                        "evidenceCode": "ECO:0000303",
```

```
                            "source": "PubMed",
                            "id": "20364150"
                        }
                    ],
                    "value": "Insulysin"
                }
            }
        ]
    },
    "genes": [
        {
            "geneName": {
                "evidences": [
                    {
                        "evidenceCode": "ECO:0000303",
                        "source": "PubMed",
                        "id": "20364150"
                    },
                    {
                        "evidenceCode": "ECO:0000312",
                        "source": "HGNC",
                        "id": "HGNC:5381"
                    }
                ],
                "value": "IDE"
            }
        }
    ],
    "comments": [...], // 15 items
    "features": [...], // 119 items
    "keywords": [...], // 20 items
    "references": [...], // 22 items
    "uniProtKBCrossReferences": [...], // 289 items
    "sequence": {
        "value": "MRYRLAWLLHPALPSTFRSVLGARLPPPERL...MAAKL",
        "length": 1019,
        "molWeight": 117968,
        "crc64": "8A28AEF75EDA0EDA",
        "md5": "5E2C446CC1C54EE4406B9F6683B7F98D"
    },
    "extraAttributes": {
        "countByCommentType": {...}, // 13 items
        "countByFeatureType": {...}, // 11 items
        "uniParcId": "UPI000013D6B6"
    }
}
```