



UNIVERSITÀ degli STUDI di CATANIA

DIPARTIMENTO DI INGEGNERIA ELETTRICA,
ELETTRONICA E INFORMATICA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Alessio Giordano

A JSON HTTP NOTATION FOR EXTENDING
APPLICATION SERVERS WITH THIRD-PARTY SERVICES:
A CASE STUDY ENABLING A VIRTUAL FILE SYSTEM
FOR THE UNIVERSITY CLOUD SUITE

Relatore: Prof. Simone Palazzo

ANNO ACCADEMICO 2022/2023

Abstract

This thesis presents a language specification, a runtime implementation, and an integrated development environment for a plug-in system aimed at application servers, called the *JSON HTTP Notation (JOHN)*. It is a domain-specific language built on top of JavaScript objects due to the ubiquity of JSON deserialization, which makes it easier to write a syntax parser. It is a declarative notation of a chain of HTTP requests, as most custom-defined web services are REST interfaces exchanging HTTP messages. This pipeline receives parameters from the runtime, and it can interpolate those values, as well as the response bodies of each request, with the request directives of the next. Finally, it provides a well-defined output to the caller. A single JOHN request, besides implementing the essential components of an HTTP request, also provides advanced authorization, control flow, repetition, and merging capabilities. Additionally, the request and response bodies are decoded from and into many standard representations such as base64, web form, JSON, XML, HTML, and SOAP. The reference implementation of this specification is a Swift library named the *JOHN Runtime*, which also provides the necessary debug capabilities of the execution context to support the creation of development environments, like the *JOHN Editor* for macOS.

Sommario

Il lavoro di tesi presenta la specifica del linguaggio, un'implementazione dell'ambiente di esecuzione e un ambiente di sviluppo integrato per un sistema di plug-in destinato ad applicazioni server, che prende il nome di *JSON HTTP Notation (JOHN)*. Si tratta di un linguaggio domain-specific costruito su oggetti JavaScript per via dell'ubiquità della deserializzazione di JSON, che semplifica la scrittura di un parser per la sintassi. È una notazione dichiarativa di una serie di richieste HTTP poiché la maggior parte dei servizi web proprietari definisce interfacce REST che scambiano messaggi HTTP. Questa pipeline riceve parametri dall'ambiente di esecuzione e può interpolare quei valori, così come i body di risposta di ciascuna richiesta, con le direttive della richiesta successiva. Infine, fornisce un risultato ben definito al chiamante. Una singola richiesta *JOHN*, oltre a implementare i componenti essenziali di una richiesta HTTP, offre anche funzionalità avanzate di autorizzazione, controllo del flusso, ripetizione e paginazione. Inoltre, i body di richiesta e risposta vengono decodificati da e verso molte rappresentazioni standard come base64, web form, JSON, XML, HTML e SOAP. L'implementazione di riferimento di questa specifica è una libreria Swift chiamata *JOHN Runtime*, che fornisce anche le funzionalità di debug del contesto di esecuzione necessarie per supportare la creazione di ambienti di sviluppo, come il *JOHN Editor* per macOS.

Contents

1	Introduction	1
2	State of the Art	9
2.1	OpenAPI Specification	10
2.2	Photoshop Plug-In Architecture	14
3	The JSON Data Interchange Syntax and the Swift Programming Language	24
3.1	Parsing JSON Objects with the Swift Codable Protocol	29
3.2	Swift Non-Blocking IO and Async HTTP Client	33
3.3	Automating Initialization with Introspection and Property Wrappers	35
3.4	Debugging Task Execution with Delegation	39
3.5	SwiftUI and Result Builders	40
4	JOHN: the JSON HTTP Notation	48
4.1	Specification of the JOHN Language	48
4.1.1	Anatomy of a JOHN Plugin	49

About	50
The variable subscript syntax	51
4.1.2 Customizing the HTTP request	52
Stage	52
4.1.3 Overriding default policies	54
Redirection	57
4.1.4 Manipulating the cookie store	58
Cookies	58
4.1.5 Providing and verifying input with parameters and assertions	59
Parameters	59
Assertions	60
4.1.6 Automating pagination with repetition and merging	61
Repetitions	61
Merger	63
4.1.7 Handling groups of plug-ins together	65
4.1.8 Yielding a normalized result	66
Result	66
The result subscript syntax	66
The result value mapping	68
4.1.9 Example usage of advanced JOHN features	70

4.1.10	Quick reference of the JOHN Language syntax	72
4.2	The JOHN Runtime for Swift Server-side Development	73
4.2.1	Implementing the language's data structures	74
4.2.2	Automatic parsing of plug-ins	77
4.2.3	Using JOHN plug-ins in Swift	79
	Asynchronous execution	79
	Options and Execution Groups	80
	Type-safe result interface	84
	Error handling	89
4.2.4	Inspecting the runtime	91
	DebugDelegate	91
	LoggerDelegate	94
4.3	Developing JOHN Plug-Ins with the JOHN Editor	95
4.3.1	Anatomy of the JOHN Editor	95
4.3.2	Using the plug-in Inspector	98
4.3.3	Accessing debugging information	100
5	Case study: the University Cloud Suite	102
5.1	The JOHN-powered Virtual File System	104
5.1.1	JOHN plug-in protocols	104
5.1.2	Database tables and REST endpoints	106

5.1.3	The SharedWorker and Indexed Database	112
5.2	The Desktop-class User Experience	115
5.2.1	Password-less and Username-less Authentication	116
5.2.2	Building Desktop-class Web Components	125
	Custom Elements and the Shadow DOM	125
	Buttons	128
	Icons	129
	Analog Clock	132
	Localization	133
	Page Layouts	134
	Miscellaneous Components	135
	Windows and Window Management	136
5.2.3	Bringing Spatial File Management to the Web	139
	The Object Tree Traversal	140
	The Converters For LibreOffice-Accepted File Formats	142
5.3	Building Tools for Students	146
5.3.1	Albo	149
5.3.2	Auditorium	151
5.3.3	Syllabus	152
6	Conclusion	155

Chapter 1

Introduction

It is common to hear that technology is initially expensive, and with time — as it grows and can harness the advantages of an economy of scale — it enters a positive loop, becoming more affordable and therefore increasing its user base, which in turn enables it to reach an even lower price and so on. For software, being expensive means that its purchase is only justifiable if it can enhance the buyer's productivity, which can directly lead to increased profits or saved time that offsets the purchasing cost of the product. Unsurprisingly, when cloud computing dramatically lowered the cost of entering the software market, it manifested itself primarily through Software-as-a-Service solutions targeting enterprise companies and personal productivity. Bootstrapping such an endeavor is no easy task, as a solid corporate structure is necessary to instill trust in businesses and institutions for them to adopt the software solution. Consequentially, small ISVs or independent developers will likely target personal productivity for their solutions. In a modern environment with computing systems being commonplace, it is no longer

sufficient to digitalize a workflow from the physical world, as established digital workflows exist, and those users have to perform them. When computer-based work involves other people, implementing the solution becomes the least complex problem: for it to be a success, an entire group needs to be convinced to adopt the software, not just a single person. Regardless, a solution that implements all the functionality to satisfy a workflow's needs will also result in a program with high CPU utilization and storage space occupation, which may be expensive to support for just a few users. Additionally, individuals may be wary of locking their data into a silo that will make working with it outside the perimeters of the service cumbersome or undesirable.

Services established in the early days of networking were standardized, electronic mail protocols such as IMAP and SMTP being the most known examples, but increasingly new services implement proprietary interfaces. Even though those interfaces may be published via human-readable documentation or machine-readable specifications (see *OpenAPI*) to be used by third-party vendors, there are still issues. First, since they do not have to follow industry standards, even when implementing similar services (for instance, cloud storage), they all differ slightly, making it harder to manage on the server side. Second, to maximize the customer base, the number of services, some of which might be for internal use only in an organization, is excessive for a single entity to handle. A plug-in system is the

only way for newcomer vendors to support a large install base with each user's requirements satisfied.

There is no standardized way to embed a plug-in system into a software program, so a question that arises when tackling this challenge is what are the design goals for one developed specifically for server applications:

- The runtime environment has to be secure for the server, meaning that arbitrary code execution is not allowed and must place constraints upon resource usage during operation.
- The language syntax has to be simple to implement, avoiding locking the vendor in any specific platform stack. Either they will write their own or choose from the available open-source runtimes. In any case, the source code of a plug-in will not require changes when switching from one to another.
- Writing a plug-in is the task of a subset of customers known as power users. It takes just one of them in an organization to unlock the power of the software for everyone. It is not required to be well-skilled in software development, and the language for writing plug-ins is straightforward. Crucially, no interaction between the software vendor, the service provider, and the plug-in writer is required, excluding the vendor defining the parameters and the expected output of a particular plug-in type.

This thesis presents a solution to the previously mentioned problem given dur-

ing the development of a software suite known as the *University Cloud Suite*, a family of several web-based tools to enhance the productivity of university students. The *JSON HTTP Notation*, or *JOHN* for brevity, is a domain-specific language constructed upon JavaScript objects for expressing and combining HTTP requests and their response bodies in a pipeline that accepts known parameters as input and can yield a customized output. JOHN is used in the University Cloud Suite to enable power users to connect the cloud service provider of their choice to the suite's spatial file manager, known as the *Object Tree Traversal*. The case study section of this document examines the *University Cloud Suite*, the various components that make it up, and the way it uses JOHN.

JOHN is defined on top of the JSON syntax to make it easier to implement across a wide range of languages, as JSON deserialization is universally supported, reducing enormously the complexity required to write the syntax parser. JOHN speaks exclusively via HTTP as it is the base of modern REST APIs, and the most common instances of non-HTTP interfaces are well standardized and available to back-end software developers as libraries. As a declarative language, JOHN represents what needs to be accomplished instead of the necessary individual steps to archive it:

- A plug-in can declare the input that it expects the runtime to provide. In JOHN, this is called a parameter. Parameters can be single values, a dic-

tionary of values, or even nested objects. A parameter can be supplied with a boolean value to signify that it is required (execution will not continue if missing) or not, or even a string value to use as a fallback.

- Web services mostly use authorization based on JSON Web Tokens (JWT) through the `HTTP Authorization` header, but sometimes they implement custom login flows. JOHN plug-ins can easily set either `HTTP base` or `bearer Authorization` header with a single directive in the language. When the server asks the runtime to execute the plug-in, it can supply either a token or a pair of username and password strings. If any parameters contain the fallback value of either `username`, `password`, or `token`, or an individual request asks for a `base` or `bearer` authorization, the runtime will know which value to use or halt the execution otherwise.
- An execution context defines the shared memory for cookies and caches used to run multiple plug-ins. Each request supports an `id` directive to specify a unique string value in the context to guarantee a one-time execution, and subsequent calls will return a cached response. This directive is especially useful for the recursive execution of a plug-in to avoid overloading a server with duplicate requests, such as logins, without compromising the self-contained nature of each plug-in. Its complement is the `defer` directive that lets the runtime know that the request should only run after the group

has completed its lifecycle. An obvious use case would be logging out.

- Plug-ins support a “do-while” block of sorts via the **repeat** directive, which accepts a dictionary of three integers for the **start**, **step**, and **stop** values of the iterator that is available for use in other directives of the request. Iterations enable the retrieval of various chunks of a database query without duplicating code. A separate **merge** directive defines how to handle the pagination of a repetition, whatever it is resulting from a **repeat** directive or individual requests. While a plug-in specifies an end value for repetitions, they are disabled in the execution group unless the server provides the runtime an upper bound before running one in order to avoid infinite loops from malicious or malfunctioning plug-ins.
- Before handling a request, either as a single entity or part of a repetition, an optional **assert** directive provides a minimal control flow mechanism. It allows checking the presence of a value in a parameter, a previous iteration or request, or the containment of said value in an array of strings. A single request fail will abort the whole plug-in execution, while failures inside a repetition will conclude the loop and resume linear execution before reaching the specified end bound.
- Obviously, to declare an HTTP request requires directives to set its url, method, headers, body, query parameters, cookies and accepted status codes.

-
- A plug-in may need to make more than a single request to provide the expected result. It is required for the runtime to have a system to convert from and to many different representation of request and response bodies. This conversion can happen automatically or be overridden via the **encode** and **decode** directives. Encodings include base64 for binary data, web form and JSON for objects and XML, HTML and SOAP for trees. A variable system allows the result of each request to be browsed and interpolated with subsequent requests.
 - Finally, a **result** directive provides a mechanism to interpolate variables from the various executed requests into the expected result structure by the plug-in caller. Single values, dictionaries of values or mapping of arrays (in the functional programming sense of the word) are allowed. Reserved keywords of **assert**, **catch**, and **result** permit the conditional assignment of values to an output, constructing an enumeration.

Since the University Cloud Suite uses the Swift programming language for its server-side components, the reference implementation of JOHN is a Swift library named *JOHN Runtime*. This library provides all the features enumerated in version 0.3 of the specification, enabling servers to execute plug-ins. In addition, delegate methods are available to IDEs importing the library to inspect the runtime and provide useful debug information to a JOHN plug-in writer. As the only

JOHN implementation in existence is a Swift library, it makes sense that the first JOHN IDE is also a Swift program, called the *JOHN Editor*, taking advantage of the debugging interfaces it exposes to create a simple SwiftUI-based code editor for the Mac along the lines of the venerable AppleScript Editor, designed explicitly to check the validity of the syntax provided and test its behavior using the same runtime code as it will run on the server.

All together, the *JOHN Language Specification*, the *JOHN Runtime* and the *JOHN Editor* provide everything a software vendor needs to expose JOHN plug-in interfaces to their best users and for them to write powerful extensions of their favourite software.

Chapter 2

State of the Art

The aim of extending a piece of software to link it to uncooperating third-party services is very unusual. When it comes to plugins, best exemplified by the extensions to image manipulation programs like “The Gimp” or Adobe’s Photoshop, they process the user’s data, managed by the primary software, for the most part. When they connect to a remote service, say a syncing plugin for a cloud storage service like Dropbox or a video streaming software like OBS, it is implemented either by the third party itself or the providers of the complete end-user solution, who take responsibility for keeping it working correctly. What binds together all the examples above is the complexity of the development experience, clearly targeted toward experienced software engineers. This chapter will identify three popular ways to achieve some of the goals of the JSON HTTP Notation, describe them in their architecture, and make clear why they are not suitable for the intended user and runtime environment.

2.1 OpenAPI Specification

At first glance, it might seem that the JSON HTTP Notation is redundant, as the mere definition of APIs in a serialized file format is a solved problem in the computing industry through the OpenAPI initiative. OpenAPI is an industry specification that enables the definition of HTTP services in either YAML or JSON documents [26], which can then be parsed by a code generator [53] or a documentation compiler [6]. In essence, OpenAPI serves as an automation mechanism to reduce the time and potential error introduced by defining an API in a human-readable specification and then having a team of engineers on both sides of the application stack implementing the client and the server interfaces. This approach taken to the extreme is also referred to as “spec-driven development”, as new API features are inserted first in the machine-readable specification. OpenAPI documents, like JOHN documents, are based on Javascript objects and can be defined, as mentioned before, in JSON directly or as YAML files. YAML is a markup language focused on human readability [65], and while developed independently, it shares the semantics of JSON, and all valid JSON files are also valid YAML files [41]. It is possible to write an OpenAPI specification in YAML and round-trip back to JSON for the convenience of those maintaining the document. But ultimately, OpenAPI only ensures the correctness of the endpoint that is invoked, and in no way allows the definition of custom business logic or return values: it is a document format,

not programming language in itself. While the specification accounts for custom fields by prepending those with the `x-` prefix [34], and therefore it is possible to add the JOHN syntax not covered by OpenAPI to be consumed by the JOHN Runtime, a quick look at a sample document, even in its YAML form, makes it clear why it would not be a desirable syntax, especially for the target audience of JOHN.

Listing 2.1: An example OpenAPI document written in YAML

```
1 openapi: "3.0.3"
2 info:
3   title: "Example Service"
4   version: "1.0.0"
5   description: "This sample API serves to illustrate the Open API
6     ↪ specification"
7   termsOfService: "https://www.example.com/tos"
8   license:
9     name: "Apache 2.0"
10    url: "https://www.apache.org/licenses/LICENSE-2.0.html"
11 servers:
12   - url: "localhost:{port}"
13     description: "Development Server"
14     variables:
15       port:
16         enum:
17           - "443"
18           - "80"
19         default: "80"
20         description: "The port the server is currently
21           ↪ listening to"
22   - url: "https://api.example.org"
23     description: "Production Server"
24 paths:
25   /helloworld:
26     get:
27       operationId: "getHelloWorld"
28       deprecated: false
```

```
27         description: "Returns an example response"
28         responses:
29             "200":
30                 description: "The server is working properly"
31                 content:
32                     application/json:
33                         schema:
34                             $ref: "https://www.example.com/
                                ↪ definitions.yaml#/getHelloWorld"
```

The root element, ignoring metadata about the document inside the info section, splits the definition of an API endpoint, which is composed of the server and the path, alongside secondary information like the authorization flow, into separate sections. This allows for shorter documents as it is not required to rewrite individual paths for a different combination of server configurations, like a production environment for API clients and a development environment for its maintainers. For even more succinct declarations, it is possible to interpolate variables in server definitions alongside enumerations of accepted values and default fallbacks. Paths are prepended with a forward slash and may contain variables as servers do, intended for templating. For request and response bodies, it is possible to provide references to a well-defined schema for each combination of **Content-Type** and response codes. It is possible to supply examples of the usage of a particular API endpoint in-line with its definition or referenced from separate files: this shows the documentation-building capabilities of the specification. Although not a suitable substitute for the language syntax of the JSON HTTP Notation, OpenAPI could

be very useful as part of the plugin development process to generate a stub of a plugin from one of the endpoints described in a document.

Future versions of the JOHN Editor could use OpenAPI similarly to how the AppleScript Editor uses “scripting dictionaries” (see Fig. 2.1) to make scriptable actions of programs installed on a user’s system more discoverable [8].

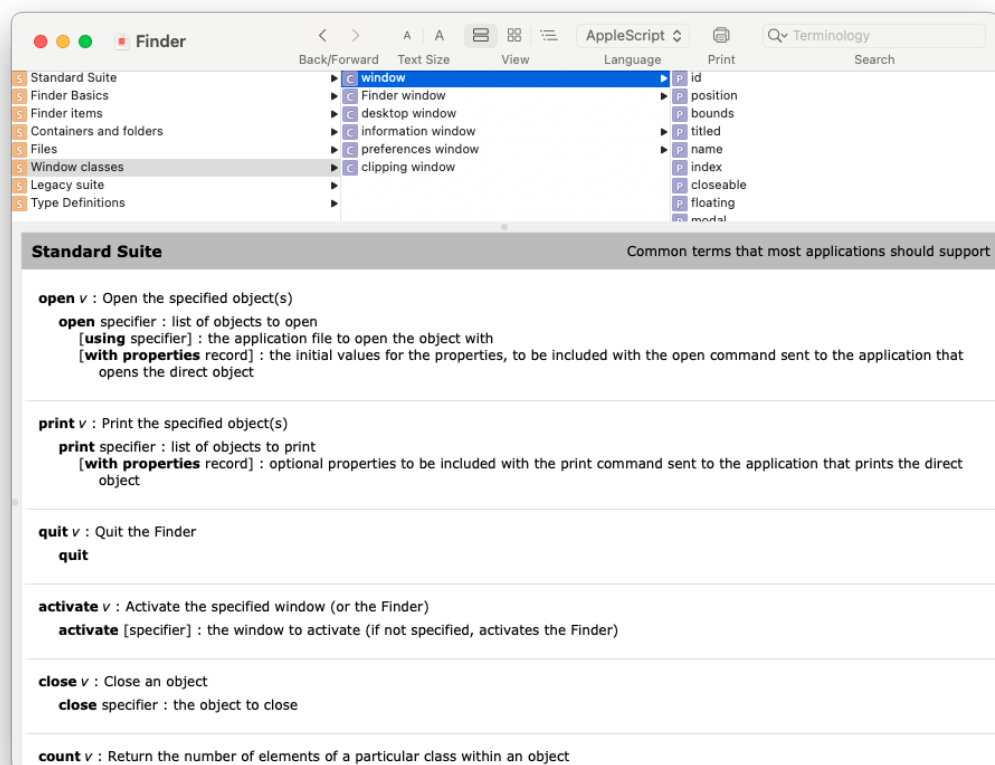


Figure 2.1: Viewing an app’s scripting dictionary in AppleScript Editor

2.2 Photoshop Plug-In Architecture

Software plug-ins have been pervasive in high-end productivity applications beginning with graphics programs like Silicon Beach's SuperPaint in the 1980s and later Adobe Photoshop [75]. The latter is an interesting case study for plug-ins as it has dominated its field for decades and into the present day, evolving its plug-in architecture to meet the demands of a changing workforce of professional software developers.

From the oldest to the newest technology, there is a constant increase in the level of abstraction of the programming language used and the isolation of the plug-in from the main application. Interestingly, for each point in time, the highest-level programming environment is picked by the software vendor for their plug-ins: in the early 1990s, the choice of an object-oriented language like C++ came with memory and performance tradeoffs compared to C or even Assembly programming that made it unsuitable for performance critical tasks; in the present day, the same is true for JavaScript, popularized by the ubiquity of the Web, with C++ receding as a more complex and unsafe, yet performant language. Performance is not a primary concern in designing the architecture: the recommended architecture for Photoshop plug-ins until very recently involved the combination of a custom implementation of JavaScript from 1997 (called ExtendScript) for the back-end logic and essentially an entire Chrome browser instance for the front-end (what

Extensibility Option	Use Case	Skill Level	Programming Language(s)	Photoshop Minimum version	Under Active Development by Adobe	Associated File Extension
UXP Scripts	Speed up repetitive tasks.	Beginner	JavaScript (ECMAScript >6)	23.5	Yes	PSJS
UXP Plugins	Build panels or other integrations to aid your Photoshop workflow. Store local data.	Intermediate	JavaScript, HTML, CSS	22.0	Yes	CCX
UXP Hybrid Plugins	Build native apps with UXP components.	Advanced	C++, JavaScript, HTML, CSS	24.2	Yes	CCX
Photoshop API	Edit thousands of Photoshop documents in the cloud.	Intermediate	Any modern language making a REST API call	n/a	Yes	n/a
Scripting with ExtendScript	Speed up repetitive tasks.	Beginner	JavaScript (ECMAScript 3)	CS3	No	JSX
CEP Panels	Build panels or other integrations to aid your Photoshop workflow.	Intermediate	ExtendScript, based on JavaScript (ECMAScript 3), HTML, CSS	14.0 to 20.0	No	ZXP
C++ SDK	Build powerful and fast plugins.	Advanced	C++	CS3	No	DLL, EXE, DMG
Generator	Generate image assets automatically, send messages to other apps, or control Photoshop.	Intermediate	JavaScript, NodeJS	14.1	No	n/a

Table 2.1: “Which one is right for me?” comparison on Adobe’s website [37]

Adobe named the Common Extensibility Platform); in other words, any time a plug-in is involved in displaying a dialog or a panel, the main application has to load a new web browser just for it, a process that might take seconds, stalling the user experience [2]. Such a design is only viable on a client computer serving a single user, being very expensive and unreliable to operate in a server context serving simultaneously connected users, unlike the proposed solution of the JSON HTTP Notation. The original C++ implementation, while certainly fast to execute and flexible in its capabilities, presents challenges of its own.

PICA, or the Plug-In Component Architecture, is Adobe's legacy plug-in sys-

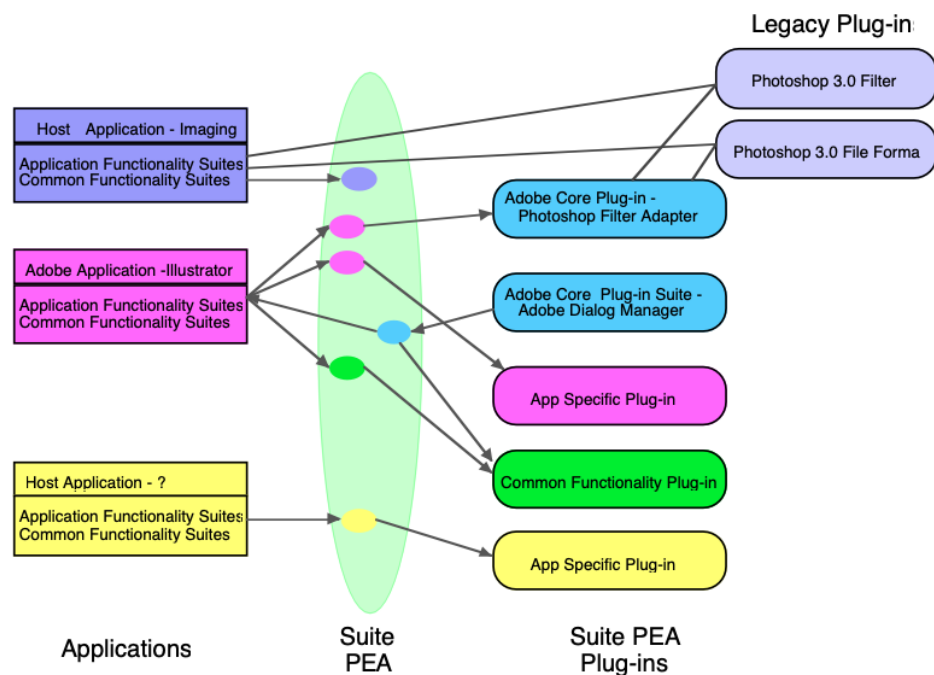


Figure 2.2: Adobe's Plug-In Component Architecture, circa 1997 [71]

tem for its software suite in the 1990s. The software came bundled with a series of C++ header files, like `PITypes.h` supplying an abstraction layer on top of the system's primitive types (simplifying the creation of cross-platform plug-ins across Macintoshes and Windows PCs), or `DialogUtilities.h` providing the means to displaying dialog boxes. At the time of launch of the main application, it searches for plug-ins in the directory (and its subdirectories) specified in the configuration files and loads them in memory. In addition to its code, a plug-in comes with additional resources in its PiPL file, standing for Plug-In Property List, such as its metadata and static properties to operate its execution.

Listing 2.2: Prototype main entry point of a native Photoshop Plugin [75]

```
1 #if MSWindows
2 void ENTRYPOINT (
3     short selector,
4     void* pluginParamBlock,
5     long* pluginData,
6     short* result);
7 #else
8 pascal void main (
9     short selector,
10    Ptr pluginParamBlock,
11    long* pluginData,
12    short* result);
13 #endif
```

When requested, like selecting an option from a menu, the main program calls a main routine that all plug-ins must implement: it receives from the main program a code indicating the operation requested and a pointer to a location in memory where the actual parameters reside; finally, pointer to a return numeric value to

communicate to the parent application a successful execution or the encounter of errors. Due to their native nature, C++ PICA plug-ins can interact with a wide range of functionality in Photoshop, such as Import/Export modules to bridge devices and compressed file formats not supported by the program or the operating system, Filters to modify the selected portion of the image, and Automation modules to script virtually any command in the program. Like the generic capabilities mentioned earlier, header files provide access to those specific modules. Besides the complexity inherent in a lower-level programming language, plug-ins loaded in the address space of the host application have detrimental consequences to the stability of the combined solution, create hindrances to updating the software to its latest version, and would be impractical to support across a wide range of platforms: the latter is exemplified by the fact that Adobe, as it ported Photoshop to mobile devices and the Web, did not implement the PICA architecture from the PC.

To alleviate some of the performance drawbacks of the ExtendScript/CEP ar-

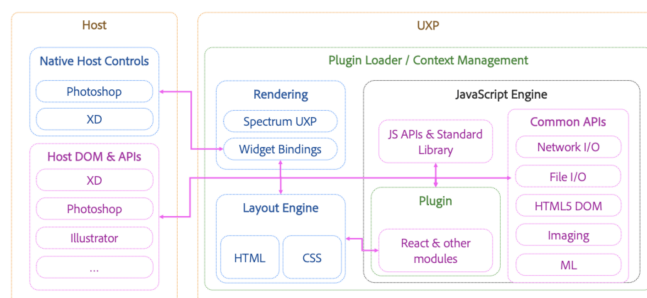


Figure 2.3: Adobe's Unified Extensibility Platform, circa 2019 [5]

chitecture, Adobe introduced the Unified Extensibility Platform in 2019, which replaces the aging JavaScript implementation with the modern V8 JavaScript runtime, the same used by the Chrome web browser, and reduced the supported HTML/CSS surface of the DOM so that it could be feasible for the company to replace the need for a complete browser instance with a custom-drawn native user interface, similarly to how the React Native framework allows the use of HTML to describe the UI of a native mobile application.

Listing 2.3: Example manifest of a UXP Photoshop Plugin [3]

```
1 {
2   "manifestVersion": 5,
3   "id": "YOUR_ID_HERE",
4   "name": "Name of your plugin",
5   "version": "1.0.0",
6   "main": "index.html",
7   "host": {
8     "app": "PS",
9     "minVersion": "23.3.0"
10  },
11  "entrypoints": [
12    {
13      "type": "command",
14      "id": "commandFn",
15      "label": {
16        "default": "Show A Dialog"
17      }
18    },
19    {
20      "type": "panel",
21      "id": "panelName",
22      "label": {
23        "default": "Panel Name"
24      },
25      "minimumSize": {"width": 230, "height": 200},
26      "maximumSize": {"width": 2000, "height": 2000},
```

```

27         "preferredDockedSize": {"width": 230, "height": 300},
28         "preferredFloatingSize": {"width": 230, "height": 300},
29         "icons": [
30             {"width":23,"height":23,"path":"icons/dark.png",
31               ↪ scale":[1,2],"theme":["darkest","dark"]},
32         ]
33     },
34     "icons": [
35         { "width": 23, "height": 23, "path": "icons/icon_D.png", "
36           ↪ scale": [ 1, 2 ], "theme": [ "dark", "darkest" ], "
37           ↪ species": [ "generic" ] }
38     ],
39     "requiredPermissions": {
40         "network": {
41             "domains": [
42                 "https://adobe.com",
43             ]
44         },
45         "clipboard": "readAndWrite",
46         "webview": {
47             "allow": "yes",
48             "domains": [ "https://*.adobe.com", "https://*.google.com
49               ↪ " ]
50         },
51         "launchProcess": {
52             "schemes":
53                 [ "https", "slack" ],
54             "extensions":
55                 [ ".xd", ".psd" ]
56         },
57     },
58 }

```

Similarly to the PiPL files of native plug-ins, UXP plug-ins come with a manifest file to describe its properties, but unlike the older architecture it accounts for a flexible system of permissions: accessing the network or displaying a web page in a

web view requires restricting it to a set of domains specified in the manifest; read/write access to the clipboard and the file system is strictly guarded, as is launching other applications. UXP plug-ins are more limited in how they can extend their host compared to native ones, as they can only populate menu entries and side panels in the program [17]. To access the functionality exposed by the parent application, a dynamic import statement is used, using Node.JS `require` syntax: for example, to access all the currently opened documents, the following instruction will do so: `const allDocuments = app.documents;` [36]. Asynchronicity is accounted for in the newer paradigm, as any operation that alters the document must be mutually exclusive and potentially long-lasting: such functions must be invoked on the main imported object through its `executeAsModal` method and can progressively update the progress of its operation [1]. This extensibility model has been adopted by Adobe for all their creative software solutions, and the pattern more generally has found use in the creation of browser extensions, as seen in the WebExtensions standard [14] and in code editors, like Microsoft Visual Studio Code [24] and Panic's Nova [31]. Although popular, there are still issues to be raised about the choice of JavaScript, especially a modern version of the language with a lot of capabilities to provide support for:

- Being an interpreted, dynamic language, once the API contract is made with third-party developers, the application must continue to bundle a full

JavaScript runtime to not break compatibility with existing plug-ins, and a language change would require extensive rewrites.

- Due to the complexity of a JavaScript runtime, it is impractical to write a new one from scratch, even for supporting the most basic of features.
- The perceived simplicity of the language syntax is mostly due to the dynamic type system that is extremely forgiving in casting operations and to the large number of online tutorials and sample codes resulting from being the only permitted language in web browsers. There is no reason to assume that a user, even if interested to use their personal computer to its fullest potential, would consider using any general-purpose programming language for writing a plug-in to aid their work unless already inclined to be a programmer. In the latter case, that user would be better off in a stricter programming environment, where the presence of a compiler helps them find the easiest mistakes before even encountering them at runtime.

For merely connecting the host application to third-party services for import/-export purposes, the declarative nature of the proposed JSON HTTP Notation is easier to approach for newcomers to programming, safer for the hosting environment, and feasible to implement from scratch and support in the long term. As for data processing, the emerging WebAssembly standard [62] and its proposed System Interface [25] enable the efficient and safe execution of almost any

programming language in a portable and near-native execution environment. In addition to that, since it only defines a low-level binary instruction format, it does not constitute an API contract with plug-in developers. Future versions of the JSON HTTP Notation specification might account for WebAssembly extensions to allow for processing tasks.

Chapter 3

The JSON Data Interchange Syntax and the Swift Programming Language

The JSON HTTP Notation solution presented in this work comprises a language specification, a runtime serving as its reference implementation, and a simple development environment for macOS.

The JSON Data Interchange Syntax is integral to the design of the JOHN Language. JavaScript is one of the world's most popular programming languages [45], a direct consequence of being the only one capable of manipulating the Document Object Model of web pages. One key characteristic of the language is that while it is object-oriented and deliberately inspired by the Java syntax, unlike the latter, it does not feature class-based objects [19], even though the sixth edition of the language addressed the lack of class declarations altogether. In JavaScript, instead of having a conceptual separation between classes carrying the structure (inheritance) and the behavior (methods) of objects and the latter holding the state

only, in the absence of classes, both are carried by and inherited from the objects themselves. Objects can be constructed manually either via literals (the same way one would declare a string or a boolean value) or prototype constructors (invoked with the familiar **new** syntax) and may contain zero or more properties that can be added or removed even after the object's creation. Each property accepts either primitive values (including **null** and **undefined** for expressing its absence) or other objects (including arrays). JavaScript's primitives do not include characters or integers of varying length: the former are treated as part of strings, while the latter are always IEEE 754-compliant 64-bit floating point numbers. With most general-purpose programming languages incorporating elements of object-oriented programming, and with JavaScript presenting a straightforward implementation, around the turn of the century, they started to be informally used as an interchange format in client-server applications, as a simpler alternative to XML, with the first formal definition published in 2001 [20].

Listing 3.1: An example JSON document describing a blog entry

```
1 {  
2   "title": "Sample Article",  
3   "author": {  
4     "name": "Mario Rossi",  
5     "email": "mario_rossi@example.com"  
6   },  
7   "snippet": "Lorem ipsum dolor sit amet adipiscing elit",  
8   "web_link": "https://www.example.com"  
9 }
```

A JSON document features a single top-level object literal, with its possible

contents being those of a JavaScript object literal as defined in the third edition of the language: recent additions in the JavaScript syntax, like `BigInt` for storing numeric values that cannot fit in 64 bits, are not supported in JSON. A crucial aspect of the syntax is the exclusion of comments: this is by design, as allowing them would permit custom parsing directives, breaking the universality of the serialization format. Choosing such a universal format as the fundamental syntax for JOHN instead of building new constructs reduces the scale of work required for building a parser while incentivizing a healthy ecosystem of multiple competing implementations. Also, since the syntax of JavaScript objects is common in most C-like programming languages, it makes the language more approachable to experienced software developers while providing valuable skills for first-time programmers, repurposable in other programming domains.

The actual implementation of the JOHN specification in a library and development environment uses a language that, similarly to JavaScript, enjoys a unique advantage enshrining its position as one of the world's most used programming languages, and that is Swift [57]. Swift is the object-oriented and open-source programming language primarily contributed to by Apple to develop its software platforms. What sets Swift apart is that it has been designed as a systems programming language while finding use for crafting front-end applications: it strives to provide high performance while also catering to the ergonomics associated with

more dynamic and higher-level languages. Swift’s approach to memory management best exemplifies this careful balance: for safety and convenience purposes, the language manages it automatically, but to ensure fast and deterministic performance, it does not use a system of “garbage collection” but relies on Automatic Reference Counting [12]. Automatic Reference Counting works with the compiler keeping a counter inside the object’s reference and inserting, as it compiles the program, instructions to increment and decrement it: the first occurs any time the source code assigns the object reference to a variable; the latter when the block of code holding the variable reaches completion; when decrementing the counter brings it to zero, the referenced instance gets deallocated from the heap. The compiler, built with LLVM [56], is very involved during the writing of Swift syntax, as it allows the developer to mostly ignore the complex and strict type system that underlies the language, with type inference making sure that the program is correct. The developer, however, is not dispensed with the task of handling the possibility of values being absent unexpectedly, which is why the language mandates the initialization of all variables and constants before using them, and if their type is optional (meaning it accepts null values), before the property can be read, it also requires the use of a `guard` statement to verify the presence of any values. While formally object-oriented, using classes and reference types to model the data required by the application is discouraged, instead advocating for what

the project refers to as protocol-oriented programming [69]:

- Usage of value semantics (structs) in place of reference semantics (classes) to reduce the potential of bugs and increase the performance of the software; the former, because value types are immutable, eliminating the possibility of unintended side effects in manipulating them; the latter, as the lifecycle of the data structure is contained to the block of code that initialized it, allowing it to be stored in the stack area of the system's memory, which makes creating and destroying it computationally cheap, and eliminates the need to deploy Automatic Reference Counting [73].
- Reuse of code via composition of many protocols instead of multiple levels of class inheritance; this enables better reasoning of the structure of the program because there cannot be superclasses and subclasses able to alter the abstraction crafted by the protocols; the code is reusable by specifying the implementation of methods inside so-called protocol extensions, which is what sets apart Swift's protocols from interfaces featured in other object-oriented programming languages [72].
- Preference of generic methods over polymorphism; making a method generic over a protocol enables the type inference system to resolve the function dispatch at compile time instead of requiring a table look-up at runtime to pick the appropriate method implementation [73].

The unique combination of a predictably performant language, ergonomics in the design of its syntax, and the wide range of supported platforms made the choice of Swift obvious for the project that inspired the creation of JOHN, i.e., the University Cloud Suite. In turn, it was the fact that the reference implementation of the JOHN specification happened to be a Swift library that made the creation of a macOS development environment for JOHN plugins, the JOHN Editor, only a matter of assembling a GUI front-end to the library itself, which would not have been a justifiable time investment otherwise.

3.1 Parsing JSON Objects with the Swift Codable Protocol

In early versions of the Swift language, its evolution efforts revolved around settling the basic constructs of the syntax, relying on the API bridged from the previous Cocoa/Objective-C ecosystem. Parsing JSON objects received during communications with servers being frequent in client applications, the Cocoa framework has long provided a mechanism to do so in the form of the `JSONSerialization` object [28]. When calling the `jsonObject(with:)` method with a valid JSON source string (encoded in either UTF-8, UTF-16, or UTF-32), the parser generates a type-erased dictionary or array. Swift, unlike Objective-C, is a type-safe language, which means working with loosely-typed JSON objects and type-erased dictionaries generated by the parser is cumbersome and error-prone.

Listing 3.2: Accessing a property decoded from a JSON source with JSONSerialization

```
1 let json = "{\"title\": \"Sample Art...\".data(using: .utf8)!
2 if let entry = try? JSONSerialization.jsonObject(with: json) {
3     // Getting the author's email address
4     if let author = (entry as? [String: Any])?["author"] {
5         if let email = (author as? [String: Any])?["email"]
        ↪ as? String {
6             print("The author's email address is \("
        ↪ email)")
7         }
8     }
9 }
```

With the introduction of the Codable protocol [16], conforming a struct definition to it enables the compiler to synthesize the property names and the functions used to encode and decode the data. In what is the most concise demonstration of what protocols are capable of in Swift, a single method call on the `JSONDecoder` class produces a type-safe and ergonomic accessor to the properties of the JSON object, and, on top of that, it verifies that the properties expected to be present are there and with the correct type.

Listing 3.3: Accessing a property decoded from a JSON source with the Codable protocol

```
1 struct Entry: Codable {
2     let title: String
3     struct Author: Codable {
4         let name: String
5         let email: String
6     }
7     let author: Author
8     let snippet: String
9     let web_link: URL
10 }
```

```
11 |
12 | let json = "{\"title\": \"Sample Art...\".data(using: .utf8)!
13 | if let entry = try? JSONDecoder().decode(Entry.self, from: json) {
14 |     // Getting the author's email address
15 |     print("The author's email address is \"(entry.author.email)\"
    |     ↪ )
16 | }
```

Codable’s automatic behavior is generally effective, but when it is not apparent how to bridge between the conventions of Swift and those of JSON, the need to override the synthetic methods arises [21]. The most obvious example is the conversion between the “snake_case” convention of multi-word JSON properties and the “camelCase” convention employed in Swift source code. While it is possible to declare variable names in Swift containing the underscore character (and any other Unicode symbol, including emojis), a more desirable approach is to override the property keys stored inside the `CodingKeys` enumeration, which the decoder uses to parse the JSON source.

Listing 3.4: Overriding the compiler-generated `CodingKeys` enumeration

```
1 | let webLink: URL
2 | private enum CodingKeys: String, CodingKey {
3 |     case title
4 |     case author
5 |     case snippet
6 |     case webLink = "web_link"
7 | }
```

Similarly, it is possible to override the decoding process by overriding the initializer provided by the protocol, for example, to handle a property that can assume several types or for those that lack equivalents in JSON, like enumerations with as-

sociated values. The decoder parameter supplied to the initializer is a container for the parsed JSON data that is queryable to decode primitive types or custom types conforming to the Codable protocol. It is also possible to obtain sub-containers from the main one: they are of the “unkeyed” kind for getting access to arrays or “keyed” for decoding using the `CodingKeys` defined above.

Listing 3.5: Overriding the initializer generated by the compiler

```
1 public init(from decoder: Decoder) throws {
2     do {
3         // If a single string value is found, it is the email
4         // ↪ address
5         email = try decoder.singleValueContainer().decode(
6             // ↪ String.self)
7         // Obtaining the author's name from the email
8         name = email[..<(email.firstIndex(of: "@") ?? email
9             ↪ .endIndex)]
10                .replacingOccurrences(of: "_", with: " ")
11                .replacingOccurrences(of: " ", with: " ")
12                .replacingOccurrences(of: ".", with: " ")
13                .capitalized
14    } catch {
15        // Otherwise decode the struct
16        let container = try decoder.container(keyedBy:
17            ↪ CodingKeys.self)
18        name = try container.decode(String.self, forKey: .
19            ↪ name)
20        email = try container.decode(String.self, forKey: .
21            ↪ email)
22    }
23 }
```

3.2 Swift Non-Blocking IO and Async HTTP Client

The optimal usage of a server's resources means maximizing the number of clients handled simultaneously. The textbook implementation of socket listening instantiates a new thread dedicated to each incoming client request, utilizing blocking read and write system calls to transfer data until the connection is closed, gracefully or not. The problem with this approach is that thread creation is an expensive operation: for instance, on a BSD-like system like macOS, each new thread involves allocating a default of 512 kilobytes (and no less than 16) of memory for serving as the stack, besides that required by the kernel to operate the thread and takes many microseconds [58]. This kind of performance optimization would be excessive in small systems, but it is critical when the number of users grows. High-performance and scalable web services employ a technique known as NIO for Non-Blocking IO, which means realizing an abstraction layer on top of a pool of threads for sharing among all the clients and employing the kernel event polling interfaces provided by the operating system, to check the availability of data before performing a blocking system call. When using the Swift language for server-side development, the Swift-NIO library [52] makes building this architecture easier by supplying the abstractions on the system's resources (`EventLoops` for threads, `EventLoopGroups` for pool of threads, `Channels` for sockets), efficient data structures to make the most of the server's available memory (`ByteBuffer` providing

a copy-on-write buffer that recycles the memory after read operations), and interfaces to implement the processing logic required by the networking protocol used in the communication (`ChannelHandlers` that can be chained one after the other to implement various functionality, such as encryption, encapsulation of packets, or even path routing). For proprietary networking protocols, Swift-NIO is used to directly implement them, while the Swift community maintains `ChannelHandler` implementations for established standards like HTTP [48], TLS [50], and IMAP [49].

Since connecting to web services is frequent in servers, the Swift Server Workgroup [54], in addition to NIO, maintains an even higher-level abstraction called the Async HTTP Client [10], a library for making HTTP requests in Swift, using the previously mentioned Non-Blocking architecture.

Listing 3.6: Sending a simple POST request via the Async HTTP Client

```
1 let url = URL(string: "https://www.example.org")
2 let body = "Hello World"
3 let httpClient = HTTPClient(eventLoopGroupProvider: .createNew,
    ↪ configuration: .init(redirectConfiguration: .disallow))
4
5 var request = HTTPClientRequest(url: url)
6 request.method = .POST
7 request.body = .bytes(.init(string: body))
8 let response = try await httpClient.execute(request, timeout: .
    ↪ seconds(30))
9 for try await buffer in response.body {
10     print(String(buffer: buffer))
11 }
12 try await httpClient.shutdown()
```

A user of the library instantiates a client once (the creation of a new `EventLoopGroup`

would otherwise negate all the performance improvements of Non-Blocking IO) and then use idiomatic Swift code on it, such as the concurrency syntax for lightweight parallel programming, to request the desired resource and handle the response body chunk by chunk. The library obscures the inner workings of Swift-NIO, handles the parsing of cookies in received responses, and follows redirects automatically (unless specified otherwise, like in the example above).

3.3 Automating Initialization with Introspection and Property Wrappers

The same concerns about bridging data incoming from the network (intrinsically loosely typed) and the types defined in a Swift program discussed in the Codable section also apply to the result of HTTP requests. While an example of protocol-oriented programming, the Codable protocol¹ has capabilities not available to other protocols, as it was necessary to change the C++ source code of the compiler to enable the automatic synthesis of the decoding initializer and the `CodingKeys` set of property keys. It is possible, however, to replicate some of the ergonomics for other forms of automatic property assignments using a combination of two capabilities of the Swift language: property wrappers and introspection.

Without property wrappers, the assignment would either occur in a long and

¹The following considerations hold for versions of the Swift programming language and compilers older than 5.9, which introduced a macro capability, enabling code synthesis at the compilation stage without unique exceptions for system types.

complex initializer or deferred to each property's getter and setter, using a separate private variable as its backing store. Property wrappers [39] allow for the annotation of a property in a struct for its replacement with a separately defined data structure that provides its value, implementing custom behavior without exposing this detail at the point of use. It is possible to define a property wrapper once and use it with multiple types, including optional ones, using generics and conditional protocol conformance. In addition to the default initializer (mirrored with a plain `@annotation`), it is also possible to have multiple initializers whose parameters are specified within parentheses following the annotation, according to the function signature of the initializer.

Listing 3.7: Defining a simple property wrapper with reference semantics

```
1 @propertyWrapper
2 class Stringified<Value: LosslessStringConvertible> {
3     typealias Value = Value
4     var string: String
5     var wrappedValue: Value {
6         get { .init(string)! }
7         set { string = newValue.description }
8     }
9     init(wrappedValue: Value) {
10         self.string = wrappedValue.description
11     }
12     init(value: String) {
13         self.string = value
14     }
15 }
16 struct Result {
17     @Stringified var title: String = ""
18     @Stringified(value: "0") var views: Int
19 }
```

Initializing a property from a type-erased source requires that its underlying type is representable by it. The primitive types encountered in programming are usually perfectly representable as strings, and the JSON sources, used in the JSON HTTP Notation and most REST API endpoints, represent all types as strings, being text files themselves. Swift’s protocol-oriented programming is a solution to this problem, as most primitive types provided by the Swift standard library conform to the `LosslessStringConvertible` protocol [29], which guarantees the existence of an initializer taking a string literal as its only parameter. Utilizing a generic property wrapper that operates on `LosslessStringConvertible` types, it becomes possible to transform type-erased strings to the desired primitive type simply by annotating the property.

To further automate the synthesis for all the marked properties within a struct definition, it is necessary to automatically identify them and apply the initialization of the property wrapper to each one, and recursively if the property happens to be a struct in its own right. While the design of the Swift language encourages static programming practices, there are avenues to archive runtime dynamism, such as introspection. Introspection refers to the ability of objects to describe themselves and enumerate their properties at runtime [27]. In Swift, it is possible to reflect an object, meaning employing introspection to access its children (properties), using the `Mirror` struct [30].

Listing 3.8: Reflecting a struct using Mirror to initialize it from a dictionary

```

1  /// Mirror provides read-only access to the properties, so only
    ↳ reference types (enforced via the conformance to AnyObject)
    ↳ allows the assignment via introspection
2  protocol StringAssignable: AnyObject {
3      func assign(_ string: String)
4  }
5  extension Stringified: StringAssignable {
6      func assign(_ string: String) {
7          self.string = string
8      }
9  }
10 extension Result {
11     init(dictionary: [String: String]) {
12         self.init()
13         let mirror = Mirror(reflecting: self)
14         for child in mirror.children {
15             /// Mirror will surface all the stored properties, but
                ↳ not the computed properties
16             /// title and views are computed properties, backed by
                ↳ the _title and _views objects
17             if let label = child.label?.dropFirst(),
18                 let assignable = child.value as? StringAssignable,
19                 let value = dictionary[String(label)] {
20                 assignable.assign(value)
21             }
22         }
23     }
24 }
25 extension Result {
26     init(title: String, views: Int) {
27         self._title = .init(wrappedValue: title)
28         self._views = .init(wrappedValue: views)
29     }
30 }
31 // The resulting values are the same
32 Result(title: "Lorem Ipsum", views: 5)
33 Result(dictionary: ["title": "Lorem Ipsum", "views": "5"])

```

From there, by optionally type casting to the struct defining the property

wrapper, it is possible to initialize all the marked properties of the data structure.

3.4 Debugging Task Execution with Delegation

To be able to debug the execution of a plug-in, it is not sufficient to inspect the product of the supplied inputs, but every step of the way to measure the timing and outputs of the stages composing the plug-in. Doing this requires the runtime implementing the JOHN specification to provide hocks for the caller to insert custom logic at the various execution milestones: before and after sending HTTP requests, responses, and decoding the **Content-Type**. Many frameworks in the Cocoa ecosystem, and therefore in the Swift programming language, provide complex functionality that may require customization or observation and do so via the pattern of delegation [18]. The class providing the main functionality, which may be a web view [63] or an XML parser [64], offers a property usually named solely **delegate**. This property stores a reference to an instance of a custom class, either subclassing a delegate class or conforming to a delegate protocol, that the framework makes available. The main class does not retain, from a memory management perspective, a strong reference to the delegate instance: it is the client of the delegating object that has to instantiate and store it, as creating and assigning the delegate in a one-line instruction would result in its instantaneous deallocation, because of the way Automatic Reference Counting works. As for the delegate object, it does not need to implement all the methods defined in the

base class, but only those events it wishes to receive notifications about and the behaviors it may want to override: as the main class reaches various points in its lifecycle, it will check the existence of the method implementation it is about to invoke, and only proceed to so so if possible. While methods are not required to follow a precise naming convention, most delegate base classes do so:

- When the delegate offers the opportunity to prevent an event from occurring, the method features the verb “should” followed by the event name and returns a boolean value.
- If the main class does not offer the opportunity to prevent the signaled event from occurring, the method presents the “will” verb and no return value.
- Lastly, the delegate method uses the “did” verb when it notifies its client of the result of an operation, with the value provided as part of the arguments.

Inspired by the `XMLParser` class used in the JOHN Runtime for decoding response bodies of RSS and SOAP endpoints, the same delegation approach provides a custom `HTMLParser` library and the `DebugDelegate` that enables the construction of development tools such as IDEs for JOHN plug-ins.

3.5 SwiftUI and Result Builders

The primary use of the Swift programming language is to develop client-side applications for iOS and macOS-based devices. Although those platforms used im-

perative UI frameworks derived from NeXTSTEP’s Application Kit, the transition from Objective-C to Swift and the introduction of increasingly more constrained devices, like smartwatches, spurred the creation of a new framework. SwiftUI [55] is a declarative UI framework that defines its views via a Domain-Specific Language (DSL) in Swift, made possible by a feature called Result Builders [4]. A DSL allows embedding a rule set inside a general-purpose programming language like Swift, enabling the programmer to describe a problem domain more directly. Result Builders, combined with a capability known as “trailing closure arguments” (which is the ability to place a function literal passed as an argument to a function call outside the parentheses and even remove them altogether if it is its only parameter [15]), allow for the omission of most language statements associated to the assignment and collection of values, which is what composing the UI with a set of provided widgets is all about. The compiler will transform the expression, implicitly reinserting variable assignments, function calls to collect and process them, and return statements.

Result Builders are available to all Swift programs, just by marking any type, like structs or enumerations, with `@resultBuilder` and then implementing a set of methods that the compiler uses to collect the values:

- `buildBlock(_:)` receives a variadic list of arguments, which are the values collected via the DSL, and transforms them to another type; by specifying

what type to use for the former and the latter, possibly using protocols and generics, it is possible to decide what types the result builder will work with; it is also possible to overload this method with different arguments.

- The ability to mix and match different types in a result builder requires converting all types to one defined in the argument list of a `buildBlock(_:)`; this transformation requires the presence of the `buildExpression(_:)` method.
- Language features that interrupt the control flow (like `catch` or `break`) are not allowed inside result builders; those that merely alter it (statements such as `if`, `switch`, and `for-each`) have to be implemented explicitly by defining `buildOptional(_:)` to handle the case where the `if` statement is false (and therefore the collected value is a `nil`) or `buildEither(first:)` and `buildEither(second:)` to support switches and `else-ifs`.

Once defined, it is possible to apply the name of the data structure housing the result builder, like a property wrapper, to properties and function arguments whose type is a closure, meaning a function definition utilized as a value.

Listing 3.9: Defining a data model with `ObservableObject` and a view with the `View DSL`

```
1 class EntryStore: ObservableObject {  
2     @Published var entries: [Entry] = []  
3 }  
4  
5 struct MainView: View {  
6     @EnvironmentObject var entryStore: EntryStore  
7     @Environment(\.openURL) var openURL
```

```
8   let titleSize = 24.0
9   @Environment(\.dynamicTypeSize) var accessibilitySize
10
11   var body: some View {
12       List {
13           ForEach(entryStore.entries, id: \.web_link) { entry in
14               Button {
15                   openURL(entry.web_link)
16               } label: {
17                   VStack {
18                       Text(entry.title)
19                           .font(.system(size: titleSize + (
20                               ↪ accessibilitySize == .large ? 12.0 :
21                               ↪ 0)))
22                       Text(entry.author.name).font(.footnote)
23                   }
24               }
25           }
26       }
```

In SwiftUI, a view is a struct conforming to the View protocol [60], whose only requirement is the implementation of a body property, which is where the result builder is implicitly applied. Inside the body property, it is possible to combine any kind of value conforming to the View protocol, both system-provided widgets and custom-defined views. The appearance and behavior of views, being structured data, is customizable either by supplying parameters to their initializer or by applying so-called “modifier-style methods” which is a method (in this case either specific to a concrete view or applicable to any View through the use of generics) that instead of mutating the structure in place, returns the new modified

value instead; the latter are usually preferred because they can be chained one after the other in any desired order, unlike function arguments. Being a struct definition, it can contain any number of properties and, since a result builder accepts valid Swift code, they are available in the body property to use their values in the UI to render. A struct's properties do not notify SwiftUI that they have changed and that the view's rendering is now invalid: this works well for constants, but variables, expecting to be changed after the view's initial render, have to be marked with either two property wrappers that fill in this essential functionality: `@State` when first initializing a piece of data dependency [46], and `@Binding` when accessing a writable reference to a state variable defined higher in the view hierarchy [13]. A `@State` value is handed to a `@Binding` lower in the hierarchy by passing the underlying struct backing the `@State` property wrapper directly to the lower view as an argument in its initializer. By default, property wrappers do not expose the underlying struct: the `@State` and `@Binding` definitions feature a `projectedValue` property containing the struct, accessible by prepending the variable name (where the wrapper is applied) with a dollar sign `$` [38]. If the value held by a `@State` property is either a string or an integer or conforms to the `RawRepresentable` protocol [40] with either type as its raw value (this protocol allows for the initialization from it and the conversion to it), then by replacing the property wrapper with `@AppStorage` [9] or `@SceneStorage` [44] and supplying a

unique string value as its key within parentheses, it is possible to persist the value across executions, respectively globally to the application or for each window. *The following used to be true for versions of the Swift language and compiler older than 5.9, which introduced a macro system and consolidated state management for both value types and reference types under the `@State` property wrapper.* `@State` and `@Binding` work on value types, meaning structs and enumeration defined by the programmer or primitive types of the Swift standard library. However, complex applications may prefer to model their data as a globally unique object, requiring reference semantics. Classes have to explicitly opt-in participation in SwiftUI's change detection system by first conforming to the `ObservableObject` protocol, then marking each property whose change should invalidate a view's rendering with the `@Published` property wrapper [32]. Because the variable storing an object keeps only a reference, which does not change as the `@Published` properties mutate, it is not possible to use `@State` and `@Binding` with objects. A data dependency with a reference type is defined at the top of the relevant view hierarchy by marking the variable with the `@StateObject` property wrapper [47]. If the object's reference is kept strongly somewhere else (for example by implementing the singleton pattern in its class) it is possible to make its reference in the view weak by using the `@ObservedObject` property wrapper instead [33]. To pass an object to the views below the one holding its reference, in place of `@Binding` there is the

`@EnvironmentObject` property wrapper, which instead of requiring to be passed to each layer of subviews manually through their initializers, is passed by applying the `.environmentObject(_:)` modifier-style method [23]; this convenience means however that the Swift compiler is not able to verify the correct initialization of an `@EnvironmentObject` property, and failing to do so will result in an application crash upon attempts to display the view. A similar mechanism exists for value types, both system global variables (like the user’s preferred accent color or accessibility settings) and custom-defined values, using the `@Environment` property wrapper and the `.environment(_:, _:)` modifier [22]. While `@EnvironmentObject` can acquire the reference by the type alone, `@Environment` values require the specification of a key as the first argument of its initializer and the modifier. The key is type-safe because it is defined as a structure conforming to the `EnvironmentKey` protocol and then inserted in a system-provided type, `EnvironmentValues`, via an extension to enable the code completion system to suggest it at the call site.

Listing 3.10: Defining an app with a single window type with the `Scene` DSL

```
1 @main
2 struct MyApp: App {
3     @StateObject var entryStore = EntryStore()
4     var body: some Scene {
5         WindowGroup {
6             MainView().environmentObject(entryStore)
7         }
8     }
9 }
```

In addition to the `View` protocol and DSL, the SwiftUI framework also sup-

plies an App protocol [7] and Scene DSL [43] to express the windows, file associations, menu bars, and preference panes of applications using the same declarative paradigm.

Chapter 4

JOHN: the JSON HTTP Notation

The JSON HTTP Notation is a system made of a language definition, a reference implementation of the runtime, and a development environment to ease the writing and testing of plugins. This chapter will go into detail about each one.

4.1 Specification of the JOHN Language

The JOHN Language extends the JavaScript Object Notation syntax by defining a set of keywords and operators with their prescribed behavior.

A JOHN plug-in, meaning a source file written using the JOHN Language, is a valid JSON file: a plugin writer uses the constructs of the language inside **string** literals, as defined in ECMA-404 [20], either as the name of an object member or its value.

In addition to strings, the JOHN Language may employ **number** or **boolean** types for its values, also defined in the standard [20].

The preferred file extension for JOHN Language sources is `.john` although the standard `.json` extension is also valid.

Implementations of the JOHN Language should parse the JSON source file, construct a JOHN data structure in memory (verifying the correct use of its properties in the process), and finally attempt its execution. The latter boils down to sending HTTP requests and managing the objects served as input to those requests and obtained from them. After the series of requests specified in the plug-in (if any) is carried out without issues, the runtime constructs an output object, transforming the results following the format defined in the plug-in.

From here on out, the specification refers to those who create and maintain plug-ins using the JOHN Language as “plug-in authors”, and individuals and organizations running them via a runtime implementation of the specification as “plug-in operators”; since “user” can ambiguously refer to both users of the language, users of the runtime, and users of the complete software solution, it is best to avoid its usage.

4.1.1 Anatomy of a JOHN Plugin

All valid JOHN Language sources must contain a root JSON object with an `about` member detailing its metadata, a `pipeline` member holding an array of requests to make, and a `result` member specifying what data structure to return to the caller.

Listing 4.1: Example of a simple JOHN plug-in

```
1 {  
2     "about": {  
3         "name": "Example Plug-In",  
4         "version": 1,  
5         "protocol": "JOHN_Simple_Example_Plugin"  
6     },  
7     "parameters": {},  
8     "pipeline": [  
9         {  
10             "url": "https://www.example.com"  
11         }  
12     ],  
13     "result": "$1"  
14 }
```

About

The **about** member accepts **About** objects containing metadata about the plug-in intended to aid the plug-in operator in identifying and assigning the correct plug-in to its intended task. This member is mandatory for the JOHN root object. Valid **About** objects must contain:

- A **name** string contains the human-friendly name of the plug-in. Its principal intended usage is identifying the plug-in inside a preference pane in a GUI.
- A **version** number indicates the unique build number of a plug-in. The plug-in operator may use it when upgrading a plug-in to newer versions.
- A **protocol** string uniquely identifies a plug-in among related ones. It enables a plug-in author to upload a group of plug-ins together and have the

receiving party be able to sort them automatically. It is not a globally unique identifier across all possible plug-ins. While it is certainly possible to use a reverse domain notation, it is not required.

The variable subscript syntax

Each HTTP request described inside the `pipeline` array produces a response including several HTTP headers and a response body decoded according to the `Content-Type` header sent by the source server. Anytime a property of either the `pipeline` or the `result` accepts a `string` value on its right-end side, it is possible to interpolate the value of previous requests inside the string by using a dollar sign `$` followed by the number of the request, starting at 1. When the value referenced by the variable is not a single value (a `string`, `number`, or `boolean`) but a vector or a hierarchy, a subscript syntax is available using a pair of square brackets `[]` and, respectively, an `integer` or a `string` inside them.

Listing 4.2: The response body used in JOHN subscript syntax example

```
1 {  
2     "items": [  
3         { "name": "Luca", "age": 56 },  
4         { "name": "Mario", "age": 37 },  
5         { "name": "Anna", "age": 16 },  
6         { "name": "Luisa", "age": 34 },  
7         { "name": "Flavio", "age": 73 },  
8         { "name": "Simona", "age": 89 }  
9     ]  
10 }
```

For example, if the second request in the pipeline of a JOHN plug-in were to

produce a response body of a list of people inside an array, in turn, nested in a property, then, from the third request of the pipeline onwards, it is possible to access the name of the sixth person using `$2[items][6][name]`.

Each node of the output, whatever the root variable or one of its subscripts, is simultaneously capable to answer single-value requests and subscripts: this allows the runtime to augment a single value with extra properties, or to surface complex structures such as XML-like markup.

When the syntax is used to interpolate the value of the subscript with other components of the containing source string, only the single-value, if present, is used. Hierarchical and vector values are ignored.

4.1.2 Customizing the HTTP request

Valid JOHN plug-ins must contain a `pipeline` member accepting an array of `Stage` objects: it can be empty if the desired result of the plug-in is static, and sending any HTTP request is not required.

Stage

At a high level, ignoring, for now, advanced directives provided in the JOHN Language, a stage is effectively a single HTTP request and therefore only requires a `url` field for the `string` value of the resource to connect to; by default, JOHN implementations should assume a `GET` method and yield the response body as the stage output, decoded automatically according to its `Content-Type` response

header.

It is possible to customize the properties of the HTTP request, as defined in the standard [74], using the following optional properties:

- The `method` property is a case-insensitive `string` value for the HTTP method to use when sending the request. Implementations should uppercase it before use.
- The `headers` property accepts key-value pairs of strings (meaning a JSON object holding only string values) sent as HTTP headers when executing the stage.
- The `query` property also accepts key-value string pairs and acts as a convenience to append the specified query parameters to the URL of the stage. Implementations are not required to guarantee the uniqueness of query parameter names as they get appended to the URL.
- The `authorization` property accepts one of three `string` values: `"basic"`, `"bearer"`, or `"none"`, with the latter one equivalent to providing no property at all. This property serves as a convenience for the plug-in operator to specify either the HTTP `Basic` or `Bearer` authentication separately from the plug-in and other parameters. The runtime implementation shall provide the operator with a way to supply a username-password pair or a text token in

requesting the execution of a plug-in. When encountering an `authorization` property during execution, the runtime first verifies the presence of the required values and then constructs the appropriate `Authorization` header according to the specification. If the verification fails, the runtime aborts execution.

- The `body` property accepts either a single string value or a key-value dictionary as in headers and query parameters. In the first instance, the string is used as-is for the HTTP request body: if the text represents any encoding, it is on the plug-in author to specify it in a custom `Content-Type` header. In the latter case, by default, the implementation must encode it as a JSON object and set the `Content-Type` header accordingly.

4.1.3 Overriding default policies

So far, two default behaviors of implementations of the JOHN language have been presented without also discussing how to override them:

- The default encoding of key-value pairs in the request body to JSON objects
- After receiving the response to a request, in the absence of other directives in the stage, by default, its body (if present) is decoded according to the `Content-Type` header and stored for interpolation via the subscript syntax.

It is also worth mentioning that the HTTP status code associated with the

request is, by default, ignored concerning the processing of the response body. It is possible to change those behaviors by using the following properties of the **Stage** object:

- The **status** property accepts an array of integers representing the valid HTTP status codes for the response to be considered correct. If a stage includes this property and does not contain the returned response's status code, then the JOHN implementation must abort the execution of the plugin.
- The **yield** property accepts one of three string values "body", "headers", or "cookies". In the latter two instances, the runtime shall make available the key-value pair of either headers or cookies received as part of the response in place of the body when using the variable subscript syntax for the corresponding stage.
- The **encode** property accepts either "json" or "form" as its string value. The first one is equivalent to having no property at all. The latter one will encode the body (if provided as a JSON key-value pair object and not a string) using the same encoding as that used for forms submitted from HTML pages: each key and value is percent-encoded to escape reserved entities, the two concatenated with an equal sign; finally, all the pairs are in turn concatenated with each other, separated by an ampersand symbol.

Before sending the request, the runtime sets the `Content-Type` header to `"application/x-www-form-urlencoded"`

- The `decode` property acts on the received response body and accepts one of the following values:
 - `"auto"` which is the default behavior and picks one of the other ones based on the `Content-Type` header.
 - `"raw"` which means plain-text; this is the default for when no appropriate `Content-Type` is present.
 - `"json"` or `"form"` which should produce a key-value dictionary accessible via the subscript syntax; they are picked by default respectively for `Content-Type` `"application/json"` and `"application/x-www-form-urlencoded"`.
 - `"xml"` or `"html"` constructing a composite structure where the integer subscript accesses the children of an element and the string subscript accesses its attributes; these are the default behavior of `Content-Type` matching exactly `"text/xml"` or `"application/xml"`, or containing the `"+xml"` suffix for `"xml"`, while values of `Content-Type` matching `"text/html"` or `"application/xhtml+xml"` will default to `"html"`.
 - `"soap"` which works analogously to `"xml"` but unwraps the SOAP [\[67\]](#) envelope to yield its body as the root element, reducing the depth of

- the subscript used to traverse it; this is the default decoding strategy for a `Content-Type` header equal to `"application/soap+xml"` or just `"application/soap"`.
- `"xml-json"` or `"soap-json"` which attempt to parse every text node of the markup as JSON and inserts the decoded object in the structure, available to the subscript syntax; this behavior must be opted in by using the `decode` property and it is never applied automatically.
 - `"base64"` which encodes the output as a base64 encoded string, and is the default behavior for `Content-Type` equal to `application/octet-stream` or beginning with either `audio/`, `image/`, `video/` or `font/`.

It follows that JOHN implementations must support decoding for all of them.

Redirection

While the status code, unless specified otherwise, is ignored when processing the body, it is not when involving `3xx` status codes, as they indicate redirects. By default, the runtime implementation shall attempt no more than five requests when following the URL specified in the `Location` header, and the plug-in author can override this number using the `redirects` property of a `Stage` object, which accepts an integer value. It is possible to set it to 0, and in that case, no redirect following will occur: this is useful if the output desired from the request is precisely the `Location` header.

4.1.4 Manipulating the cookie store

It is not sufficient for JOHN runtime implementations to parse the **Set-Cookie** header returned as part of the response to yield "cookies" when desired in a plug-in stage. Implementations must implement the complete HTTP state management mechanism as defined in RFC 6265[68]:

- For the duration of a plug-in execution, a cookie store must be kept, similar to the authorization parameters and the output of the various stages.
- When sending a request, and before applying the headers specified for the stage, they must assemble a standards-compliant **Cookie** header.
- After receiving a response, even in the middle of following redirections, they must parse the **Set-Cookie** headers and change the cookie store by adding, updating, or removing cookies.

Cookies

The Internet standard prescribes an algorithm to determine what cookies to send based on domain, subdomain, and path [68]. The JOHN Language provides a **cookies** property as part of the directives of stages to manipulate this subset of cookies before executing the request:

- It is possible to assign it a boolean value, where **false** removes all the cookies that would have been sent from the store, while **true** is the default behavior

that does no manipulation at all.

- Alternatively, it is possible to assign it a key-value pair where the left-end side is a string containing a particular cookie name, and the right-end side is either a string to set as its value or a boolean to apply the deletion on a per-cookie basis.

4.1.5 Providing and verifying input with parameters and assertions

Complex plug-ins involve multiple stages that rely on the results of the previous ones in the pipeline. If they manipulate data, they might require input information and ways to check along their execution that those values are present and correct.

Parameters

The variable interpolation syntax starts counting stages from 1 because index 0 points at parameters. Implementations of the JOHN Language must allow the plug-in operator to supply either a string value (or types that are representable losslessly by a string) or any structure traversable via the integer and string subscript. Plug-in authors can explicitly set out the inputs they require via the `parameters` member of the root JOHN object, which accepts a `Parameter` value type either as a single value or recursive key-value tree structure; the `Parameter` can be:

- A boolean type that represents whatever the single-value parameter or the particular key in the tree structure is required for execution or not. In the

former case, if the parameter is not provided, then the plug-in should not be executed.

- A string value matching either `"username"`, `"password"`, or `"token"`, which instructs the runtime to substitute the parameter with the respective value, as defined for the `authorization` directive of a Stage.
- Any other string value, that shall be interpreted by the runtime as a default value used for the parameter if no other is provided.
- A JSON object whose keys represent nested parameters of the current parameter, either top-level or another nested one. Implementation should check for the input to contain all the keys that are either set with a `true` requirement or recursively containing a nested object.

Assertions

If the interpolation of a variable in a string parameter of a stage fails, then the execution is aborted. Plug-in authors can explicitly check for the presence of values at specific variable subscripts before a stage is executed, and even test for its value being included in a set of strings by including the `assert` property in the stage, which is a key-value pair where the key indicates the variable subscript to test for, and the value is an object containing:

- An `exists` property that accepts an optional boolean value that can either

check for the variable substitution to succeed or fail.

- A **contained** property that accepts an array of strings and will succeed the assertion if the value resolved by the variable is also a member of the array.

If an assertion fails in the context of a stage, its execution shall stop. If the stage has not run at least once then the whole plug-in should abort execution, as no suitable value for its stage output is available.

4.1.6 Automating pagination with repetition and merging

Usually, HTTP endpoints avoid returning large numbers of items in a single request, preferring a form of paginated access. The JOHN Language provides ways to indicate the runtime to execute a stage more than once and to continuously merge their outputs so that they appear to subsequent pipeline stages as if the remote resource returned them all at once.

Repetitions

Plug-in authors can express their intent to loop a stage by providing a 'repeat' property inside it:

- Its **start** number value indicates the integer index to start counting at. It is optional and defaults to 0. It must be a positive number.
- Its **step** number value indicates how many integer units to increment the counter after each iteration. It is optional and defaults to 1. It must be

greater than 0.

- Its **stop** number value is the integer index after which to conclude the execution of the stage. It must be a positive number as well. This property is also optional and defaults to the stop value provided for the whole plug-in as it is executed, multiplied by the **step** value. Repetition is a negotiation between a plug-in author and its operator, where the former specifies its upper bound of repetitions, and the latter theirs, which should default to no repetition at all. If the former is greater than the latter, the execution of the plug-in is aborted.
- Its **variable** string value is the subscript that is used on the stage to retrieve the iterator, the latter being an object with the following properties:
 - a **chunk** property equivalent to the absolute value of the **step**
 - an **offset** property obtained by summing all the previous chunks with the **start** value
 - a **total** property returning the sum of the **chunk** and the **offset**

This value is also optional and defaults to the character "i".

After each iteration, by default, a provisional output of the stage is constructed containing an array of the outputs produced by each iteration. The provisional

output is augmented with an **Iterator** property retrieved through the string subscript indicated by the **variable** property of the **repeat** directive and appended to the output pipeline for successive iterations of the stage to access. It is possible to escape iteration before reaching the **stop** value by combining the **repeat** directive with the **assert** directive: in the context of repetitions, assertion failures will only conclude the loop instead of aborting the whole plug-in.

Merger

Plug-in authors can explicitly merge a stage's output with previous ones, similar to how repetition merges the various iterations. It is also possible to customize how the merging works in both occurrences, beyond the simple appending operation into an array. Those preferences are expressed via a **merge** property in a stage, accepting the following object as its value:

- Its **stage** property accepts an array of numbers, representing the indices of the previous stages to merge with the current one. If it contains a greater index than the current one, the runtime must abort the execution of the plug-in. This property is optional if the only source of merging is a **repeat** directive.
- Its **default** and **override** properties accept, respectively, a single value and a list of key-value pairs of **Policy** strings. The key in the pair is the string subscript that the policy pertains to. When accessing a string subscript,

implementations must first check for the iterator variable if present to return that instead, then for the key in the `override` list, and finally use the `default` policy. Single value and integer subscripts always refer to the `default` policy. A `Policy` string can assume one of the following values:

- `"none"`, which is the default, and means that the stages and iterations are not merged at all and are accessed as a vector containing each output in chronological order of appending.
- `"first"` and `"last"` redirect the subscript to the first and the last output that is being merged, respectively.
- `"keep"` redirects the subscript to each page, beginning with the first one, until it resolves any value and it is returned, and if no value is returned, then it behaves like any other non-resolving subscript.
- `"replace"` does the same as `"keep"` but beginning with the last page and going backwards.
- `"append"` which does not resolve any value for single-value subscripts, but in case of integer subscripts, concatenates each merged output that represents an array, and then applies the subscript to the combined array, while in the case of string subscripts, does that for each array that is children of the referenced string subscript in every merged output.

4.1.7 Handling groups of plug-ins together

The JOHN Language does not currently provide constructs allowing for recursion or automatic authentication management. Advanced logic like this, not expressible through the JOHN Language, requires multiple plug-ins executed more than once to accomplish the desired task. JOHN Language implementations must allow plug-ins to share an “execution group” that maintains the same context across them, mainly the cookies. Once in a group, it is possible to mark individual stages with an `id` property, expected to be unique in relation to the group, signaling the runtime to execute the request only once in the group and reuse its response for successive invocations without sending any HTTP request. This property, if present, accepts a string. It is also possible to mark the stage with a `defer` property, accepting an optional boolean indicating whether to skip its execution until all the requests in the group have completed theirs. The affordance to use to provide plug-ins to an execution group and to run the deferred stages is not specified as it is an implementation detail. Combining the `defer` and the `id` properties allows the definition of clean-up requests to run only once, or non-standard authentication procedures that rely on cookies and require explicit login and logout procedures: those would be wasteful or suspicious to repeat for each plug-in execution.

4.1.8 Yielding a normalized result

Although services that accomplish the same tasks will gravitate toward similar interfaces, they may feature inconsistencies requiring their conversion into a common representation.

Result

The `result` property of the root JOHN object allows the plug-in author to describe the structure of the plug-in's output. The value accepted by the property is a recursive key-value array of subscripts on the left-end side and their result on the right-end side, the latter expecting either a `Result` type or another key-value list that implementations must flatten out by concatenating subscripts together; the root `result` property is equivalent to an empty "" string subscript; it is also possible to chain together multiple subscripts in a single left-end key. The first subscript in a key, whether the only subscript or not, can have its brackets omitted.

The result subscript syntax

The result subscript syntax is a superset of the one defined for variables: integers or strings between square brackets [] indicating array and hierarchy subscripts. To support the output manipulation capabilities, a handful of additions are present:

- The jolly subscript, indicated by the * asterisk symbol, resolves any integer and any string provided at its position of the subscript.

-
- The range subscript `<*<` resolves the range of indices indicated at its opposite ends. It is also possible to omit one of them (including the less than sign `<`), and in that case, they will, respectively, either be replaced with the index 0 or considered as an indication of a minimum floor for the array subscript, accepting infinite values forward. It is also possible to define a bound with the variable subscript syntax, that will resolve either the single-value if it is convertible to a number, or the index of the vector if the latter is present. The range always includes its defining bounds.
 - The pattern subscript supports inserting asterisks in a string subscript, each resolving any number of characters inside them. Testing for the pattern starts at the beginning of the string and proceeds rightward. The testing of string literal components occurs lazily unless they are the last component of the testing pattern, in which case they are tested eagerly from the end. Multiple contiguous asterisks are equivalent to a single one. Implementations must perform the testing in a case-sensitive way.

Catch-all style subscripts, meaning the jolly, pattern, and range subscripts with a non-defined upper bound, cannot be resolved statically as they represent infinite values. In cases where a plug-in's result is transferred to a finite static product, implementations must discard those subscripts. In dynamic contexts, instead, they must provide a mechanism to query the plug-in output of arbitrary subscripts using

the JOHN result subscript syntax. Since multiple subscripts might resolve a subscript, implementations must keep track of the specificity of a subscript according to the following rules:

- Exact matches always have the highest value.
- The higher the number of catch-all symbols contained in the pattern subscript, the lower the specificity is. When they are the same, their position in the pattern is considered, with the pattern string with the earliest occurring asterisk `*` having the lower specificity.
- Ranges with fully defined bounds have equal specificity.

When implementations have to choose between more than one matching result subscript with maximum specificity, the one defined earliest in the **result** directive is the chosen one. Plug-in authors should not rely on the specificity system provided here for completeness and instead avoid overlap between subscripts when possible.

The result value mapping

The **Result** type represents what a result subscript key resolves to. It accepts either of the following values:

- A string literal to resolve the subscript with a value unconditionally. Upon access, the following steps must occur:

-
- for each catch-all subscript in the left-end side, the JOHN implementation replaces each catch-all symbol on the string with the actual value, starting from the first and going rightward.
 - then the entire string literal is tested as a single variable subscript, which if resolved will incorporate the full output tree (with the single-value, vector, and hierarchy subscripts all reachable) at the corresponding result subscript
 - finally, if the previous step failed to resolve a single variable, the string literal is scanned for variables to interpolate, as defined previously in this document, and their resolved values substituted inside it.

If the literal contains more catch-all symbols that the left-end side subscript can fill, then no value is returned for the subscript.

- A **Condition** object resolves the subscript only if the specified assertion, as defined previously for stages, is verified. It contains three properties:
 - **assert**, which accepts an **Assertion** object.
 - **result**, which is the **Result** value to choose if the assertion succeeds and it is mandatory for valid **Condition** objects.
 - **catch**, which is the **Result** value picked if the assertion test fails instead, and it is optional.

Since, from a JSON perspective, a key-value list of **Results** and a **Condition** object are syntactically the same, implementations must first check for the reserved keyword **"result"** to be present inside it: result subscript literals containing only the **"result"** keyword are not allowed, as they will map to **Condition** objects and their other properties discarded.

4.1.9 Example usage of advanced JOHN features

The Dropbox indexing plug-in of the University Cloud Suite showcases assertions, merging of stages with and without repetition, and result normalization.

Listing 4.3: Indexing a Dropbox drive for the University Cloud Suite

```
1 {
2   "about": { /* Omitted for brevity */ },
3   "pipeline": [
4     {
5       "url": "https://api.dropboxapi.com/2/files/list_folder",
6       "method": "POST",
7       "authorization": "bearer",
8       "body": { /* See Dropbox API */ },
9       "status": [ 200 ]
10    },
11    {
12      "assert": {
13        "$2[has_more]": { "contained": ["true"] }
14      },
15      "repeat": { "stop": 150 },
16      "merge": {
17        "stage": [1],
18        "default": "append",
19        "override": {
20          "has_more": "last",
21          "cursor": "last"
22        }
23      },
24    }
25  ]
26 }
```

```

24         "url": "https://api.dropboxapi.com/2/files/list_folder/
           ↪ continue",
25         "method": "POST",
26         "authorization": "bearer",
27         "body": {
28             "cursor": "$2[cursor]"
29         },
30         "status": [ 200 ]
31     }
32 ],
33 "result": {
34     "cursor": "$2[cursor]",
35     "items[0<*<$2[entries]]": {
36         "id": "$2[entries][*][id]",
37         "name": "$2[entries][*][name]",
38         "iso8601": "$2[entries][*][server_modified]",
39         "bytes": "$2[entries][*][size]",
40         "path": {
41             "full": "$2[entries][*][path_display]"
42         },
43         "kind": {
44             "assert": {
45                 "$2[entries][*][.tag]": { "contained": ["file"] }
46             },
47             "result": "file",
48             "catch": {
49                 "assert": {
50                     "$2[entries][*][.tag]": { "contained": ["folder"] }
51                 },
52                 "result": "folder",
53                 "catch": "unknown"
54             }
55         }
56     }
57 }
58 }
```

4.1.10 Quick reference of the JOHN Language syntax

For the convenience of plug-in authors, a reference of all the possible JOHN commands is provided here.

Listing 4.4: The JOHN Language syntax

```

1 {
2   "about": {
3     "name": String,
4     "version": Int,
5     "protocol": String
6   },
7   "parameters": String | true | false | "username" | "password" |
      ↪ "token" | [String: ...],
8   "pipeline": [
9     {
10      "id": String,
11      "defer": Bool,
12      "assert": [String: {
13        "exists": Bool,
14        "contained": [String]
15      }]
16      "repeat": {
17        "start": Int
18        "step": Int
19        "stop": Int
20        "variable": String
21      }
22      "merge": {
23        "stage": [Int],
24        "default": "first" | "keep" | "append" | "replace" |
          ↪ "last" | "none",
25        "override": [String: "first" | "keep" | ...]
26      }
27      "authorization": "basic" | "bearer" | "none",
28      "redirects": Int,
29      "url": String,
30      "method": String,
31      "status": [Int],

```

```

32         "headers": [String: String],
33         "query": [String: String],
34         "cookies": true | false | [String: true | false | String
    ↪ ],
35         "body": String | [String: String],
36         "yield": "headers" | "body" | "cookies",
37         "encode": "json" | "form",
38         "decode": "auto" | "raw" | "json" | "form" | "xml" | "xml
    ↪ -json" | "soap" | "soap-json" | "html" | "base64"
39     }
40 ],
41 "result": String | {
42     "assert": [String: {
43         "exists": Bool,
44         "contained": [String]
45     }],
46     "result": ...,
47     "catch": ...
48 } | [String: ...]
49 }

```

4.2 The JOHN Runtime for Swift Server-side Development

The JOHN Runtime is a library module for the Swift programming language that implements the JOHN Language in a dynamic runtime environment. A key goal is for interactions with JOHN plug-ins to behave according to the language conventions. To that extent, the peculiar features of the Swift language featured in Chapter 3, such as the Codable protocol, introspection, property wrappers, and protocol-oriented programming, are used pervasively in the library. The Async HTTP Client, maintained by the Swift Server Workgroup, is the backbone of the

library, providing the high-level interface used to send HTTP requests. On the flip side, since it depends on the Swift-NIO framework (which requires a POSIX-compliant operating system to operate), it is the reason why the JOHN Runtime does not currently compile on Windows systems. The JOHN Runtime is available to Swift projects targeting Apple and Linux platforms across Intel and Arm architectures, with Linux on Intel systems being the one used by most server-side deployments.

4.2.1 Implementing the language's data structures

To represent internally all the data structures that are accessible through the variable subscript syntax as described in the specification, the JOHN Runtime makes use of a generic `IOProtocol` type, whose name reflects its double nature as both input of a pipeline stage and its output.

Listing 4.5: The `IOProtocol` definition

```
1 protocol IOProtocol {  
2     var wrappedValue: Any { get }  
3     var text: String? { get }  
4     var number: Double? { get }  
5     var boolean: Bool? { get }  
6     var indices: Range<Int>? { get }  
7     subscript(_ index: Int) -> (any IOProtocol)? { get }  
8     var keys: [String] { get }  
9     subscript(_ key: String) -> (any IOProtocol)? { get }  
10 }
```

As visible from the protocol definition, the variable backing store allows conforming types to simultaneously provide single-value, vector, and key-value col-

lections on the same node. In addition to the language specification, the JOHN Runtime provides access to the single value wrapped in a node via either a string, boolean, or double-precision floating point number. The usage of the protocol occurs in three internal structures to implement essential behaviors required for plug-in execution:

- **IOPayload** allows the wrapping of Foundation's types (this is the Swift standard library) inside an **IOProtocol** interface. When using single-value accessors, they return the wrapped value as-is if its type is the same as the accessor (string, boolean, or double); otherwise, they attempt a conversion: most Foundation types are losslessly convertible to strings, so the string accessor is the primary way to access single values. Vector and dictionary accessors work similarly by type-casting the wrapped value to an array of type-erased values (the **Any** type) or a dictionary of String keys and type-erased values, respectively. Finally, it provides initializers for many base types, as well as for parsing JSON-formatted strings.
- **IOMarkup** allows the representation of XML and HTML document trees as **IOProtocol**-compliant objects. It works by wrapping three values: a string value for the tag name, a generic vector of **IOProtocol** elements for children, and a generic key-value list of **IOProtocol** attributes.
- **IOPagination** allows wrapping several generic **IOProtocol** objects with an

access policy and an iterator as defined in the JOHN Language specification.

Both three concrete types represent elements inside them as generic `IOProtocol` elements to enable the mixing of them together: the "xml-json" and the "soap-json" decoding types defined in the specification require the insertion of an `IOPayload` for the json inside an `IOMarkup` for the XML; in turn, it is possible to have pagination on the whole structure, which will merge the various `IOMarkup` objects via an `IOPagination`. The generic nature also grants the flexibility to extend the supported data formats in the JOHN Language without requiring changes to the variable subscript implementation.

Listing 4.6: The `Variable` struct, represented as a protocol for brevity

```

1 enum Path { case attribute(String), child(Int) }
2 protocol Variable {
3     var stage: Int { get }
4     var path: [Path] { get }
5     init(string: String) throws
6     static func findVariableRanges(_ string: String) -> [Range<
7         ↪ String.Index>]
8     func resolve(with outputs: [(any IOProtocol)?]) throws -> any
9         ↪ IOProtocol
10    func canBeResolved(with outputs: [(any IOProtocol)?]) -> Bool
11    func substitute(outputs: [(any IOProtocol)?], in string: String,
12        ↪ urlEncoded: Bool = false) throws -> String
13 }
```

The `Variable` struct implements the parsing and substitution logic related to the variable subscript syntax. As will be discussed later on, despite the overlap across the variable and the result subscript syntax from the perspective of plug-in authors, their implementations do not bear any relationship.

4.2.2 Automatic parsing of plug-ins

As stated in Chapter 3, the rationale for choosing JSON as the foundation of the JOHN Language syntax is to dramatically reduce the work required for parsing plug-ins before executing them. The JOHN Runtime implements all the JOHN constructs defined in the specification as structs conforming to the `Codable` protocol and only goes the extra mile in overriding the default provided parsing logic for directives like `body` or `result` that can accept multiple types and recursion. The primary source of complexity in the parsing plug-in is undoubtedly the variable and result subscript syntax. In both cases, the library implements a manual character-by-character parser and substitution engine: since the library evolved side-by-side with the language, and the result subscript is a late addition in its development, the two subscripts, while appearing the same to the plug-in author, are entirely different code bases in the runtime. Parsing of the former only occurs when attempting to resolve, substitute, or test for a variable. Parsing of the latter is done simultaneously with the rest of the plug-in through custom `Codable` initializers that first decode a dictionary with string keys and then map those keys to Subscript types; how the library handles subscripts containing dependencies on stage outputs through variables to resolve is a different topic unrelated to source code parsing.

Listing 4.7: Retrieving a JOHN plug-in from a database via the Fluent ORM

```

1 import Foundation
2 import Vapor
3 import Fluent
4 import JOHN
5
6 public final class MyModel: Model {
7     public static let schema: String = "my_model"
8     @ID(key: .id) public var id: UUID?
9     @Field(key: "plugin_key") public var plugin: JOHN?
10 }
11
12 public struct MyModelMigration: AsyncMigration {
13     public var name: String { "CreateMyModel" }
14     public init() {}
15     public func prepare(on database: Database) async throws {
16         try await database.schema(Provider.schema)
17             .id()
18             .field("plugin_key", .json)
19             .create()
20     }
21 }
22
23 // Fetching the plug-in
24 let plugin: JOHN = try await MyModel.find(model_id, on: request.db)
    ↪ .plugin

```

Using Swift in server-side development use cases highlights the benefits of the Codable from a plug-in operator standpoint. The most popular web framework for Swift, the Vapor project, using the Codable protocol in its Fluent ORM, allows for the automatic storage and retrieval of any Codable-conforming struct or class during CRUD operations in relational databases. Once a JOHN plug-in is stored inside the database, from the point of view of the plug-in operator, retrieving it, ready for execution, is a matter of a single line of source code.

4.2.3 Using JOHN plug-ins in Swift

A key design goal for the JOHN Runtime is to provide a set of interfaces to interact with JOHN plug-ins that would feel right alongside the classes and methods surfaced by the Swift standard library and the other frameworks experienced by macOS, iOS, and server-side Swift developers. This consideration goes beyond the mere parsing of the plug-in’s source code: it involves both its execution (which embraces modern Swift asynchronicity), the reading of its result (which is automatic and strongly typed), and the eventual handling of errors (through raised exceptions).

Asynchronous execution

To work around the inherently blocking operations (such as the network communications handled by JOHN plug-ins), Swift traditionally relied on a pattern of completion handlers passed as function parameters. Version 5.5 of the language introduced a better way to express completion handlers linearly with the flow of execution with the **await** keyword to improve the legibility of Swift asynchronous programs, avoiding multiple nested anonymous functions (known in Swift as closures). The JOHN Runtime exposes only asynchronous interfaces to its users, taking full advantage of the syntactic benefits of the language’s **async/await** capability.

Listing 4.8: An excerpt of the JOHN struct

```
1 public extension JOHN {  
2     public func execute(on group: JOHN.Group? = nil, with  
        ↪ options: JOHN.Options = .none) async throws ->  
        ↪ AnyResult { /* ... */ }  
3 }
```

Invoking the asynchronous `execute` method on the plug-in itself and passing optionally the execution group to run the plug-in into, and the options and parameters to use while doing so, is the primary way to execute a plug-in. It is a primary, but not the exclusive way, because as it is custom for Swift idiomatic code, there is extensive use of default parameter values, overloading, and alternative ways to express the same intention, but in ways that may make more sense depending on the context.

Options and Execution Groups

Listing 4.9: Initialization of `JOHN.Options`

```
1 public enum Provider {  
2     case createNew, sharedEventLoop(any EventLoopGroup)  
3 }  
4 public enum Repetitions {  
5     case allowed(cap: Int)  
6 }  
7 public enum Input {  
8     case text(String), json(Data), dictionary([String: Any]),  
        ↪ array([Any]), webForm(String)  
9     internal var ioPayload: IOPayload? { get }  
10 }  
11 public enum Credentials {  
12     case basic(username: String, password: String), bearer(token  
        ↪ : String)  
13 }  
14
```

```
15 public extension JOHN.Options {
16     public init(provider: Provider = .createNew, repetitions:
        ↪ Repetitions? = nil, input: Input? = nil, credentials:
        ↪ Credentials? = nil, debug: (any DebugDelegate)? = nil)
        ↪ { /* ... */ }
17 }
```

As described in the JOHN Language specification, the JOHN Runtime provides a way to input parameters and authentication credentials and to define the allowed iterations in the `repeat` directive via a `JOHN.Options` data structure. As an implementation detail required by the internal usage of the Async HTTP Client library to power the HTTP engine, an option is present regarding the `EventLoop` provider it uses (refer to Chapter 3 for the Swift Non-Blocking IO architecture). Finally, there is a way to assign a delegate to inspect the functioning of the runtime, to be analyzed later in the chapter.

Listing 4.10: Initialization of `JOHN.Group`

```
1 public enum Share: CaseIterable {
2     case cookies, requests
3 }
4 public extension JOHN.Group {
5     init(defaults: JOHN.Options = .none, sharing: Set<Share> = .
        ↪ init(Share.allCases)) throws { /* ... */ }
6 }
```

As described in the language specification, execution groups in JOHN provide a shared cookie store and unique and/or deferred execution of requests. The JOHN Runtime allows plug-in operators to specify whether they wish to opt out of one or both behaviors when initializing a `JOHN.Group` object. As an additional

convenience, it is possible to provide a shared set of `JOHN.Options` used for all plug-ins in the group.

Listing 4.11: Convenience initializers for `JOHN.Options` and `JOHN.Group`

```

1 public extension Options {
2     public static func repetitionsAllowed(_ cap: Int, on
        ↪ provider: Provider = .createNew) -> Self { /* ... */ }
3     public static func input(_ input: Input, on provider:
        ↪ Provider = .createNew) -> Self { /* ... */ }
4     public static func credentials(_ credentials: Credentials,
        ↪ on provider: Provider = .createNew) -> Self { /* ...
        ↪ */ }
5     public static func debug(_ debug: (any DebugDelegate)?, on
        ↪ provider: Provider = .createNew) -> Self { /* ... */ }
6     public static func provider(_ provider: Provider) -> Self {
        ↪ /* ... */ }
7 }
8
9 public extension Group {
10     static func defaults(_ defaults: JOHN.Options) throws ->
        ↪ Self { /* ... */ }
11     static func sharing(_ sharing: Share...) throws -> Self { /*
        ↪ ... */ }
12 }

```

In both `JOHN.Options` and `JOHN.Group` there are additional static methods to initialize only the desired preference and rely on the default value for the others. They rely on the type system of the Swift language that allows the omission of the base type when it is inferable in other ways: using just `.input()` or `.sharing()` in the function parameter reduces the verbosity of the code significantly.

Listing 4.12: Closure-based `JOHN.Group` initialization

```

1 public extension JOHN {
2     static func withExecutionGroup(defaults: JOHN.Options = .none,
        ↪ sharing: Set<Group.Share> = .init(Group.Share.allCases),
        ↪ body: ((Group) async throws -> ())) async throws

```

```

3 }
4 public extension JOHN.Group {
5     func execute(_ plugin: JOHN, with options: JOHN.Options = .none)
6         ↪ async throws -> AnyResult
7 }

```

Similarly, it is possible to use execution groups by creating them separately or using a convenience method on the root JOHN struct.

Listing 4.13: Excerpt from the AnyResult definition

```

1 public struct AnyResult: IOProtocol, Sequence {
2     @Source var sources: [(any IOProtocol)?]
3     let path: Path
4     let wrappedPayload: (any IOProtocol)?
5
6     typealias Path = (map: [Subscript: Result], inherited:
7         ↪ Subscript)
8     //
9     @propertyWrapper
10    class Source {
11        internal let outputs: [(any IOProtocol)?]
12        public var wrappedValue: [(any IOProtocol)?] { outputs }
13        public var projectedValue: Source { return self }
14        init(wrappedValue: [(any IOProtocol)?])
15        public subscript(variable: String) -> (any IOProtocol)?
16    }
17    //
18    internal init(sources: Source, path: Path, wrappedPayload: (any
19        ↪ IOProtocol)?)
20    internal init(sliceOf result: AnyResult, at pathComponent:
21        ↪ Subscript.Element?) throws
22    internal init(sliceOf result: AnyResult, at path: Subscript)
23        ↪ throws
24    internal init(rootOf outputs: [(any IOProtocol)?], via map:
25        ↪ Result)
26    internal init(rootOf sources: Source, via map: Result)
27    //
28    public subscript(path: Subscript.Element...) -> Self?
29    public subscript(path: Subscript) -> Self?
30 }

```

Plug-in execution returns a value of type `AnyResult` that conforms to the `IOProtocol` and exposes its otherwise private methods to the library’s users. This value represents a view of the output of the stages, meaning it only contains a subscript path and a reference to the output itself. The output is wrapped inside a `Source` class (enabling reference behavior for memory efficiency) alongside the result subscript defined by the plug-in author. Each `AnyResult` value, in addition to the accessors mandated by the `IOProtocol`, also provides subscripts for the result subscript syntax defined in the JOHN Language specification. Because the `Subscript.Element` type adopts the `RawRepresentable`, `ExpressibleByIntegerLiteral`, and `ExpressibleByStringLiteral` protocols, and because the `AnyResult` object provides overloads for the subscript method with support for variadic arguments, plug-in operators can use the familiar array subscript with square brackets `[]` to slice an `AnyResult` type using primitive types like strings and integers, converted into JOHN result subscript types automatically.

Type-safe result interface

Using subscripts directly on `AnyResult`, while powerful and flexible, has three downsides:

- The entire plug-in output persists in memory because each slice of `AnyResult` keeps a reference to the `Source` class. Therefore, it is essential to get the required data from it as soon as the plug-in concludes its execution and to immediately allow the Automated Reference Counting system (see Chapter

3) to deallocate all `AnyResult` slices.

- Extracting the desired information from the result is a process that is tedious and wordy, especially if repeated for each plug-in execution.
- Swift is a typed programming language, and handling type-erased types like `AnyResult` is cumbersome as it requires repetitive type-casting operations.

The JOHN Runtime leans on the strengths of the Swift programming language, harnessing its advanced capabilities such as dynamic introspection, protocol-oriented programming, and property wrappers to enable the automatic translation of the `AnyResult` value returned by plug-ins and the type-safe structs and objects declared by the plug-in operator.

Listing 4.14: The `ExpectedResult` protocol

```
1 public protocol ExpectedResult {  
2     /// An initializer with no parameters is necessary to provide  
3     ↪ default values for all non-JOHN parsable properties in  
4     ↪ the synthesized initializers  
5     init()  
6     public init?(executing plugin: JOHN, on group: JOHN.Group? =  
7     ↪ nil, with options: JOHN.Options = .none, at path:  
8     ↪ Subscript = .init()) async  
9 }
```

Conforming a struct (or object) to the `ExpectedResult` protocol automatically creates a new initializer that receives a JOHN plug-in as input, executes it, and automatically assigns the values from its `AnyResult` value to the struct's properties.

Listing 4.15: An excerpt from the ExpectedValue property wrapper definition

```

1 @propertyWrapper
2 public class ExpectedValue<Value>: SettableValue {
3     internal var _sourcePath: Subscript
4     internal var _wrappedValue: Value
5
6     public var wrappedValue: Value { _wrappedValue }
7
8     private init(_wrappedValue: Value, _sourcePath: Subscript) {
9         self._wrappedValue = _wrappedValue
10        self._sourcePath = _sourcePath
11    }
12 }
13
14 // MARK: @ExpectedValue("key") var value: Int = 0
15 public extension ExpectedValue where Value: SettableWithRawValue {
16     ↪ /* ... */ }
17
18 // MARK: @ExpectedValue("key") var value: Int?
19 public extension ExpectedValue where Value: SettableWithRawValue,
20     ↪ Value: ExpressibleByNilLiteral { /* ... */ }
21
22 // MARK: @ExpectedValue("key") var value: [Int]
23 // MARK: @ExpectedValue("key") var value: Set<Int>
24 public extension ExpectedValue where Value:
25     ↪ SettableWithCollectionOfRawValues { /* ... */ }
26
27 // MARK: @ExpectedValue("key") var value: CustomType where
28     ↪ CustomType: ExpectedResult
29 public extension ExpectedValue where Value: ExpectedResult { /* ...
30     ↪ */ }
31
32 // MARK: @ExpectedValue("key") var value: [CustomType] where
33     ↪ CustomType: ExpectedResult
34 // MARK: @ExpectedValue("key") var value: Set<CustomType>
35 public extension ExpectedValue where Value:
36     ↪ SettableWithCollectionOfExpectedResults { /* ... */ }

```

To tell the ExpectedResult initializer what properties to handle and what re-

sult subscript to use to fetch the value, each one is marked with the `@ExpectedValue` property wrapper, declaring its type in the property itself and providing its subscript as the only argument to the wrapper, inside parentheses. When a property lacks a default value provided using the `=` assignment operator, its type must be `Optional`, for the runtime needs to have a value to assign it for the initializer to function correctly, as Swift requires the initialization of all non-optional stored properties.

Listing 4.16: The `SettableValue` protocol

```
1 public protocol SettableWithRawValue {
2     init?(_: String)
3 }
4 public protocol SettableWithCollectionOfRawValues {
5     init(_: [String])
6 }
7 public protocol SettableWithCollectionOfExpectedResults {
8     init(_: [any ExpectedResult])
9     static func children(of: AnyResult) -> [any ExpectedResult]
10 }
11 extension Bool: SettableWithRawValue {}
12 extension Character: SettableWithRawValue {}
13 extension Double: SettableWithRawValue {}
14 extension Float: SettableWithRawValue {}
15 // ...
```

The way the type-casting works is by requiring all single-value, vector, and dictionary types to conform to the `SettableWithRawValue`, `SettableWithCollectionOfRawValues`, and `SettableWithCollectionOfExpectedResults` protocols, respectively. These protocols, inspired by the `LosslessStringConvertible` defined in the Swift standard library and already implemented in most primitive types of the

language, guarantee the possibility of initializing the value just using a string. The way the assignment works is for the `ExpectedResult` initializer to use the `Mirror` introspection on itself to enumerate all of its properties (as discussed in Chapter 3), conditionally casting them to the `ExpectedValue` type to assign them through the `AnyResult` subscript and the `SettableWithRawValue` initializer; if the casting fails it also attempts to cast the value to `ExpectedResult` itself, allowing for the process to occur recursively.

Listing 4.17: Direct initialization of `SettableValue`-conforming types

```
1 public extension SettableWithRawValue {
2     init?(executing plugin: JOHN, on group: JOHN.Group? = nil, with
        ↪ options: JOHN.Options = .none, at subscript: Subscript =
        ↪ .init()) async { /* ... */ }
3 }
4 public extension SettableWithCollectionOfRawValues {
5     init(executing plugin: JOHN, on group: JOHN.Group? = nil, with
        ↪ options: JOHN.Options = .none, at subscript: Subscript =
        ↪ .init()) async { /* ... */ }
6 }
7 public extension SettableWithCollectionOfExpectedResults {
8     init(executing plugin: JOHN, on group: JOHN.Group? = nil, with
        ↪ options: JOHN.Options = .none, at subscript: Subscript =
        ↪ .init()) async { /* ... */ }
9 }
```

The three protocols also allow the initialization of their conforming types directly through the execution of a plug-in without needing to declare structured types.

Error handling

The JOHN Language specification defines failure scenarios where the implementation must abort execution. The JOHN Runtime models them through Swift's exception-handling mechanism. Plug-in operators may wish to handle:

- `Execution.Error.statusNotValid` when the HTTP response of a stage has a status code not contained in the specified list of valid status codes.
- `Execution.Error.unableToEncodeObjectAsForm` when the plug-in author incorrectly provided a hierarchical object tree and then specified its encoding type as a web form (which only encodes a flat list of key-value pairs).
- `Assertion.Error.assertionFailed` when at least one of the assertions defined for a stage failed to test successfully, and it did not do so as part of a repetition (where the language specification allows for assertion failures as loop-breaking mechanisms).
- `Authorization.Error.missing` when a parameter or a stage specifies the requirement for either a username-password combo or a token and the plug-in operator has not provided one.
- `Authorization.Error.base64EncodingFailure` when a fatal error occurred in encoding the "`$username:$password`" string in base64 for the HTTP Basic Authorization.

-
- `Merger.Error.outOfBounds` when a stage requests its merger with a stage whose index is greater than or equal to the current one.
 - `Parameters.Error.missingRequiredParameter` when the plug-in operator did not provide one of the parameters marked as required (a boolean value of true).
 - `Repetition.Error.notAllowed` when the plug-in operator did not allow enough iteration counts (or at all) upon executing the plug-in or initializing its execution group for the iterations attempted by a stage to occur.
 - `Variable.ParsingError.underminatedSubscript` and `Variable.SplittingError.underminatedSubscript` if upon substituting the variable values inside a string, the variable parser encounters an opening bracket [without a matching closing bracket].
 - `Variable.ParsingError.NaN` if the first digit of a variable's stage index (the number immediately after the dollar sign "\$") is not a number, meaning that the variable is malformed.
 - `Variable.SubstitutionError.overflow` if the variable has a stage number higher than the total number of executed stages at that time.
 - `Variable.SubstitutionError.notFound` if the variable subscript does not resolve to any value.

The JOHN Runtime throws additional errors, used internally to signal status, but those do not bubble up to the plug-in operator. Plug-in operators may still encounter other exceptions raised by the Async HTTP Client concerning networking errors, for which they should consult the relevant documentation[11].

4.2.4 Inspecting the runtime

For most use cases, awaiting a plug-in as it executes and handling its thrown exceptions is sufficient. In other circumstances, the plug-in operator needs more insight into the progress of a plug-in execution and the data generated along the way, either for displaying status, logging purposes, or developing an integrated development environment.

DebugDelegate

It is possible to attach an object conforming to the `DebugDelegate` protocol to either a single plug-in or a whole execution group. Since the protocol requires conformance to `AnyObject`, only classes can adopt it. An important aspect to note is that, as typical in Cocoa APIs like the `XMLParser`, the delegate's reference kept by the JOHN Runtime is weak, meaning it does not count against the Automatic Reference Counting system. The plug-in operator must store the delegate elsewhere in a variable, or it risks being automatically deinitialized before the plug-in execution even begins.

As the runtime executes the plug-in, at appropriate moments of its execution

and if a delegate is present, it will call the delegate’s methods to notify it about the event that occurred, alongside relevant information.

Listing 4.18: DebugDelegate’s events emitted before they occur

```

1 public extension DebugDelegate {
2     func debug(willDeferStage stage: Int) {}
3     func debug(willExecuteDeferredRequest request: HTTPClientRequest,
4         ↪ index: Int) {}
5     func debug(willStartRepetitionAt start: Int, repeatingUntil stop
6         ↪ : Int, step: Int) {}
7     func debug(willStartCachedRepetitionWith id: String,
8         ↪ repeatingUntil count: Int) {}
9     func debug(willStartHTTPRequest request: HTTPClientRequest) {}
10    func debug(willStartHTTPRedirect request: HTTPClientRequest,
        ↪ count: Int) {}
11    func debug(willStartBufferingBody length: Int) {}
12    func debug(willProcessBodyAs type: String) {}
13 }

```

Following Cocoa’s naming conventions, those events emitted before they occur have their first argument with a leading “will” in their name. Because of their nature, those events do not carry the result of their operation.

Listing 4.19: DebugDelegate’s events emitted after they occur

```

1 public extension DebugDelegate {
2     func debug(didStartPlugin plugin: JOHN, withOptions options:
3         ↪ JOHN.Options) {}
4     func debug(didStartStage stage: Int) {}
5     func debug(didIncreaseCounter counter: Int, startingAt start:
6         ↪ Int, repeatingUntil stop: Int, step: Int) {}
7     func debug(didEndRepetitionAt counter: Int, startingAt start:
8         ↪ Int, repeatingUntil stop: Int, step: Int) {}
9     func debug(didEndCachedRepetitionWith id: String, repeatingUntil
10        ↪ count: Int) {}
11    func debug(didMergeStage stage: Int, withStages stages: [Int],
12        ↪ defaultPolicy: String, overriddenBy policies: [String:
13        ↪ String]) {}

```

```

8   func debug(didPassExistsAssertion value: String, assertion: Bool
    ↪ ) {}
9   func debug(didPassContainedAssertion value: String, assertion: [
    ↪ String]) {}
10  func debug(didFailExistsAssertion value: String, assertion: Bool
    ↪ ) {}
11  func debug(didFailContainedAssertion value: String, assertion: [
    ↪ String]) {}
12  func debug(didStoreCookie name: String, value: String, domain:
    ↪ String, path: String) {}
13  func debug(didDeleteCookie name: String, domain: String, path:
    ↪ String) {}
14  func debug(didSetCookieOnRequest request: HTTPClientRequest,
    ↪ name: String, value: String) {}
15  func debug(didAuthorizeRequest request: HTTPClientRequest,
    ↪ withUsername username: String, password: String) {}
16  func debug(didAuthorizeRequest request: HTTPClientRequest,
    ↪ withBearerToken token: String) {}
17  func debug(didEndHTTPRequest request: HTTPClientRequest,
    ↪ response: HTTPClientResponse) {}
18  func debug(didEndHTTPRedirect request: HTTPClientRequest, count:
    ↪ Int, response: HTTPClientResponse) {}
19  func debug(didStoreCachedHTTPResponse response:
    ↪ HTTPClientResponse, withId id: String) {}
20  func debug(didRetreiveCachedHTTPResponse response:
    ↪ HTTPClientResponse, withId id: String) {}
21  func debug(didEndBufferingBody length: Int, value: String) {}
22  func debug(didEndStage stage: Int, output: Any?) {}
23  func debug(didEndPlugin plugin: JOHN, result: AnyResult) {}
24 }

```

Most events, especially those relevant to plug-in development environments, are sent after their occurrence and carry the operation's results. Those events' first argument starts with "did".

LoggerDelegate

The most straightforward implementation of a `DebugDelegate` is a delegate that prints strings to standard output in response to each event. The JOHN Runtime already provides such delegate with the `LoggerDelegate` class.

Listing 4.20: The initializers of `LoggerDelegate`

```
1 import Foundation
2 import Logging
3
4 public class LoggerDelegate {
5     public init(standardOutputWithPrefix prefix: String? = nil,
6                 verbose: Bool = false,
7                 unsafe: Bool = false) { /* ... */ }
8
9     public init(logger: Logger,
10                level: Logger.Level? = nil,
11                verbose: Bool = false,
12                unsafe: Bool = false) { /* ... */ }
13 }
```

In addition to standard output, through the use of Apple’s open-source Swift-Log library, it is possible to designate a custom handler for the messages emitted by the delegate. This logging framework, widely used across Swift server applications, provides a standardized interface for third-party logging backends to receive log messages. The project’s official source code repository^[51] showcases many examples, like relays to cloud services and messaging apps, alongside more traditional logging solutions targeting the file system.

4.3 Developing JOHN Plug-Ins with the JOHN Editor

Because the JOHN Language syntax is valid JSON, there are already many editors and other tools suitable for producing JOHN plug-ins. Nevertheless, when the plug-in increases in complexity, having a dedicated tool capable of running and inspecting it as it grows is a meaningful assistance.

The JOHN Editor is a document-based SwiftUI application for Macintoshes running macOS 13 and later versions that provides a graphical user interface to the JOHN Runtime for Swift, inspired by the AppleScript Editor that ships with the operating system for writing scripts to automate the usage of the Mac.

4.3.1 Anatomy of the JOHN Editor

Being document-based, it launches straight into an untitled document featuring the skeleton of a valid JOHN plug-in. The user interface (Fig. 4.1) is straightforward:

- The primary view is an editable text view containing the JOHN source code of the plug-in in monospace font. Its color denotes the status of the code, whether it is the untouched placeholder code (light gray), a modified source code yet to be verified (black or white depending on the operating system's current color scheme), a valid JOHN plug-in (blue), or whether it failed to parse as it contains syntax errors (red).

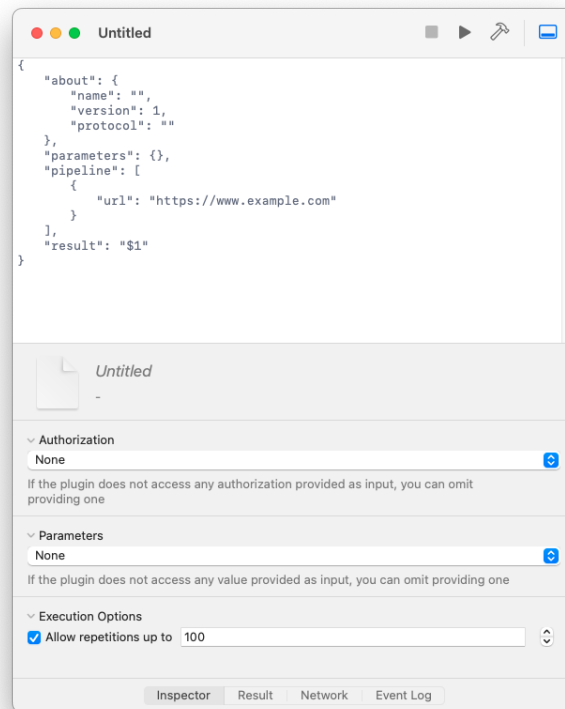


Figure 4.1: The JOHN Editor displaying an untitled document

- The accessory view is an optional inspector view that is resizable and provides access to the input and output of the plug-in's execution.
- The toolbar at the top of the window provides access to the document proxy and a series of buttons to stop the execution if ongoing, to start it if not, to build the plug-in without executing it, and to toggle the visibility of the accessory view.
- The status bar at the bottom of the window features a series of buttons to toggle the visibility of individual tabs in the accessory view (clicking the tab

related to the current view will hide the accessory view) and, on the left-end side, a status indicator for when the plug-in is ongoing execution, for when it succeeds, and for when it fails.

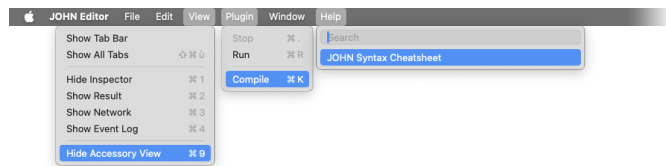


Figure 4.2: The menu bar commands of the JOHN Editor

As expected from a Mac program, all the actions of the toolbar and the status bar are also surfaced from the menu bar (Fig. 4.2) and provided with keyboard shortcuts: the combinations are the same as those used in the AppleScript Editor. From the Help menu, it is possible to open the same quick reference of the JOHN Language syntax seen earlier in this chapter.

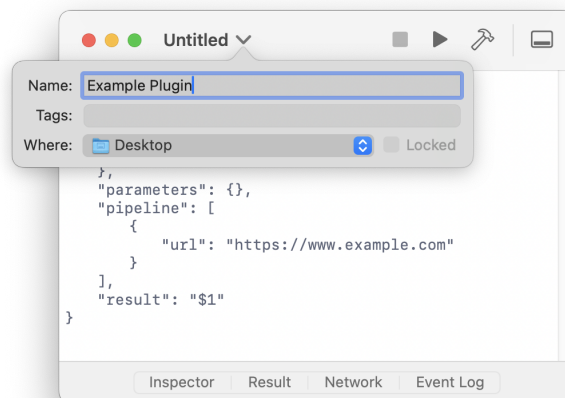
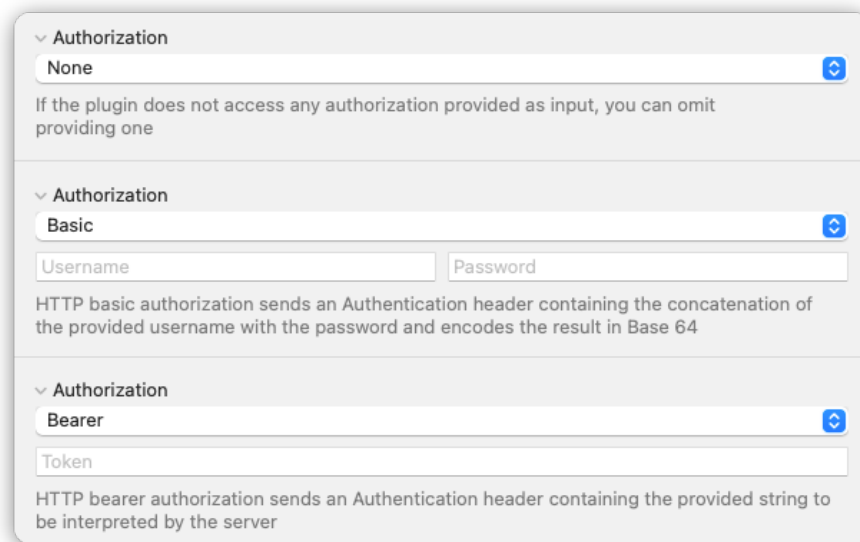


Figure 4.3: Clicking the document title allows to rename or move it

Clicking on the document title reveals an informational popover (Fig. 4.3), letting the user rename, move, and lock the file to prevent further changes.

4.3.2 Using the plug-in Inspector

The Inspector tab in the accessory view provides a quick view of the plug-in metadata and allows the user to provide the values to use in `JOHN.Options`.



The image shows a UI component titled "Authorization" with a dropdown menu. The dropdown is currently set to "None". Below the dropdown, there is a text label: "If the plugin does not access any authorization provided as input, you can omit providing one". The component is expanded to show three options: "None", "Basic", and "Bearer". Each option has a corresponding text input field. For "Basic", the input fields are labeled "Username" and "Password". For "Bearer", the input field is labeled "Token". Below each input field, there is a descriptive text: "HTTP basic authorization sends an Authentication header containing the concatenation of the provided username with the password and encodes the result in Base 64" for Basic, and "HTTP bearer authorization sends an Authentication header containing the provided string to be interpreted by the server" for Bearer.

Figure 4.4: Users can choose from three authorization options

The Authorization disclosure group (Fig. 4.4) allows to supply either none, Basic, or Bearer authentication credentials.

Parameters

None

If the plugin does not access any value provided as input, you can omit providing one

Parameters

Text

Type a string value in the field to be provided to the plugin at runtime

Parameters

JSON

Type the JSON source in this text field and it will be parsed and provided to the plugin at runtime

Parameters

Key-Value Dictionary

Key Value

Fill in the key-value pairs to be provided to the plugin at runtime

Figure 4.5: Users can input four different parameter types

The Parameters disclosure group (Fig. 4.5) allows to specify either none, JSON, textual, or key-value as the input data to the plug-in.

Execution Options

☒ Allow repetitions up to 100

Figure 4.6: To run plug-ins containing the **repeat** directive, repetitions must be first enabled

The Execution Options disclosure group (Fig. 4.6) provides as its only control a way to enable and set the limit of iterations to allow in the **repeat** directive.

4.3.3 Accessing debugging information

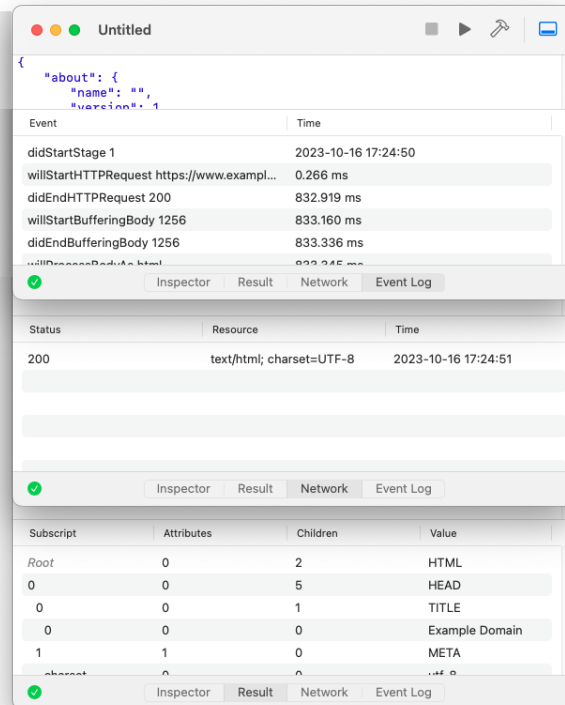


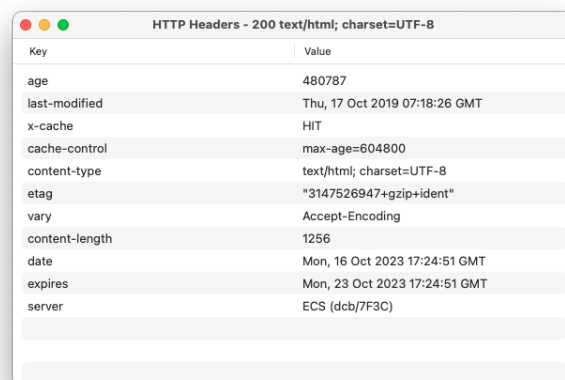
Figure 4.7: The JOHN Editor collects many kinds of debugging data

The remaining tabs of the accessory view (Fig. 4.7) provide debugging information relative to the plug-in's execution:

- The Result tab gives access to the plug-in's output as defined by its result subscript. As soon as the plug-in finishes running, the JOHN Editor interrogates the **keys** and **indices** properties on the **AnyResult** type returned by the JOHN Runtime and uses them to recursively traverse the entire tree structure of the output. For each node, those properties, alongside its string

value and a flattened value of its subscript, are inserted in the Result table view.

- The Network tab contains a list of all the HTTP responses collected during the execution of the various stages. Each row lists its status code, the returned **Content-Type**, and the timestamp at which the runtime received the response. It is possible (Fig. 4.8) to double-click on any entry to view a list of the returned response headers in a secondary window.
- The Event Log tab displays all the events emitted by the runtime's **Logger-Delegate** in non-verbose form. The first entry also shows the timestamp at which the plug-in started running, while all the others provide the relative duration of their operation.



The screenshot shows a window titled "HTTP Headers - 200 text/html; charset=UTF-8". It contains a table with two columns: "Key" and "Value". The table lists various HTTP response headers and their corresponding values.

Key	Value
age	480787
last-modified	Thu, 17 Oct 2019 07:18:26 GMT
x-cache	HIT
cache-control	max-age=604800
content-type	text/html; charset=UTF-8
etag	"3147526947+gzip+ident"
vary	Accept-Encoding
content-length	1256
date	Mon, 16 Oct 2023 17:24:51 GMT
expires	Mon, 23 Oct 2023 17:24:51 GMT
server	ECS (dcb/7F3C)

Figure 4.8: Double-clicking an HTTP response reveals its headers

Chapter 5

Case study: the University Cloud Suite

In the software industry, sometimes products emerge serendipitously from other endeavors. What is arguably the most notable desktop program created in the 2010s, the Slack groupware, was initially constructed as a tool for internal use in a gaming company, became their primary product, and ultimately resulted in their acquisition for \$27.7 billion by Salesforce [42]. The creation of the JSON HTTP Notation arose during the early development of the University Cloud Suite. This effort, when completed, will provide a family of web-based applications designed to enhance the personal productivity of individual college-enrolled students. The key design goal, and the one justifying the investment in JOHN, is digital sovereignty: for a student, the guaranteed access to all the information they have ever encountered or produced about their studies, while often looked past by, is of fundamental importance, one truly appreciated only after experiencing significant failures. It is an all-encompassing effort, from the storage mechanism backing the data pro-

duced by the suite and the structure of its proprietary file formats all the way to the ability to plug in existing services, either chosen by the student or imposed on them by their academic institution. Such a system guarantees that the user must at all times have full access to their information no matter what happens to the cloud solution. The only way to ensure this is by handling the storage through a service brought externally by the users themselves and that the file formats consist of directory structures of standard file types like JPEGs, MP3s, and TXT documents rather than binary formats or SQLite-based databases. Since the users' expectations regarding web applications are for their data to be available across every device, it is not enough for them to supply the local storage on their devices via modern file system APIs (introduced in the web standards in the 2010s) but also the cloud storage provider of their choice via a plug-in system powered by JOHN. This virtual file system provides the foundation for the desktop-class user experience of the University Cloud Suite: after signing in, users have access to a desktop and can browse files via a spatial file manager, and every program or document spawns its independent window, granting complete freedom in positioning them in the workspace. This chapter will provide an overview of the University Cloud Suite and the essential components that power its experience, beginning with the plug-in architecture and the JOHN-based virtual file system.

5.1 The JOHN-powered Virtual File System

The current implementation of the Virtual File System in the University Cloud Suite involves:

- JOHN plug-in protocols to implement CRUD operations with third-party cloud storage services.
- A handful of relational database tables to index and merge the contents of the external drives and keep track of their indexing status.
- REST endpoints for clients to operate on the drive or to mirror the index locally and maintain it in sync.
- A JavaScript `SharedWorker` and an `IndexedDB` table act as counterparts to the server on the client and enable virtually instantaneous read operations on the file system.

5.1.1 JOHN plug-in protocols

A set of fundamental operations is required to operate a remote file system:

- An indexing operation, traversing the whole structure to make the client aware of all the available folders and files. This operation is surfaced via the `AUX_StorageProvider_SetupDriveIndexing` plug-in protocol.

-
- A download operation to read the contents of files found during indexing. The `AUX_StorageProvider_GetFileDownloadURL` protocol provides this functionality.
 - An insert operation allows for creating folders and uploading files. This operation is implemented via the protocols `AUX_StorageProvider_CreateFolder` and `AUX_StorageProvider_GetFileUploadURL` respectively.
 - An edit operation to rename the path of a resource and, as a consequence, either rename the resource in place or move it to another sub-tree in the hierarchy. This is achieved via the `AUX_StorageProvider_MoveFileOrFolder` plug-in protocol.
 - A deletion operation to remove either files or folders at the specified path. This is available via `AUX_StorageProvider_RemoveFileOrFolder`.

In addition to essential protocols, the University Cloud Suite virtual file system also exposes additional interfaces to improve the user experience:

- To avoid having to index the whole structure any time the system fetches changes, an update protocol is exposed as `AUX_StorageProvider_UpdateDriveIndexing`, returning the same result as the usual indexing plug-in, but also requiring a piece of information as input, used to timestamp the previous update and only return changed items.

- To avoid having to re-upload files just to copy them to a new path, such an operation is surfaced as the `AUX_StorageProvider_CopyFileOrFolder` plug-in protocol.
- To prevent encountering errors due to storage space being full, the ability to read the available, used, and total space, in bytes, provided by the drive is surfaced as `AUX_StorageProvider_GetStorageQuota`.
- If supported by the provider, a way to fetch a thumbnail preview of the file usable as the icon for the resource is available via `AUX_StorageProvider_GetThumbnail`.

5.1.2 Database tables and REST endpoints

The University Cloud Suite is a web application written in Swift using the Vapor framework and Fluent ORM, employing a PostgreSQL database for its persistent storage. As discussed thoroughly in previous chapters, the combination of Swift-native frameworks taking full advantage of the `Codable` protocol, such as the Fluent ORM and the JOHN Runtime, allows decoding JOHN entities automatically upon fetching from the database. Consequently, storing plug-ins as fields in the database is convenient, preferably using the `JSONB` type supported by PostgreSQL, making decoding the dictionary data faster.

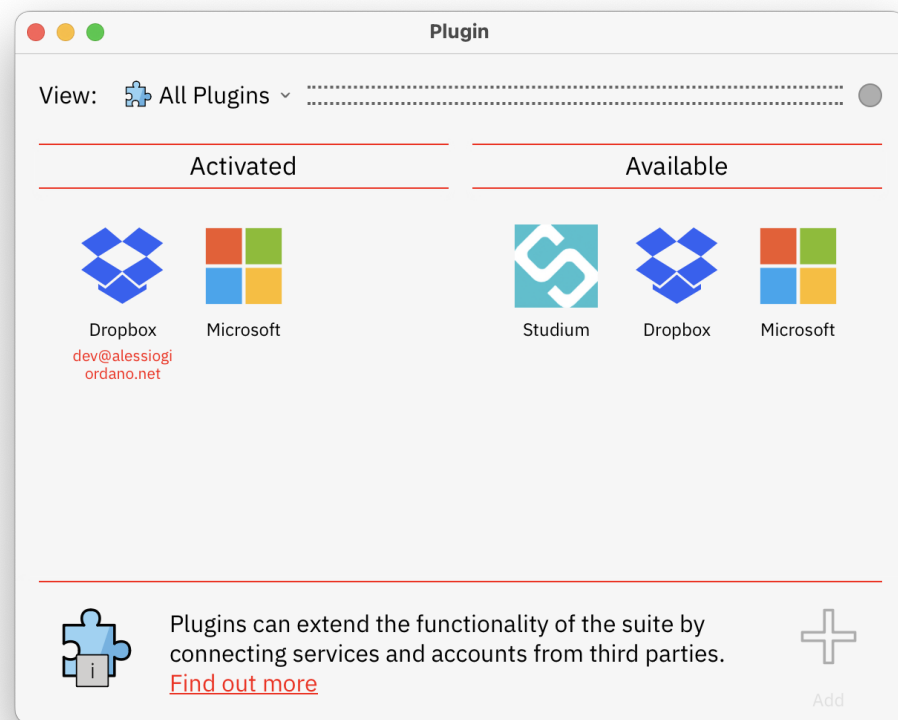


Figure 5.1: The Plug-in Chooser Preference Pane

Key to the vision behind the University Cloud Suite is allowing users to bring their services with them. From the standpoint of the user interface, a plug-in chooser (Fig. 5.1) provides a split view of the registered user accounts and available services for further extension of the product.

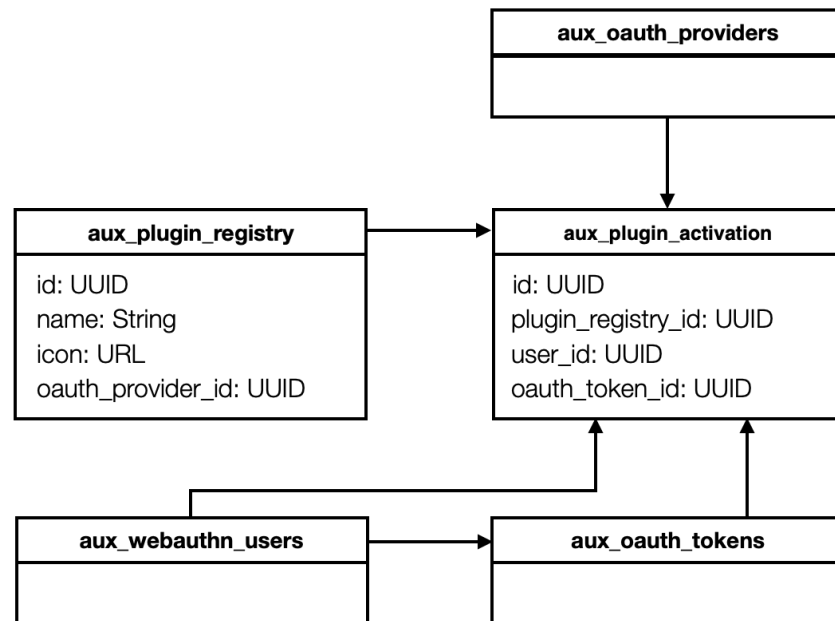


Figure 5.2: The Plug-in Database Architecture of the University Cloud Suite

To render this view, the database must contain a registry of plug-ins and OAuth-2 tokens for the plug-ins to be discovered, connected, and finally used by the various programs of the suite (Fig. 5.2). Users can filter plug-ins by type, presented as the program they extend: individual plug-ins get defined by the OAuth provider they connect to while serving multiple purposes.

The actual JOHN plug-ins, like the storage plug-in provider, are stored in a separate table, referencing the registry for discoverability and authentication (Fig. 5.3).

The virtual file system is made available to clients via three kinds of REST endpoints:

- The `/api/storage/drives` endpoint concerns actions affecting the storage service as a whole; this includes listing the connected cloud storage providers and their quota and managing the mirroring of the index on the client.
- The `/api/storage/$drive_id` exposes actions on individual drives, meaning mounting or ejecting them.
- The `/api/storage:$path` endpoint handles individual file system nodes directly through their path.

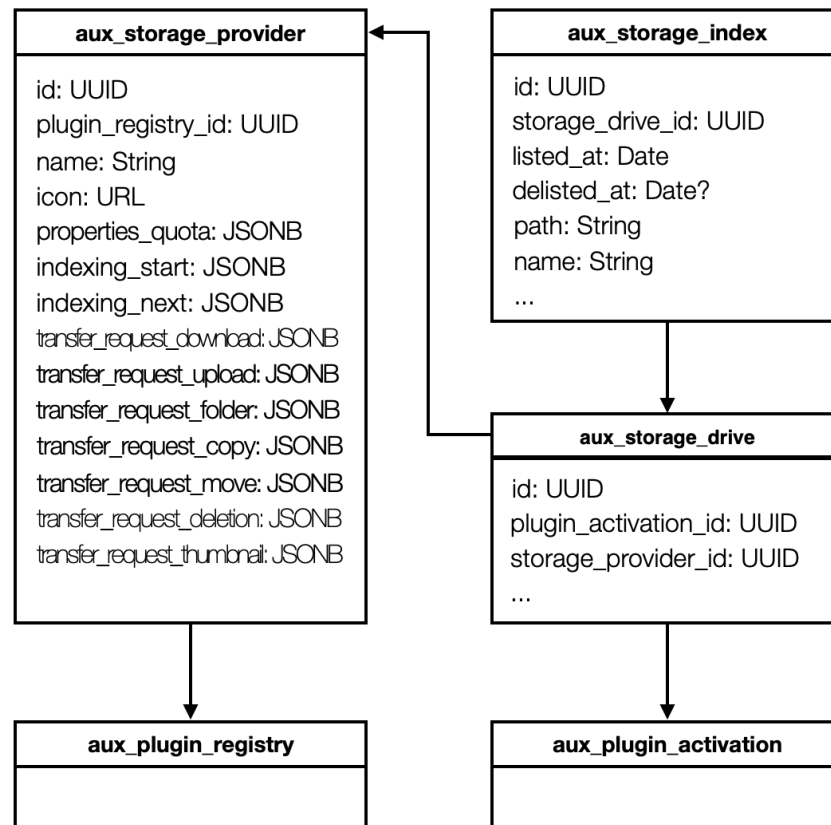


Figure 5.3: The Storage Database Architecture of the University Cloud Suite

The latter set of endpoints merits a discussion of its own.

Listing 5.1: The Path Query Endpoint

```
1 POST /api/storage:${path}
2 {
3   ....get: Bool?
4   ....jpg: Bool?
5   ....ls: Bool?
6   ....put: Bool? | String? (iso8601)
7   ....stat: Bool?
8   ....rm: Bool?
9   ....mv: String?
10  ....cp: String?
11 }
12 ~ {
13   ....get: String?
14   ....jpg: String?
15   ....ls: {
16     .....info: {*}?
17     .....children: [{*}]
18   }?
19   ....put: String?
20   ....stat: {
21     .....file_limit: Int?
22   }?
23   ....rm: Bool?
24   ....mv: Bool?
25   ....cp: Bool?
26 }
```

Although plain HTTP supports, by design, the basic file-handling operations through the GET, PUT, and DELETE methods, those unrelated to the task of browsing web pages, such as enumerating the children of a directory or moving and copying a node from one path to another, are absent in the standard. The Web-based Distributed Authoring and Versioning specification [70] extends HTTP with methods for this missing functionality and with support for locking resources for

mutually exclusive access. Using the endpoints defined in WebDAV would have the additional benefit of allowing access to the virtual file system directly via the file manager of the user's machine, which usually handles the protocol natively under disguised names such as "Connect to Server" (Finder and Nautilus), or "Map a network drive" (Windows Explorer). Unfortunately, WebDAV (and plain HTTP) would not work for file upload operations as the relevant JOHN plug-in only returns a file upload URL that is the receiver of the `PUT` or `POST` operation. While this architecture is justified in the reduced latency for clients in uploading files (as there is no relaying party in the middle) and bandwidth savings on the server (which is significant if hosting the suite on a cloud provider like Amazon Web Services, that charges separately for the amount that it consumes), this esoteric choice precludes the use of WebDAV clients in write mode.

Therefore, to implement all the features required by the virtual file system while keeping open the ability to support WebDAV (read-only) in the future, the path-based endpoint deliberately avoids using the HTTP methods semantically, instead opting for a query-style interface using the `POST` method. Specifying one or more members in the query executes the corresponding command using only a single HTTP request.

Listing 5.2: The Path `GET` Endpoint

```
1 | GET /api/storage:${path}  
2 | ~ 302 FOUND
```

Since the HTTP `GET` method supports redirects, this endpoint is exposed as

well, enabling the use of resources stored inside the virtual file system directly in HTML-defined elements such as `` or `<video>`.

5.1.3 The SharedWorker and Indexed Database

Heavy users of cloud storage solutions often use them through a background process, constantly syncing their files and directories on their device and enabling browsing them from the native file manager with instantaneous navigation from one location to another. Enumerating the contents of a directory stored in a network disk, even using native software, introduces latency that makes traversing a remote file system unpleasant; adding a poorly optimized web interface into the mix worsens the experience further. Therefore, syncing, other than enabling offline usage, is the only way to experience a cloud service as if it were a local one. A key design goal of the University Cloud Suite, to be discussed later on in the chapter, is to provide a desktop-class experience to the user, and traversing the virtual file system makes no exception. Modern JavaScript runtimes provide functionality similar to processes and threads in the form of Workers. A **Worker** is a script that runs in a separate loop from the main script of the web page invoking it. The only way the two can communicate is through `window.postMessage()` calls, similar to how frames and their host document exchange messages. The web standards define three kinds of Workers in addition to the standard one:

- A **ServiceWorker** belonging to the whole origin and primarily used to im-

plement a few methods involved in cache management for enabling offline usage of web applications

- A **SharedWorker** is exactly like a plain **Worker** but guaranteed to be run only once at a time, and therefore sharing its handle across all its instantiations, on the same page or through pop-up windows and tabs.

The University Cloud Suite runs, across all logged-in pages, a **SharedWorker** named **SGWorker**, whose purpose is to initialize the local mirror of the virtual file system, periodically poll the server for changes and handle requests coming from the various components of the client holding a reference to it. As the suite is further developed, it may require a **ServiceWorker** for offline access, at which point the mirroring functionality will probably be best handled there. On the other hand, the current arrangement has the benefit of scaling gracefully in the absence of support of the more advanced kind of Workers since all it is needed is to change the name of the constructor invoked in starting the script: this is the case for Android devices, that at the moment of writing do not support **SharedWorkers**, but also rarely manage more than one window at the time, making the use of a plain **Worker** less problematic. Of course, the possibility of more than one instance of the **SGWorker** running requires synchronization across them to prevent, for instance, duplicate polling of the server within short intervals of time: this is achieved through the Web Locks API [61].

Listing 5.3: The Storage IndexedDB Table

```
1 aux_storage
2 ....sync_status
3 .....drives: [String]
4 .....timestamp: Date
5 ....index
6 .....{ID}
7 .....binary
8 .....bytes
9 .....delistedAt
10 .....drive
11 .....foreignId
12 .....id
13 .....listedAt
14 .....modified
15 .....name
16 .....path
17 .....shared
18 .....trashed
```

The virtual file system is persisted locally and offline through a couple of `IndexedDB` tables, one for storing metadata about the drives and the syncing status, and the other holding the list of files and directories, indexed through the same primary key generated on the PostgreSQL database it was fetched from. The `IndexedDB` is an advanced storage capability that is widely supported across browsers and, unlike rudimentary forms of persistence like cookies and `LocalStorage`, fully supports storing both structured JSON data and binary data.

The combination of the local database with the rest of the components discussed enables the experience of using any cloud storage provider, through a web interface, with similar responsiveness to that of using the local disk on a computer.

5.2 The Desktop-class User Experience

Choosing a desktop-class user experience, in the broadest terms, means mirroring the metaphors of a real-world office environment through software. A few examples will help crystallize this idea:

- Users do not have to remember arcane pseudonyms and codes to enter their workspace: they only carry a physical key and use it to unlock a door or cabinet when necessary.
- Users are not required to do only one thing at a time, and, when context switching, having to throw away all their work and start again as if they had just entered the building: they can keep track of as many activities with their related information as they can fit on their desk.
- Users do not need to care about what machinery was employed to produce and print some piece of information: they just pick it up and consume it.
- Users do not reach out to some all-knowing assistant to obtain the most basic stuff they use daily: what they leave on their desk in a specific place will stay there indefinitely, and, with time, they learn to rely on them.

A real-world office works this way because object permanence, direct manipulation, and form coinciding with function are all properties of physical reality. Software is

quite the opposite of all of the above, meaning it requires consistent and painstaking work and commitment from the developer to implement those qualities in a computer program. Building on the foundation of the JOHN-powered virtual file system, the University Cloud Suite strives to provide such desktop-class experience to its users:

- A password-less and username-less authentication experience through dedicated or platform authenticators using the Web Authentication standard.
- A window-oriented navigation experience between pages through pop-up windows spawned by a spatial window manager that remembers their size and positioning and propagates them across devices.
- A way to traverse the virtual file system, via the Object Tree Traversal (OTT) file manager, and to preview the most common file formats, using the Converters for LibreOffice-Accepted File Formats (CLAFF) library.

5.2.1 Password-less and Username-less Authentication

The Web Authentication API (WebAuthn) [66] is a standard for adopting the FIDO 2 specification in browsers and servers. All passwords, regardless of how strict their creation, storage, and update policy is, will always have the weakness of being a secret known and shared by multiple parties, of which the user is one. The premise of FIDO is to give up on passwords entirely and employ asymmetric

encryption techniques to allow the authenticating secret to be kept safe in one location and never shared:

- The application server generates a string value known as the challenge and makes it available to the client.
- A system capable of performing cryptographic operations and storing secrets safely, called the authenticator, either embedded in the user's device or a dedicated peripheral, generates a pair of keys for the domain name of the server, of which one is private, used to encrypt data blocks, and one public, used to decrypt them.
- The client requests the authenticator to encrypt the challenge using the generated private key and then sends the encrypted result, with the public key, back to the server.
- The server decrypts the challenge using the public key, verifies that it matches the original, and then stores the public key in its database for future log-ins.

Although this is a simplified description of the sign-up flow, especially for the generation of the key pair and validation of the challenge, it is sufficient to grasp its benefits for the robustness of the authentication scheme: all the information exchanged between the client and the server is of no use for identity theft. The colloquial term for this type of credential is “passkey”, reflecting its ambition to

displace passwords in the public's subconscious. On the web, a script communicates with authenticators through the Credential Manager API, whose `create` and `get` methods allow for, respectively, signing up and logging in through passkeys when specifying `publicKey` as the desired credential.

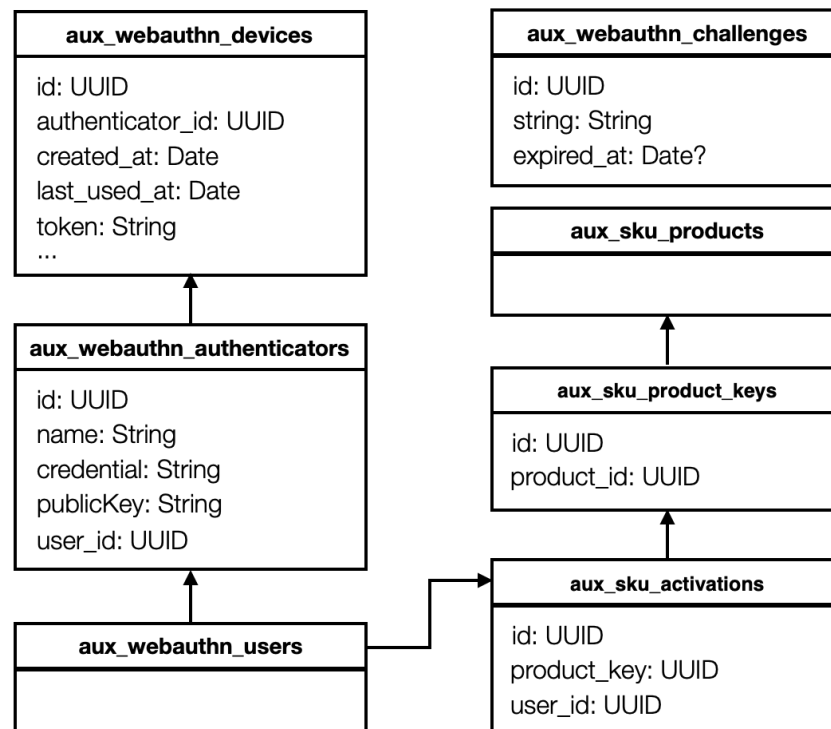


Figure 5.4: The WebAuthn Database Architecture

WebAuthn enables the University Cloud Suite to fully realize its vision of unlocking a user's virtual workspace with only a physical key, just like a real-world office. An **Authenticator** object models a passkey in the database, and it links a user session or **Device** with a **User**, which is the reference used by all other services in the suite to identify the owner of a resource.

Remember the Product Key?



You can link an existing account via the Product Key

What Product Key?
When you signed up your existing account, entering a Product Key was required. Enter the same code down here to create new credentials on this device, linked to your account. If you lost it, you can [contact us](#)

Product Key

XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX

The Product Key is made up of 32 characters grouped by 4

 
Back Forward

In need of help? Reference the [guide](#)

The University Cloud Suite © 2022-23 Alessio Giordano

Figure 5.5: Entering the Product Key

More than one **Authenticator** can be associated with a user: currently, as signing up is gated by a product key, this code is used to restore this connection when creating new credentials; ultimately, as the University Cloud Suite is going to be a subscription-based product, with the product key generated upon subscribing, it will act as a form of backup code in case a credential is stolen or lost.

Listing 5.4: Creating new credentials

```
1 import { WACHallenge } from "./WACHallenge.mjs";
2 import { HPDevice } from "./HPDevice.mjs";
3 import { HPBase64URL } from "./HPBase64.mjs";
4
5 let challengeObject = await WACHallenge.request();
6 let challengeBytes = challengeObject.bytes;
7 let credential = (await navigator.credentials.create({
8     publicKey: {
9         rp: WAAuthenticator.rp,
10        challenge: challengeBytes,
11        user: {
12            id: new ArrayBuffer(32),
13            name: authenticatorName,
14            displayName: authenticatorName
15        },
16        authenticatorSelection: {
17            residentKey: 'required',
18            userVerification: "preferred"
19        },
20        pubKeyCredParams: [
21            {type: "public-key", alg: -7},
22            {type: "public-key", alg: -257}
23        ], // ES256, RS256
24        extensions: {credProps: true}
25    }) as PublicKeyCredential | null);
```

The user's authenticator associates credentials to an origin (the domain name serving the HTTP requests) and a specified user identifier, which, more often than not, will coincide with the username specified during registration. To fully achieve a physical authentication experience, however, not having to devise and remember a username is essential.

Listing 5.5: The credentials returned by the `create()` method

```
1 let jsonCredential = {
```

```
2      id: credential.id,  
3      rawId: HPBase64URL.encodeBuffer(credential.rawId),  
4      response: {  
5          attestationObject: HPBase64URL.encodeBuffer((  
              ↪ credential.response as  
              ↪ AuthenticatorAttestationResponse).  
              ↪ attestationObject),  
6      clientDataJSON: HPBase64URL.encodeBuffer(credential.response  
              ↪ .clientDataJSON)  
7      },  
8      type: credential.type  
9  };
```

Thankfully, there is a unique identifier property as part of the credential object created by the authenticator, allowing its retrieval in the server.

Listing 5.6: Retrieval of previously created credentials

```
1 let credential = (await navigator.credentials.get({  
2     publicKey: {  
3         rpId: WAAuthenticator.rp.id,  
4         challenge: challengeBytes,  
5         allowCredentials: [],  
6         userVerification: "preferred", //required  
7     })  
8 }) as PublicKeyCredential | null);
```

Choosing a fixed user id of 32 bytes initialized to zero via `new ArrayBuffer(32)` for all users, the University Cloud Suite establishes a bidirectional relationship between any given physical authenticator and a user account.

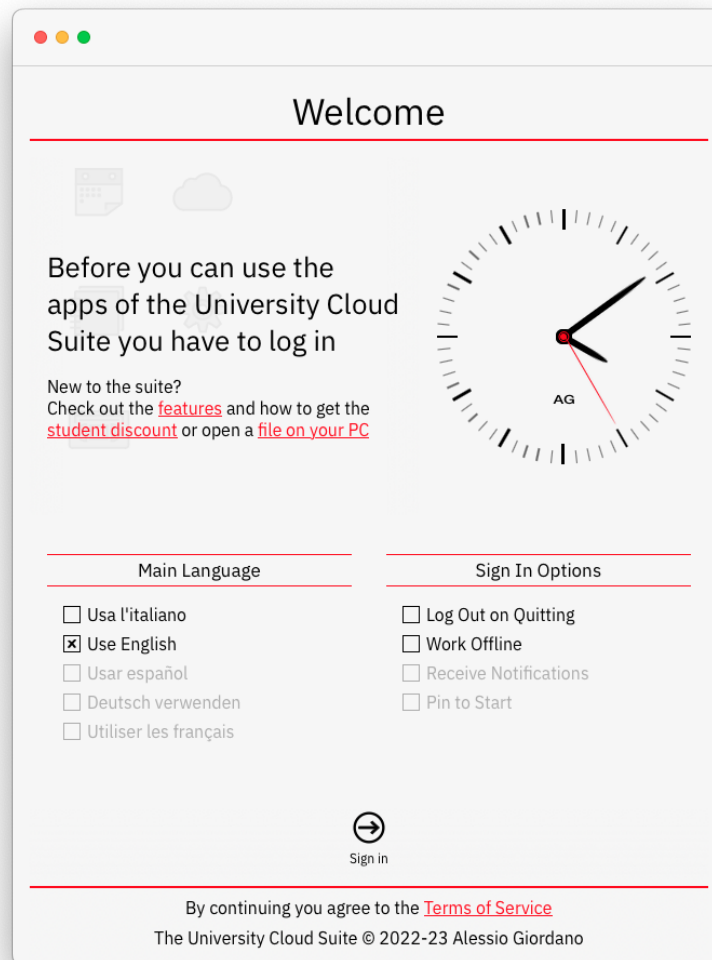


Figure 5.6: The Welcome page of the University Cloud Suite

On a less technical note, it is worth discussing how the user experiences this authentication flow, which merges signing in and logging in into a single action, beginning at a faded-out version of the homepage of signed-in users (Fig. 5.6). Before signing in, there is the ability to override the language settings and to specify whether the session should be ephemeral, storing the token as a session

cookie, and deleted after closing all tabs related to the domain.

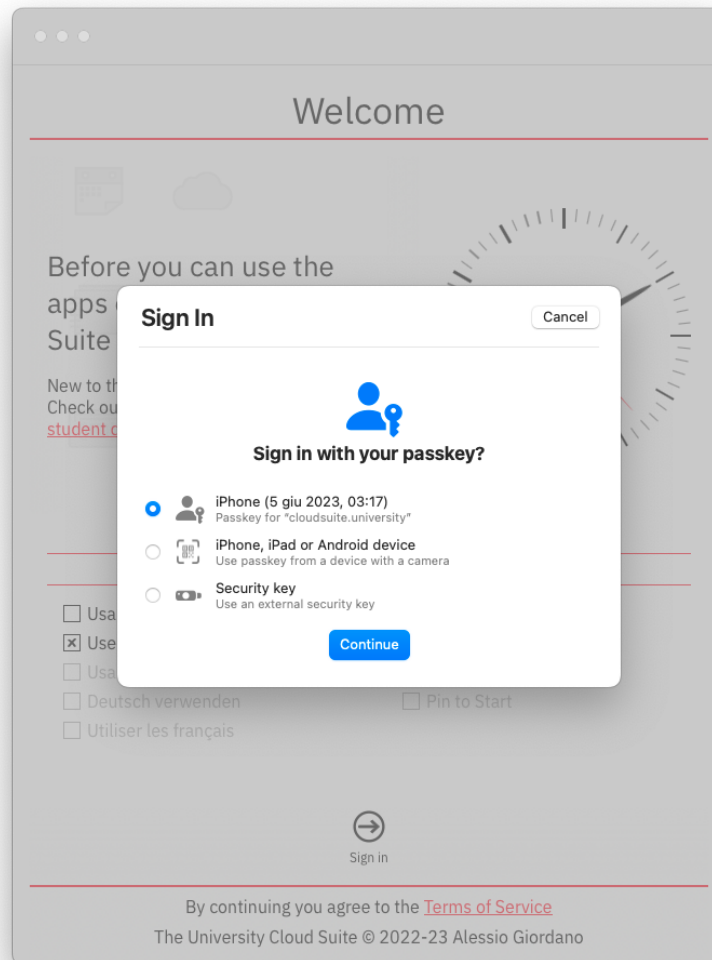


Figure 5.7: The prompt displayed by calling `credentials.get()`

Selecting the lone “Sign in” button at the bottom of the page invokes the `navigator.credentials.get()` and displays the variety of methods available to select an authenticator (Fig. 5.7). This step, handled by the browser, allows one to choose either the local authenticator on the device, the connection to another

device via peer-to-peer networking established through scanning a QR code, or a removable peripheral via USB or NFC.

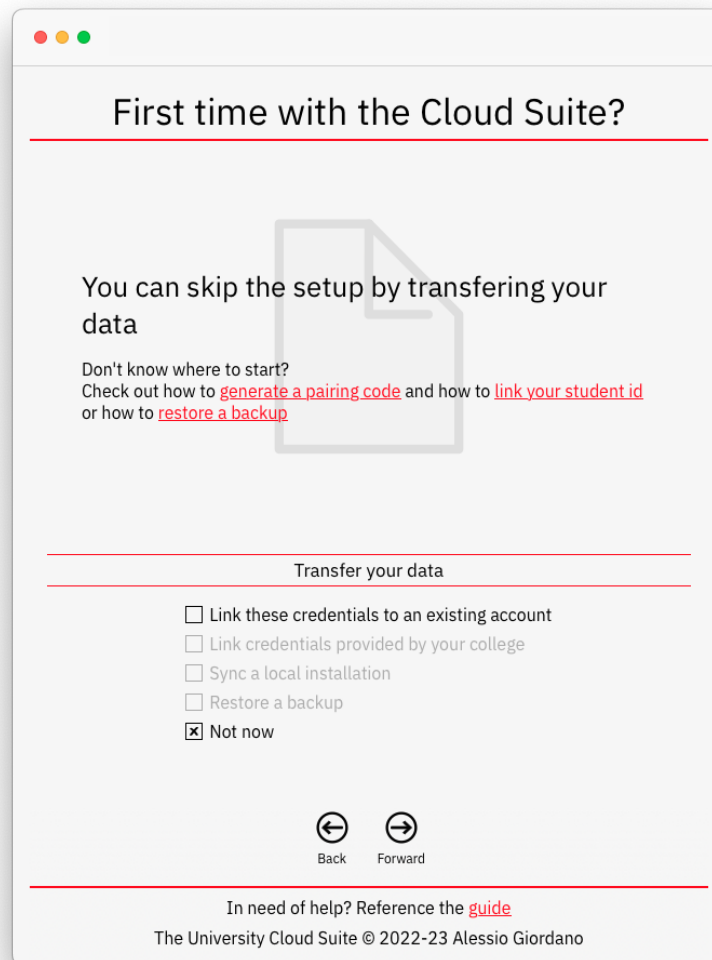


Figure 5.8: The Onboarding page of the University Cloud Suite

If the user provides a valid credential, then the signed-in page will fade in, but if the previous step is dismissed, or if the chosen authenticator contains no valid credential, the page will transition to the sign-up flow, presenting a series of options

to register the account, along with the ability to either go back and try again with a different credential or to move forward, invoking the `navigator.credentials.create()` and the product key entry screen seen before.

5.2.2 Building Desktop-class Web Components

The choice of semantic HTML elements defined in the specification is representative of the language's legacy as a format for documents and forms. Elements expected in applications, such as menus, toolbars, or lists with recycled views, are absent, while form elements, like select dropdowns and sliders, are complex to style as they require vendor-prefixed CSS rules. This absence has been filled over the decades by many libraries and frameworks. To deliver the desktop experience required by the University Cloud Suite, the Auxiliary Toolkit, providing a collection of essential user interface elements, has emerged organically during its development.

Custom Elements and the Shadow DOM

To ease the creation of custom elements and interoperability between user interface elements of different frameworks, the Web Components API has been part of the web standards since 2016.

Listing 5.7: An example element built using Web Components

```
1 export class CustomElement extends HTMLElement {  
2   constructor() {  
3     super();
```

```
4         const root: ShadowRoot = this.attachShadow({ mode: "
           ↪ open" });
5         root.innerHTML = "<slot name=attribute></slot>";
6     }
7     connectedCallback() {}
8     disconnectedCallback() {}
9     adoptedCallback() {}
10    attributeChangedCallback(name: string, oldValue?: string |
           ↪ null, newValue?: string) {}
11 }
12 customElements.define("custom-element", CustomElement);
```

By registering a custom element, it is possible to organize the logic and styling of what would have otherwise been a massive hierarchy of obscure `<div>` elements in a separate module, and by registering a shadow root in the element's constructor, it is possible to encapsulate the element even further by hiding away its hierarchy in a separate DOM tree associated with it, the same way `<audio>` and `<video>` elements have a hidden hierarchy for the browser-provided playback controls.

It is possible to reflect elements children of the custom element through the Slot API: any element containing a `slot=value` attribute will become, strictly from a layout perspective only, children of the `<slot name=value>` element contained in the Shadow DOM. All other elements will be assigned to the unnamed slot element if present. Finally, it is possible to use attributes to pass information down from outside the element into the Shadow DOM. CSS variables are also an option since, unlike all other CSS rules, they are inherited across the Shadow DOM boundary.

Listing 5.8: Usage of the sample custom element

```
1 <!DOCTYPE html>
2 <html>
```

```
3 <head>
4     <script type="module" src="CustomElement.mjs"></script>
5     <style>
6         * { --variable-attribute: "value" }
7     </style>
8 </head>
9 <body>
10     <custom-element data-attribute="value">
11         <p slot="attribute">value</p>
12     </custom-element>
13 </body>
14 </html>
```

Relying only on browser-based custom element primitives and having definitions separated in separate module source files allows for deliberately importing only those elements required by the current page. With an average occupation of about 20 kilobytes, reaching just shy of 60 for the heaviest source file, the toll on the network is minimal and consequently on the interpreter for parsing and executing the sources.

Components in the toolkit provide hooks via CSS variables to customize their appearance. Many of them are shared across components, such as the accent color, benefitting from centralizing their definition. The `TKCSSVariables` module defines them all, with default values, overridable in the page's CSS that employs them.

Buttons



Figure 5.9: Examples of `<link-button>`s and `<checkmark-button>`

The most essential UI component is the button, which lets users trigger actions as they wish. All buttons in the Auxiliary Toolkit either use directly or wrap a `<plain-button>` which provides the pressed and disabled appearance, handles input cancellation, and triggers the `pressed` event, as it executes the JavaScript source provided as its `action` attribute, upon clicking the button.

The `<link-button>` combines the `<plain-button>` semantics with a link-like appearance of the wrapped text nodes, using the current accent color.

The `<checkmark-button>` is a plain button featuring a checkbox on its left-hand side, remembering and handling the toggling of the mark automatically and providing convenient methods for getting and setting its state. More `<checkmark-buttons>` can be combined as part of a `<checkmark-group>` which provides a way to get and subscribe to the state of all of the contained buttons, as well as handling radio button functionality through the `radio` attribute and the mutual exclusion of sets of buttons through the `mutex` attribute.

Icons



Figure 5.10: Examples of `<icon-buttons>`

Icons are arguably the most recognizable element of a graphical user interface, and the Auxiliary Toolkit makes it easy to display one through the `<icon-image>` component. This element provides attributes such as `symbol`, `extension`, and `outline` to automatically construct the image's URL drawn as the icon. It also features a `SymbolFactory` singleton whose purpose is to handle the lifecycle of the URLs by storing their data in blobs that get reused for all elements sharing the same symbol, deleting them once all icons referencing them have been removed from the page. This system, dramatically reducing the page's memory usage, combined with a fade-in animation when the symbol is first shown on the page, virtually eliminates stutters and flashes related to displaying icons.

A variant of icons is the `<icon-banner>`, where it is possible to add any HTML element inside the component, making the icon image appear large and slightly translucent under them: this is useful for crafting headers at the top of a web page.

The primary usage of icons is as buttons, so it is natural to have a `<icon-button>` as part of the toolkit. Icon buttons combine the attributes of icons with those of buttons, as well as supporting labels and descriptions specifying the appropriate strings using the `label` and `description` attributes. As a counterpart to the pressing action, icon buttons also allow the primary label to be renamed, with the component handling the whole interaction, by specifying a `rename` attribute with the same semantics as the `action` attribute. Finally, by slotting an icon image inside the `overlay` slot, it is possible to insert a smaller icon in front of the primary one to signify a particular modality or to describe the action performed without introducing a text label. If preferred by the developer, it is possible to supply both text labels as slots rather than attributes.

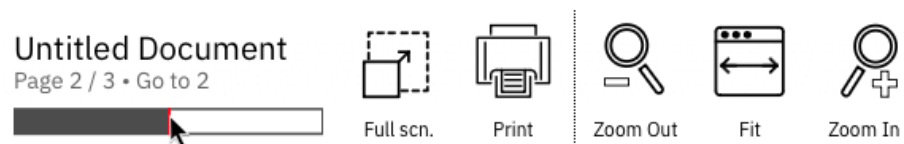


Figure 5.11: Example of an `<icon-bar>`

The toolkit provides several facilities to hold multiple icon buttons together:

- To arrange icons in a row, forming a toolbar, it is possible to use an `<icon-bar>`. This component also provides ways to slot elements on the left-end side, aligning them horizontally to the icon, label, or description rows defined by the buttons, using the appropriate slot names.

-
- Laying icons on a scrolling grid has been a staple feature of graphical file managers since the earliest days of computing. The `<icon-view>` component provides this layout, with the ability to set a `disabled` that propagates across every single icon button contained in the view.
 - As an example of a complex application of the slot-based Shadow DOM API, the `<icon-group>` lets one or more grids be defined in a comma-separated `sections` attribute, creating dynamically a corresponding slot to supply the icons, and another with a trailing `-header` to provide the heading displayed above the grid. Each section is visually split from the following one through a line separator of the same tint of the accent color.
 - Lastly, the `<icon-desktop>` is a grid layout resembling a computer desktop, with icons flowing from the top-left downwards to fill the column and then back to the top for successive columns. It is possible to arrange icons on the opposite end of the desktop through the `alignment=right` attribute. When the component is too small to display the entire grid, it overflows horizontally with a slight gradient hinting at the disappearing icons. Because the component uses the `scroll-snap` CSS property, the scrolling value of the element always aligns the right edge of the view with the closest column of icons.

Analog Clock



Figure 5.12: Examples of `<analog-clock>`

While certainly not required for a desktop-class experience, to fulfill the aesthetics envisioned for the University Cloud Suite, a customizable live `<analog-clock>` is part of the toolkit. As its name suggests, the component displays the current time via an analog clock and updates it every second by using the `requestAnimationFrame()` function, enabling the smooth animation of the seconds' hand, mimicking the subtle oscillation a real-world clock hand would make. It is possible to customize the visibility of the clock hands related to hours, minutes, and seconds by setting the corresponding attributes to `hidden`, or to a particular integer value to set their position fixed instead. To set the visibility of the marks, analogous attributes are available by appending the `-marks` suffix. As a convenience for setting all three hands, there is a `timestamp` attribute, accepting a timestamp string like the one generated by `Date.now` and the `getTime()` method of `Date` objects. As for customizing the appearance, the colors of the hands and

the marks inherit the tint of the primary, secondary, and accent colors, while the logo text on the bottom half of the watch face is supplied through the `logo` attribute.

A final note on the screensaver capability of the clock: when the attribute is present, a button appears on the top-right corner of the clock that, when pressed, will enter a full-screen view of the clock, telling the time every hour with a series of beeps. If the screensaver attribute is not left empty but supplies a URL of an audio file or a live stream decodable by the browser, it will play while the screensaver is full screen. Because the analog clock requests a lock upon entering full-screen through the Screen Wake Lock API, supported browsers will not lock the device while the screensaver is visible.

Localization

Adapting a piece of software to the language of a particular market is a sign of the seriousness of the vendor providing it. The Auxiliary Toolkit supports localization through the `<localized-string>` component, which renders the correct string value based on the `lang` attribute of the root element of the document. This localization system harnesses CSS variables and the `content` CSS property to project the currently resolving value for the variable constructed by concatenating the contents of the `for` attribute with the `-string` suffix inside the custom element through the `content` property.

Listing 5.9: Example `strings.css` file

```
1 :root, :root[lang="en"], :root[lang*="en-"] {  
2     /* Use quoted strings */  
3     --hello-world-string: "Hello World";  
4 }  
5 :root[lang="it"], :root[lang*="it-"] {  
6     --hello-world-string: "Ciao mondo";  
7 }
```

Changing the variable's value depending on the `lang` attribute in a separate strings CSS file will propagate it through the Shadow DOM and into the component. If the variable cannot be resolved, the text node inside the localized string will become the fallback value.

Other components of the toolkit, like icons, support a convenience `localized` attribute, which accepts a localized key prefix used by the element to fetch automatically all the localized strings needed.

Page Layouts

The toolkit supports three main page layouts that define the appearance of all the screens of the University Cloud Suite:

- The `<web-page>` layout is used by the welcome page of logged-out users and the home page of logged-in users. It features extensive headers and footers, a toolbar area just above the latter, and when put inside a `<page-manager>` enables backward and forward navigation between more than one page, identified by its order inside the manager, which also takes care of

scrolling back to the top and animating the transition when switching pages.

- Every content window spawned by the suite features a status bar at the bottom with a breadcrumb bar containing the path of the resource and a loading indicator that pulses while data is **fetch**ed from the network. The `<status-bar>` element provides slots and attributes to set both and displays itself on top of its unslotted children.
- Applets like preference panes and application dialogs appear as pop-up dialogs using the `<dialog-app>` container. This element inverts the status bar layout by putting the path bar and the loading indicator at the top of the window and also makes the path components slightly larger and modifiable through drag and drop. This layout leaves space at the bottom for a message area and a series of toolbar buttons. Content can be slotted in either as part of the form area just below the path bar (or in place of it if absent) or below the form area altogether.

Miscellaneous Components

The Auxiliary Toolkit provides many more components, some of which are hard to describe through static pictures and text description alone, others of less significant importance: Reusable views related to suite-specific functionality, like the `<info-panel>` and the `<drive-manager>`. Input mechanisms used in dialog windows, like the `<plain-field>` and the `<dotted-field>` for inserting text or the

`<menu-item>` and `<popup-menu>` for building pull-down menus, and input mechanisms used in content windows, like the `<progress-bar>` and the `<scrub-bar>`. Elements described previously as part of the layout views, such as the `<path-component>`, the `<path-bar>`, and the `<pulsing-loader>` featured in status bars, or the `<page-header>`, `<section-header>` and `<page-footer>` used in web page layouts. Complex behavioral elements like `<column-scroll>`, a component that enables multiple columns, laid one next to the other, to scroll using just the main scroll bar of the viewport.

Windows and Window Management

The crown jewel of the Auxiliary Toolkit is its advanced windowing capability. Browsers have provided for a long time the `window.open()` call for spawning tabs and pop-up windows, but their abuse for displaying aggressive advertising has led over the years to the introduction of significant restriction in the allowed circumstances resulting in a successful opening. Those circumstances must involve an explicit user action, which, for the subsequent few hundred milliseconds, allows performing sensitive actions such as calling the open method.

The toolkit provides a convenience to spawn windows in the form of the `<window-button>` with the `view`, `path`, and `state` attributes used to construct the URL of the pop-up and the `top`, `left`, `width`, and `height` attributes to set its size and position on the screen. The component works by capturing the `pressed` event

bubbled up by any toolkit button placed inside it, most likely an icon button, and focusing on a stored window proxy if available or opening a new one. The principles of the desktop metaphor require that the windows placed on the screen have a unique bidirectional relationship with the content they display; in other words, to the user, they are the content and must never appear duplicated to avoid breaking the illusion. To prevent multiple window buttons from spawning duplicate pop-ups, the component uses the target argument of the opening procedure to supply a unique string constructed from the `view` and the `path` attributes: if an existing instance exists, even if the button has no reference to it, the browser will bring it to front and reload it with the provided URL.

As the opened window is resized and moved around the screen, this information has to be reflected back to the button for successive openings to persist this information and maintain the illusion of the metaphor. All pages in the University Cloud Suite designed to be opened inside pop-ups have their body embedded inside a `<popup-window>` component whose purpose is to listen to events related to the window's lifecycle and relay them back to the button. Unfortunately, while the resize event is dispatched by the browser, there is no such equivalent for window movement, forcing the `<popup-window>` to poll periodically to access this information. In addition to the metrics, the component also takes care of quitting the window when appropriate:

- Upon receiving a `logout` event, all windows automatically close.
- Immediately after a window is opened by a window button, the latter sets a `windowIdentifier` value in the global scope of the window and dispatches an `opened` event with this value. Existing windows receiving the event will quit if they match the newly spawned view-path pair but do not match the identifier.

While the term “event” was used to describe the communication between components, the traditional event model of the browser does not apply to multiple windows. Broadcast Channels are a relatively recent addition to the web standards, allowing messaging parsing to occur across any window or worker by creating a channel with the same name. Messaging to the channel will use the same `postMessage()` method as the one provided by window proxies and frame elements.

Each `<window-button>` uses Broadcast Channels to communicate with the window to receive metric updates, and it is also available to any custom script written to integrate the toolkit component with the server delivering the solution: they can listen to the `metrics` event dispatched by the pop-up to update the values stored in the database or send it when such value is first fetched or after it changes from other sources. This system ensures complete decoupling of the toolkit from the custom server.

The University Cloud Suite uses a second `SharedWorker` named the `WMWorker` that performs a similar role in relation to window management that the `SGWorker` does for the virtual file system. It periodically checks with the server for changes in the windows' state, updates those when the `metrics` event is detected, and synchronizes a local copy of all the states using an `IndexedDB` table.

5.2.3 Bringing Spatial File Management to the Web

When a logged-in user navigates to the home page of the University Cloud Suite, they see what resembles a computer desktop and, next to it, an analog clock.

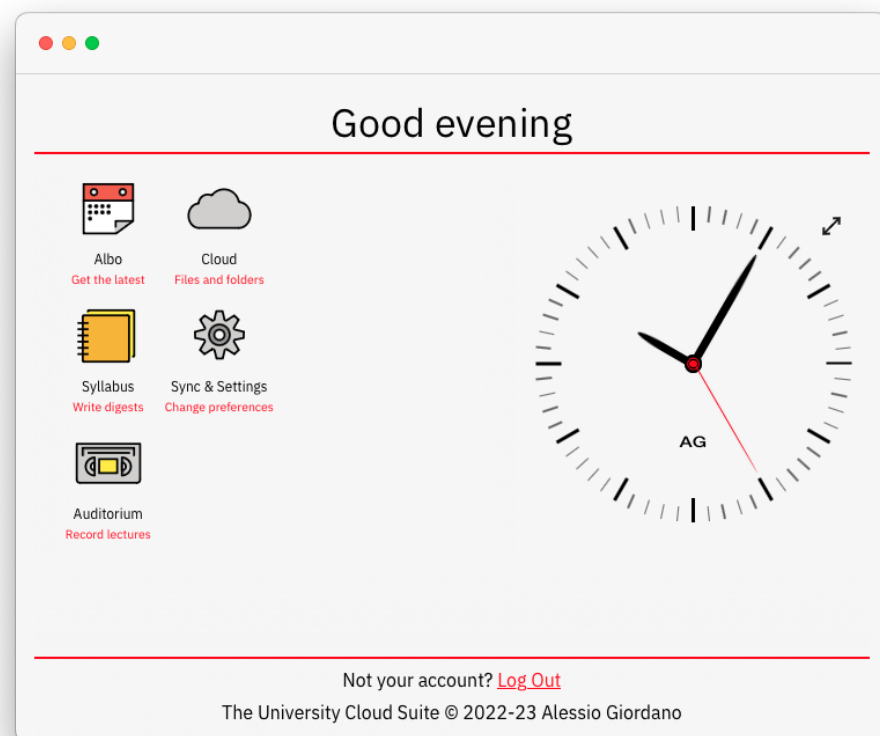


Figure 5.13: The Desktop of the University Cloud Suite

The grid of icons (Fig. 5.13) consists of the entry points to the system, with the first three icons being quick shortcuts to creating new Albo, Syllabus, and Auditorium documents (discussion of which deserves a separate section), followed by a Cloud icon for reaching the root of the virtual file system, and finally an icon for Sync & Settings which launches the control panel for the suite.

The Object Tree Traversal

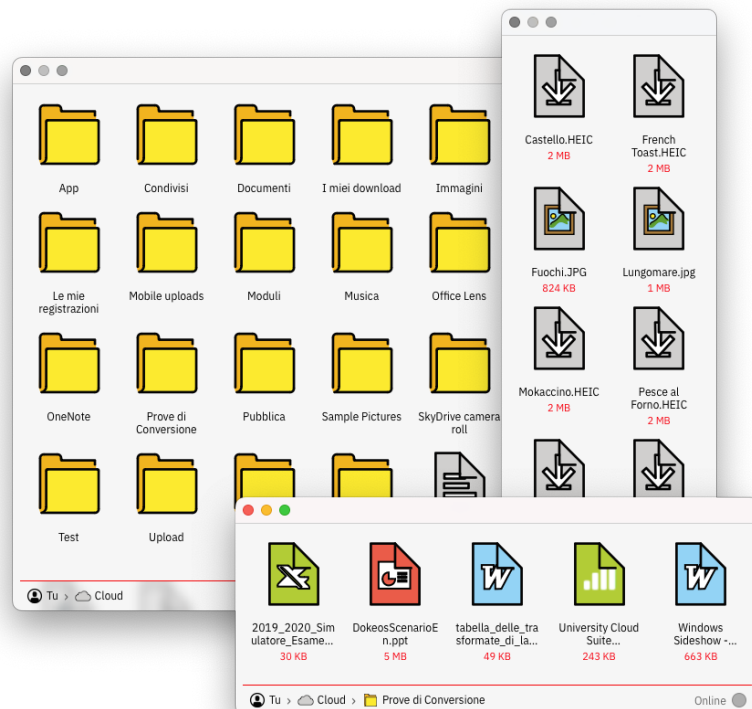


Figure 5.14: Three windows of the Object Tree Traversal

Pressing the “Cloud” button opens the root (/) folder in a window (Fig. 5.14) of the Object Tree Traversal (OTT), the spatial file manager of the University

Cloud Suite. Spatial means it relies entirely on the desktop metaphor, provided by the Auxiliary Toolkit in the case of OTT, to display the folder hierarchy to users rather than managing it directly through a navigation system or a master-detail split view. The most popular desktop file managers still in use today, Explorer for Windows systems, Finder for Macintoshes, and Nautilus for GNOME-based Linux desktops, have all begun as spatial file managers and transitioned to the navigation model, inspired by web browsers, and for a good reason: on the large hard drives found in contemporary desktops, the hierarchies of files have exploded in depth, making them unwieldy to manage spatially. The University Cloud Suite does not suffer from this problem for a few reasons:

- Since the size of cloud storage is orders of magnitude smaller than that of local disks, hierarchies are consequently much more compact.
- It is rare for computer programs to manipulate cloud storage directly to store their deep and convoluted hierarchies of private data, which is a substantial source of file system pollution on its own.
- The virtual file system incentivizes the connection of small, dedicated cloud providers exclusively for storing university-related documents. Once the scope of a hierarchy is bounded to a specific domain, the opportunity for an explosion in the depth of the directories is limited.

If the file system hierarchy is kept under control, then a spatial file manager is

better suited to map to the natural way of thinking about one's documents and information.

The Converters For LibreOffice-Accepted File Formats

Downloading a file, which is always possible by holding down the ALT key on the keyboard while selecting its icon, is only left as a fallback scenario for those instances where the suite cannot display its contents.

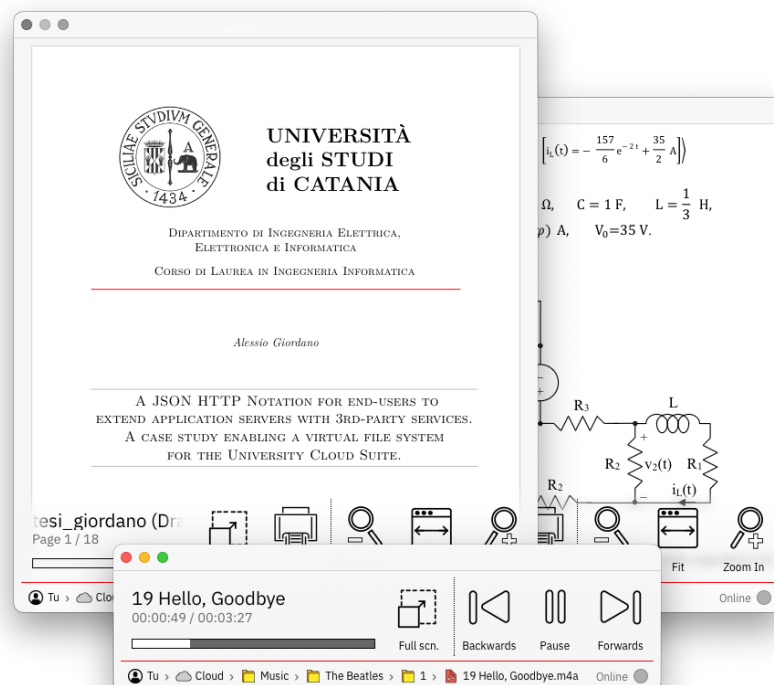


Figure 5.15: The PDFViewer.html and the MediaPlayer.html in action

The browser is natively capable of decoding a wide range of multimedia files through the use of `` and `<video>` elements (the latter of which is equally

usable with audio files), making the creation of the `PictureViewer.html` and the `MediaPlayer.html` relatively straightforward.

A similar argument can be made for displaying web pages and plain text files since those are easy to embed in an `<iframe>` with no further work required for creating the `WebViewer.html` and the `TextViewer.html`.

Unfortunately, most documents dealt with by students are in none of the formats above. Studying is a form of knowledge work and, as such, happens on office documents and PDFs, which the browser is not able to open natively.

For PDFs, the Mozilla Foundation has maintains a full implementation of the document format in a JavaScript open-source library [35], which the University Cloud Suite uses as the foundation of its `PDFViewer.html`. This library, as one of its many features, also supports the ability to project the pages of the viewed PDF as if they were slides in a presentation deck, which is a feature that has an interesting consequence: if a presentation is converted into PDF format, retaining its aspect ratio and appearance, then the issue of displaying it is already solved.

The LibreOffice open-source office suite not only can understand all the file formats encountered by students but also convert between them and do so directly from the command line in headless operation. Additionally, the headless mode can be left running continuously as a background process and controlled via a networking protocol called the Universal Network Objects (UNO) [59].

The Converters for LibreOffice-Accepted File Formats (CLAFF) is a Swift library, built in tandem with the rest of the University Cloud Suite, that provides idiomatic access to the file conversion capability of LibreOffice UNO servers. Rather than implementing the networking protocol itself, since, currently, the suite runs all of its services, including the database and the headless office daemon, on the same cloud virtual private server, it can access the Python runtime bundled with the LibreOffice program, which comes already configured with bindings for UNO. The library also contains a series of structured types that deal with filter names, which are the file types of the source and destination files, type-safe. Finally, interaction with the Python code is wrapped in asynchronous methods, making the library work well with modern Swift code.

Listing 5.10: Sample usage of the CLAFF Swift library

```
1 import CLAFF
2 if let desktop = FileManager.default.urls(for: .desktopDirectory,
    ↪ in: .userDomainMask).first {
3     try await Conversion.Text.toPDF.execute(from: try Data(
        ↪ contentsOf: desktop.appending(component: "Test.docx"))).
        ↪ write(to: desktop.appending(component: "Test-Doc-\(Date()
        ↪ .iso8601).pdf"))
4     try await Conversion.Spreadsheet.toHTML.execute(from: try Data(
        ↪ contentsOf: desktop.appending(component: "Test.xlsx"))).
        ↪ write(to: desktop.appending(component: "Test-Xls-\(Date()
        ↪ .iso8601).html"))
5     try await Conversion.Presentation.toPDF.execute(from: try Data(
        ↪ contentsOf: desktop.appending(component: "Test.ppt"))).
        ↪ write(to: desktop.appending(component: "Test-Ppt-\(Date()
        ↪ .iso8601)
6 }
```

The usage of CLAFF in the University Cloud Suite is straightforward: the

convenience `GET` request to a storage path of the virtual file system supports the specification of a file format in the query portion of the URL, which will trigger the download of the file on the server and its conversion through CLAFF into the desired format if the two do not match.

The file viewers discussed before always add the desired file format in the query of URLs they load so that, for example, every word document or presentation opened using the `PDFViewer.html` is guaranteed to be converted losslessly to a PDF.

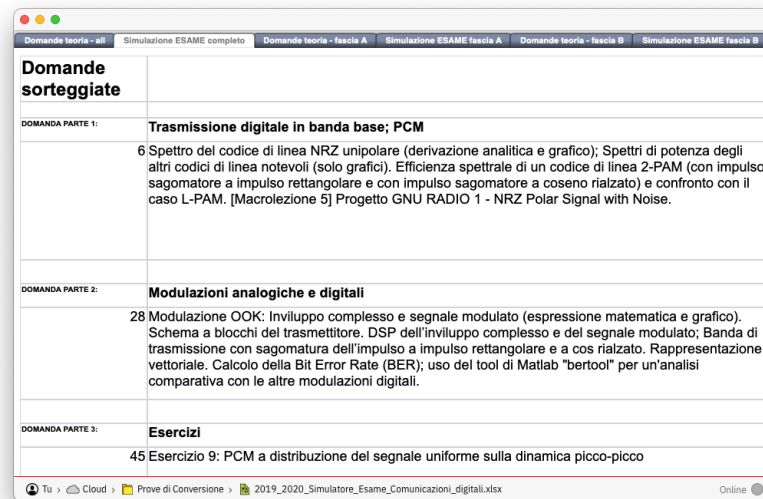


Figure 5.16: The `WebViewer.html` viewing a complex spreadsheet

For spreadsheets, the best file format for a lossless conversion is HTML, which also allows the injection of custom CSS to display only one sheet at a time and to render the sheet links as a fixed bar at the top of the window (Fig. 5.16).

5.3 Building Tools for Students

The technology created to support the University Cloud Suite, although interesting on its own, is only fully realized if it is part of the larger vision of building an integrated family of tools for the university student.

It is interesting to point out how suites of computer programs tend to come in triplets: Word, Excel, and PowerPoint forming the core Microsoft Office package; iPhoto, iMovie, and Garageband constituting the iLife suite; PhotoShop, Lightroom, and Premiere being the heart of Adobe's Creative Cloud.

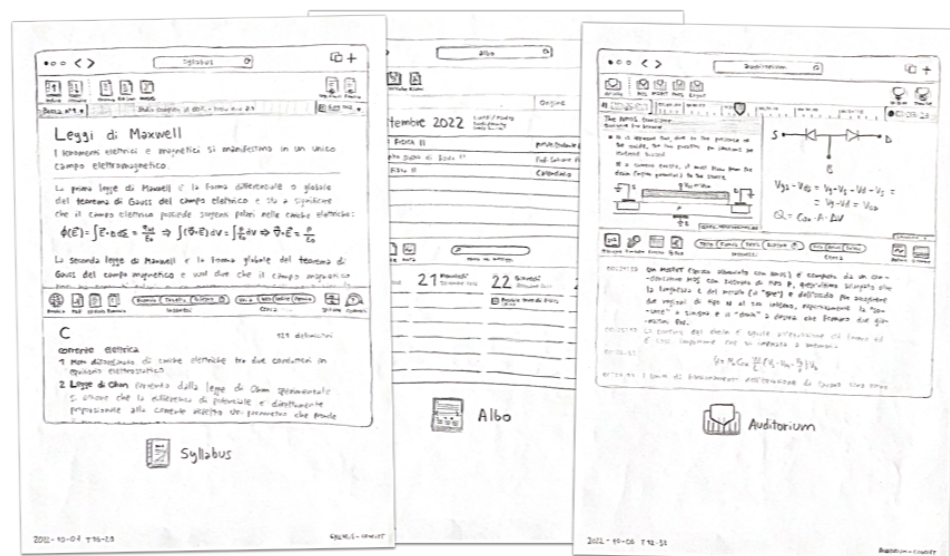


Figure 5.17: The original vision of the University Cloud Suite, circa 2022

The question that arose early in the planning process of the University Cloud Suite was: “What are the three crucial apps to students?” To answer that, one has to be reminded of the collective experience shared by all students, regardless

of their vocation:

- The academic institution makes available material, schedules, and news updates to every student through official channels.
- Students attend classes during which they capture as much information as possible for later reference.
- Material captured in the previous two points is consulted, summarized and organized with the goal of passing an exam.

With the problem clarified like this, it becomes crystal clear what the University Cloud Suite should be all about:

- A program to index and fetch information from email, the web, and proprietary platforms with the help of custom-built JOHN plug-ins, to be known as Albo.
- A universal recording tool, named Auditorium, for capturing audio, opened windows, and timestamped text during a class, and allowing to replay it at a later date.
- An application that, taking a textual description of the contents of a course, can generate a book-like table of contents to fill in with well-organized lecture notes. This software is accordingly called Syllabus.

Underlying all three programs is, of course, the virtual file system and the file manager and viewers built on top of it. In other words, to be talking of file formats in the context of the three apps is just to be describing a directory, whose “file” extension is merely an indication to display its contents a bit differently from a traditional folder:

- An agenda view that uses the contained files’ modification date to group them by calendar date.
- A theatre view that arranges content along a timeline depending on the file’s modification date (used as the ending point) and its duration (used, if available, as the starting point).
- A binder view that features a large title page and table of contents, followed by every single file (textual for the most part) rendered inline, according to its position in the internal hierarchy.

At the moment of writing, the University Cloud Suite features only an early implementation of the programs sketched out here, but nevertheless, it is worth examining each one closer.

5.3.1 Albo

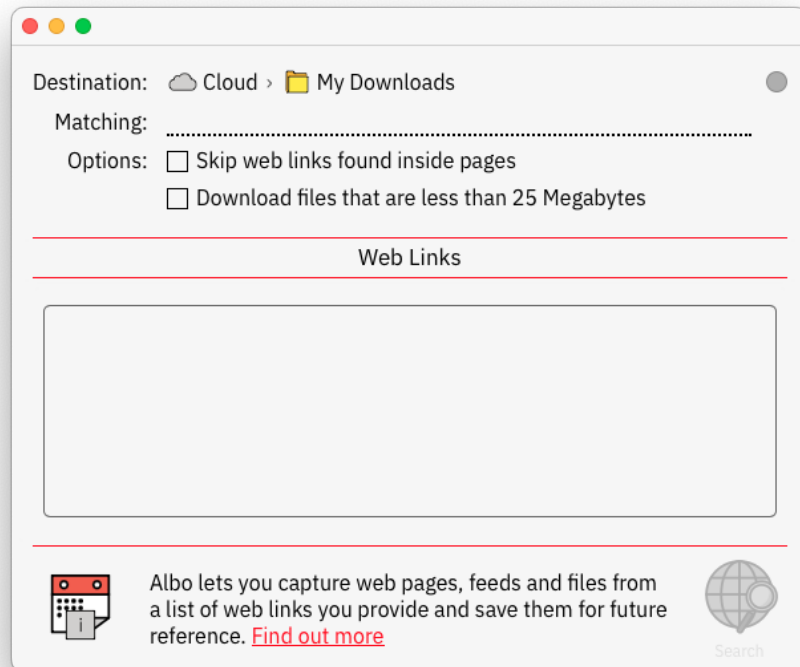


Figure 5.18: The Albo dialog window, as it appears in build 128

Albo (Fig. 5.18) allows users to crawl a list of URLs for the files they contain. Using the same HTML and XML parsers featured in the JOHN Runtime, the crawler endpoint at `/api/albo/crawler` can understand web pages and RSS feeds and capture the URLs found in `<a>`, `<link>`, and `<enclosure>` tags. Albo will follow pages, based on their `Content-Type`, up to a specified level, after which the collected URLs will be processed and stored in the specified directory. If the file is under 25 megabytes, usually the limit for email attachments, it is downloaded to

the server and uploaded to the primary cloud storage provider at the path specified in the crawl request. If the file is over the limit, or if the option to download files was not selected in the dialog before sending the crawl request, then the file is only indexed without downloading it, using the `albo.html` format.

Listing 5.11: The construction of an `albo.html` link in the University Cloud Suite

```

1 let constructedStubFile = workingDirectory.appendingPathComponent(
    ↳ url.lastPathComponent + ".albo.html")
2 let urlString = url.absoluteString
3 guard urlString.count <= 2083 else { throw GetRequestError.
    ↳ urlTooLong }
4 do {
5     try ""
6     <!-- Albo-\(AlboShortcutHeader.url.rawValue): \(urlString)
    ↳ -->
7     <!-- Albo-\(AlboShortcutHeader.cors.rawValue): \(
    ↳ requiresSameOrigin) -->
8     <!-- The University Cloud Suite © 2022- \(Calendar.current.
    ↳ component(.year, from: Date())) Alessio Giordano -->
9     <!DOCTYPE html>
10    <html><head>
11        <meta http-equiv="refresh" content="0;URL='\(
    ↳ urlString)' />
12        <title>\(url.lastPathComponent)</title>
13        </head>
14        <body></body>
15        </html>
16    "".write(to: constructedStubFile, atomically: true,
    ↳ encoding: .utf8)
17 } catch {
18     try? FileManager.default.removeItem(at: workingDirectory)
19     throw error
20 }
21 fileToUpload = constructedStubFile

```

The concept behind the idea was to choose a universal file format for expressing redirection, so the natural solution is a standard HTML file, understood by all

platforms and browsers, containing an HTTP equivalent tag in its header that redirects to the resource in addition to a comment at the top of the source file holding the same data, but in a custom format that is easier to read. Similarly to CLAFF, the GET endpoint on a path of the virtual file system is aware of this capability and can unwrap an Albo file into an HTTP redirect on the request itself.

5.3.2 Auditorium

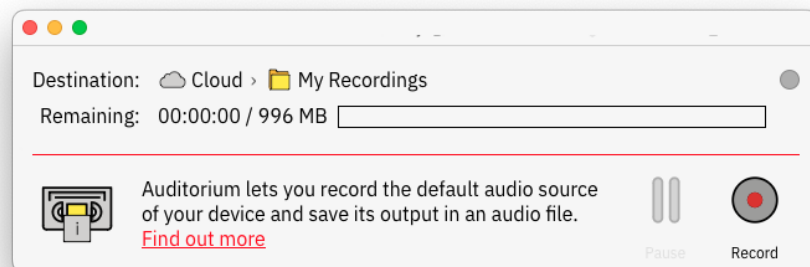


Figure 5.19: The Auditorium dialog window, as it appears in build 128

Auditorium (Fig. 5.19) is currently a simple voice recorder that uploads the final recording to the specified path of the virtual file system. An interesting note about this program is its usage of the Origin Private File System (OPFS) as the temporary storage for the audio recording. OPFS is a local and sandboxed file system with full read and write privileges available to any JavaScript worker. If the Auditorium window is closed at any point during the recording or the uploading of the audio file, the next time that Auditorium is launched, it checks if the OPFS

holds any leftover recordings. If one is present, then the option to download it locally on the device or try the upload again is shown to the user instead of the usual recording options. On a note, this also means that the University Cloud Suite uses all the persistence methods of the browser: Cookies, LocalStorage, SessionStorage, IndexedDB, and OPFS.

5.3.3 Syllabus

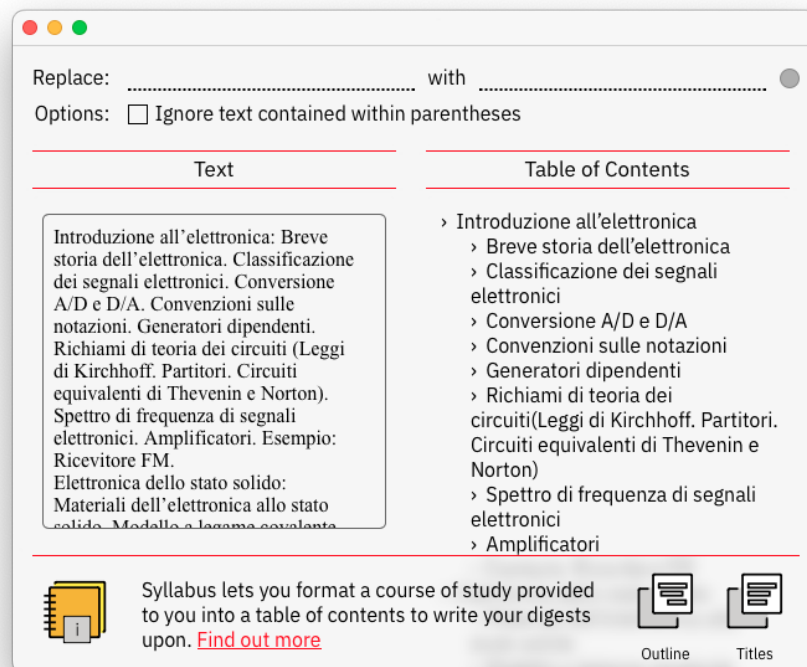


Figure 5.20: The Syllabus dialog window, containing the course description for Electronics (in italian)

word; after the first pass is complete, a second one cleans up the output, removing empty levels in between extremes. Indentation parsing considers a space character as one level of nesting, while a tab is equivalent to four spaces. By default, the colon symbol “:” is interpreted to indicate that the preceding text is on a higher level than what succeeds it; other punctuation symbols are processed in decreasing levels of importance, from the full stop “.” to the semicolon “;” to the comma “,” and in any case with less priority than whitespace. Parsing is customizable through the dialog by providing a list of substitution pairs or requesting the exclusion of content between parentheses from the output.

Chapter 6

Conclusion

If there is one common theme throughout this entire dissertation, it has to be that building flexible software is a choice that demands a very high price in complexity, but it is also a choice that equally pays back in the long run with an empowered user base that, by shaping the software product to their own needs, further makes it helpful and appealing for more people and a longer lifespan.

The JSON HTTP Notation is a language, a runtime, and a development environment that aims to make that complexity a little less daunting, with a syntax that is easier for power end users to adopt in developing plug-ins and for software vendors to implement in their custom runtime if they so wish.

It is hard to speculate on the future evolution of a solution born almost accidentally to solve the particular issues encountered during the early development of the University Cloud Suite, but for sure, being a declarative language, JOHN will have to expand as it gets adopted in more aspects of the suite that may bump into the limitations of the provided interfaces. There are many possibilities available when

it comes to the result subscript syntax, like the ability to use regular expressions to transform string values, which is currently absent, or allowing markup variables, like the output of XML or HTML documents, to be subscripted through selectors, which are a more natural way to deal with a DOM than the integer subscripts supported currently.

On the other end, the challenge to keep the specification simple so that it remains feasible to implement by many different vendors, unlike the JavaScript runtimes, discarded in the State of the Art chapter for being too complex: for example, is it necessary to support both an inline result subscript syntax and the nested one when the same plug-in can be written in both but the former, although much more succinct from the plug-in developer standpoint, requires implementing a more complex subscript parsing logic in the runtime? Such questions can only be dealt with through evidence and the concrete needs of all the stakeholders of the language.

The real question is how the University Cloud Suite will evolve, as all software projects experiencing long development periods inevitably diverge from their original roadmap and feature set to take advantage of the opportunities presented in the market. An interesting case is the natural language parser of Syllabus, conceived before the public release of the Chat-GPT product by OpenAI in late 2022 and, despite working sufficiently well, completely obsoleted by it. Also noteworthy is the recent popularity of large-screen e-ink devices with digital inking capabilities,

whose (re)markable writing experience is only matched by their lacking software ecosystem: here, the University Cloud Suite, by virtue of being the integration point of a student's entire academic career, is uniquely positioned to enhance this experience.

What will never change is the core set of values underpinning all the endeavors described here: that each computer user, regardless of the open or closed nature of the software being used, deserves to be in control of their experience and that this, ultimately, benefits the software maker as well.

The University Cloud Suite is online at <https://cloudsuite.university/> and the JOHN Language Specification, the JOHN Runtime, and the JOHN Editor are available at <https://john.cloudsuite.university/>.

References

- [1] Adobe - ExecuteAsModal Details. https://developer.adobe.com/photoshop/uxp/2022/ps_reference/media/executeasmodal/.
- [2] Adobe - UXP for CEP Developers. https://developer.adobe.com/photoshop/uxp/2022/guides/uxp_for_you/uxp_for_cep_devs/.
- [3] Adobe - UXP Manifest v5. https://developer.adobe.com/photoshop/uxp/2022/guides/uxp_guide/uxp-misc/manifest-v5/.
- [4] Advanced Operators | Swift.org Documentation. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/advancedoperators/>.
- [5] Announcing UXP in Photoshop. A high level overview of UXP the platform | Adobe Tech Blog. <https://blog.developer.adobe.com/announcing-uxp-in-photoshop-288496ab5e3e>.
- [6] API Documentation Made Easy - Get Started | Swagger. <https://swagger.io/solutions/api-documentation/>.

-
- [7] App | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/app>.
 - [8] Apple Documentation Archive - Mac Automation Scripting Guide: About Scripting Terminology. <https://developer.apple.com/library/archive/documentation/LanguagesUtilities/Conceptual/MacAutomationScriptingGuide/AboutScriptingTerminology.html>.
 - [9] AppStorage | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/appstorage>.
 - [10] async-http-client: HTTP client library built on SwiftNIO. <https://github.com/swift-server/async-http-client>.
 - [11] Async HTTP Client Reference. <https://swift-server.github.io/async-http-client/docs/current/AsyncHTTPClient/index.html>.
 - [12] Automatic Reference Counting | Swift.org Documentation. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/>.
 - [13] Binding | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/binding>.
 - [14] Browser Extensions - W3C Draft Community Group Report 10 October 2022. <https://browserext.github.io/browserext/>.

-
- [15] Closures | Swift.org Documentation. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/advancedoperators/>.
- [16] Codable | Apple Developer Documentation. <https://developer.apple.com/documentation/swift/codable>.
- [17] Code Samples — UXP for Adobe Photoshop. https://developer.adobe.com/photoshop/uxp/2022/guides/code_samples/.
- [18] Delegates and Data Sources - Apple Documentation Archive. https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html#//apple_ref/doc/uid/TP40010810-CH11.
- [19] ECMA-262 ECMAScript® 2023 language specification 14th edition, June 2023. <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>.
- [20] ECMA-404 The JSON data interchange syntax 2nd edition, December 2017. <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.

-
- [21] Encoding and Decoding Custom Types | Apple Developer Documentation. https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types.
 - [22] Environment | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/environment>.
 - [23] EnvironmentObject | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/environmentobject>.
 - [24] Extension Anatomy | Visual Studio Code Extension API. <https://code.visualstudio.com/api/get-started/extension-anatomy>.
 - [25] GitHub - WebAssembly/WASI: WebAssembly System Interface. <https://github.com/WebAssembly/WASI/tree/main>.
 - [26] Home - OpenAPI Initiative. <https://www.openapis.org>.
 - [27] Introspection - Concepts in Objective-C Programming - Apple Documentation Archive. https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Introspection/Introspection.html#//apple_ref/doc/uid/TP40010810-CH9-SW1.
 - [28] JSONSerialization | Apple Developer Documentation. <https://developer.apple.com/documentation/foundation/jsonserialization>.

-
- [29] LosslessStringConvertible | Apple Developer Documentation. <https://developer.apple.com/documentation/swift/losslessstringconvertible>.
- [30] Mirror | Apple Developer Documentation. <https://developer.apple.com/documentation/swift/mirror>.
- [31] Nova - Extensions. <https://docs.nova.app/extensions/>.
- [32] ObservableObject | Apple Developer Documentation. <https://developer.apple.com/documentation/combine/observableobject>.
- [33] ObservedObject | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/observedobject>.
- [34] OpenAPI Specification v3.1.0. <https://spec.openapis.org/oas/v3.1.0>.
- [35] PDF.js, A general-purpose, web standards-based platform for parsing and rendering PDFs. <https://mozilla.github.io/pdf.js/>.
- [36] Photoshop API — UXP for Adobe Photoshop. https://developer.adobe.com/photoshop/uxp/2022/ps_reference/.
- [37] Photoshop APIs for developers and scripters. <https://developer.adobe.com/photoshop/>.

-
- [38] projectedValue | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/binding/projectedvalue>.
- [39] Properties | Swift.org Documentation. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/properties/>.
- [40] RawRepresentable | Apple Developer Documentation. <https://developer.apple.com/documentation/swift/rawrepresentable>.
- [41] RedHat - What is YAML? <https://www.redhat.com/en/topics/automation/what-is-yaml>.
- [42] Salesforce Signs Definitive Agreement to Acquire Slack. <https://investor.salesforce.com/press-releases/press-release-details/2020/Salesforce-Signs-Definitive-Agreement-to-Acquire-Slack/default.aspx>. SAN FRANCISCO–(BUSINESS WIRE)– Salesforce (NYSE: CRM), the global leader in CRM, and Slack Technologies, Inc. (NYSE: WORK), the most innovative enterprise communications platform, have entered into a definitive agreement under which Salesforce will acquire Slack. Under the terms of the agreement, Slack shareholders will receive \$26.79 in cash and 0.0776 shares of Salesforce common stock for each Slack share,

representing an enterprise value of approximately \$27.7 billion based on the closing price of Salesforce's common stock on November 30, 2020.

[43] Scene | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/scene>.

[44] SceneStorage | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/scenestorage>.

[45] Stack Overflow Developer Survey 2023. <https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages>.
survey.stackoverflow.co/2023/section-most-popular-technologies-programming-scripting-and-markup-languages. 2023 continues JavaScript's streak as its eleventh year in a row as the most commonly-used programming language.

[46] State | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/state>.

[47] StateObject | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/stateobject>.

[48] swift-nio-http2: HTTP/2 support for SwiftNIO. <https://github.com/apple/swift-nio-http2>.

-
- [49] swift-nio-imap: A Swift project that provides an implementation of the IMAP4rev1 protocol, built upon SwiftNIO. <https://github.com/apple/swift-nio-imap/>.
 - [50] swift-nio-ssl: TLS Support for SwiftNIO, based on BoringSSL. <https://github.com/apple/swift-nio-ssl>.
 - [51] SwiftLog. <https://github.com/apple/swift-log>.
 - [52] SwiftNIO: Event-driven network application framework for high performance protocol servers & clients, non-blocking. <https://github.com/apple/swift-nio>.
 - [53] Swift.org - Introducing Swift OpenAPI Generator. <https://www.swift.org/blog/introducing-swift-openapi-generator/>.
 - [54] Swift.org - Swift Server Workgroup (SSWG). <https://www.swift.org/sswg/>.
 - [55] SwiftUI | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/>.
 - [56] The LLVM Compiler Infrastructure Project. <https://llvm.org/>.

-
- [57] The Swift Programming Language | Swift.org Documentation. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/>.
- [58] Thread Management - Apple Documentation Archive. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/CreatingThreads.html#//apple_ref/doc/uid/10000057i-CH15-SW7.
- [59] UNO (Universal Network Objects). https://wiki.openoffice.org/wiki/IT/Documentation/BASIC_Guide/UNO. The OpenOffice documentation is referenced here, as LibreOffice was originally forked from the former, and the UNO interface is inherited from it.
- [60] View | Apple Developer Documentation. <https://developer.apple.com/documentation/swiftui/view>.
- [61] W3C First Public Working Draft, 5 January 2023. <https://www.w3.org/TR/2023/WD-web-locks-20230105/>.
- [62] WebAssembly Specification — WebAssembly 2.0 (Draft 2023-07-24). <https://webassembly.github.io/spec/core/>.
- [63] WKWebView | Apple Developer Documentation. <https://developer.apple.com/documentation/webkit/wkwebview>.

-
- [64] XMLParser | Apple Developer Documentation. <https://developer.apple.com/documentation/foundation/xmlparser>.
- [65] YAML Ain't Markup Language (YAML™) revision 1.2.2. <https://yaml.org/spec/1.2.2/>.
- [66] Michael B. Jones, Microsoft Akshay Kumar, Microsoft Emil Lundberg, Yubico . Web Authentication: An API for accessing Public Key Credentials . <https://www.w3.org/TR/2007/REC-soap12-part0-20070427/>, November 2023.
- [67] Nilo Mitra, Ericsson Yves Lafon, W3C . SOAP Version 1.2 Part 0: Primer (Second Edition) . <https://www.w3.org/TR/2007/REC-soap12-part0-20070427/>, April 2007.
- [68] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011.
- [69] Dave Abrahams. WWDC 2015 Session 408: Protocol-Oriented Programming in Swift - Apple Developer. https://devstreaming-cdn.apple.com/videos/wwdc/2015/408509vyudbqvts/408/408_hd_protocoloriented_programming_in_swift.mp4, June 2015. Slides: https://devstreaming-cdn.apple.com/videos/wwdc/2015/408509vyudbqvts/408/408_protocoloriented_programming_in_swift.pdf.
- [70] Lisa M. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918, June 2007.

-
- [71] Matt Foster and Andrew Coven. The adobe pica api reference - version 1.1, 1997.
- [72] Jacob Xiao and Alex Migicovsky. WWDC 2016 Session 419: Protocol and Value Oriented Programming in UIKit Apps - Apple Developer. https://devstreaming-cdn.apple.com/videos/wwdc/2016/419lgbsyhjrmqtmq0qh/419/419_hd_protocol_and_value_oriented_programming_in_uikit_apps.mp4, June 2016. Slides: https://devstreaming-cdn.apple.com/videos/wwdc/2016/419lgbsyhjrmqtmq0qh/419/419_protocol_and_value_oriented_programming_in_uikit_apps.pdf.
- [73] Kyle Macomber and Arnold Schwaighofer. WWDC 2016 Session 416: Understanding Swift Performance - Apple Developer. https://devstreaming-cdn.apple.com/videos/wwdc/2016/416k7f0xkmz28rvlvwb/416/416_hd_understanding_swift_performance.mp4, June 2016. Slides: https://devstreaming-cdn.apple.com/videos/wwdc/2016/416k7f0xkmz28rvlvwb/416/416_understanding_swift_performance.pdf.
- [74] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

-
- [75] Thomas Ruark. Adobe photoshop api guide - version 6.0 release 1, 2000.