IJoL International Journal of Librarianship

ISSN: 2474-3542 Journal homepage: https://journal.calaijol.org

Modularity in FOLIO: Principles, Techniques and Tools

Michael P. Taylor

Abstract:
From its earliest inception, FOLIO was conceived not as an ILS (Integrated Library System), but as a true Services Platform, composed of many independent but interdependent modules, and forming a foundation on which an ILS or other library software could be built out of relevant modules. This vision of modularity is crucial to FOLIO's appeal to the library community, because it lowers the bar to participation: individual libraries may create modules that meet their needs, or hire developers to do so, or contribute to funding modules that will be of use to a broader community — all without needing "permission" from a central authority. The technical design of FOLIO is deeply influenced by the requirements of modularity, with the establishment of standard specifications and an emphasis on machine-readable API descriptions. While FOLIO's modular design has proved advantageous, it also introduces difficulties, including cross-module searching and data consistency. Some conventions have been established to address these difficulties, and others are in the process of crystallizing. As the ILS built on FOLIO's platform grows and matures, and as other application suites are built on it, it remains crucial to resist the shortcuts that monolithic systems can benefit from and retain the vision of modularity that has so successfully brought FOLIO this far.

To cite this article:

Taylor, M. (2021). Modularity in FOLIO: Principles, Techniques and Tools. *International Journal of Librarianship*, *6*(2), 1-12. https://doi.org/10.23974/ijol.2021.vol6.2.174

To submit your article to this journal:
Go to https://ojs.calaijol.org/index.php/ijol/about/submissions

INTERNATIONAL JOURNAL OF LIBRARIANSHIP, 6(2), 3-12.
ISSN: 2474-3542

Modularity in FOLIO: Principles, Techniques and Tools

Michael P. Taylor, Index Data and University of Bristol

ABSTRACT

From its earliest inception, FOLIO was conceived not as an ILS (Integrated Library System), but as a true Services Platform, composed of many independent but interdependent modules, and forming a foundation on which an ILS or other library software could be built out of relevant modules. This vision of modularity is crucial to FOLIO's appeal to the library community, because it lowers the bar to participation: individual libraries may create modules that meet their needs, or hire developers to do so, or contribute to funding modules that will be of use to a broader community — all without needing "permission" from a central authority. The technical design of FOLIO is deeply influenced by the requirements of modularity, with the establishment of standard specifications and an emphasis on machine-readable API descriptions. While FOLIO's modular design has proved advantageous, it also introduces difficulties, including cross-module searching and data consistency. Some conventions have been established to address these difficulties, and others are in the process of crystallizing. As the ILS built on FOLIO's platform grows and matures, and as other application suites are built on it, it remains crucial to resist the shortcuts that monolithic systems can benefit from and retain the vision of modularity that has so successfully brought FOLIO this far.

Keywords: FOLIO, Services Platform, Community, Modularity, Software Engineering.

INTRODUCTION: THE PHILOSOPHY OF FOLIO

FOLIO[1] — an acronym for "the Future Of Libraries Is Open" — is often thought of as an open-source ILS (Integrated Library System). While this is not incorrect, it's only one perspective on FOLIO's identity. The project was conceived in a crucible of idealism and pragmatism as a way of enabling the world-wide community of libraries to own and influence a modular platform for building many library-oriented applications that work together. While an ILS is one of the things that can be built on such a platform, and the one that has the most commercial visibility, it is far from the only possibility. Indeed, early in the development of FOLIO's foundations, it was half-seriously referred to as a DLS — a Disintegrated Library System — precisely because of the goal that each component would exist independently, with integration arising from interdependencies between modules based on explicitly specified interfaces.

For this reason, we make a distinction between two things: the FOLIO Platform, which is described in more detail below; and the FOLIO ILS, which is built on it. At one time, we referred

[1] https://www.folio.org/

to the former as an LSP (Library Services Platform), but that term seems to have been coined independently to mean an ILS that provides web-service APIs (Breeding, 2015), so we now avoid the term to prevent confusion.

The motivation for this modular, component-based approach runs deep in the FOLIO project. It arises from the desire to enable any interested party to contribute whatever module they are motivated to contribute, building the power and flexibility of the whole, and to avoid the situation in a typical ILS where a single gatekeeping organization carries the responsibility of making all changes and additions. The intention from day one was that a FOLIO library that wanted, for example, a module for booking rooms could create that module themselves and easily wire it into their system — and that they could make the room-booking module available for other FOLIO libraries to use. Modules can also be substituted by new implementations of existing interfaces: consider a library using a demerits system for circulation penalties rather than a fines system. Such a library could substitute the relevant modules with its own that matched the API expectations of the other platform components. The broader goal is to create a commons that is democratic not only in the organizational sense that many different stakeholders can contribute to high-level decision-making, but that anyone can actually create those parts of the system that are missing for their own specific requirements (or hire developers to do so), and contribute them back to the wider community.

Of necessity, the choice of which modules to select for early development efforts has been guided by the need to create a workable ILS on the FOLIO platform, so that modules have been created for inventory management, circulation, user management, etc. (However, see below for examples of the FOLIO platform being used for non-ILS applications.) In keeping with FOLIO's community focus, the prioritization of this early ILS development has been driven by a broad community representing libraries from many institutions, large and small, from many countries, arriving at a broad consensus. Similarly, the design of each individual core ILS module has emerged from collaborative discussions between subject-matter experts representing many and varied institutions, ensuring that the core ILS modules meet the needs of many kinds of library (FOLIO, 2021).

This community-driven, modular philosophy underlying the FOLIO project is only possible to implement because those values are supported by the design and implementation of the software architecture provided by the platform. We will now briefly survey that architecture.

THE FLEXIBLE ARCHITECTURE OF FOLIO

FOLIO is sometimes described as being made up of microservices, but this is a misleading perspective. In a microservices architecture, a defined set of heterogeneous services call each other in a web to perform tasks of very different kinds including both system logic and business logic. By contrast, in FOLIO there are an arbitrary number of modules, potentially drawn from many different sources, all of which implement aspects of an application area in a way that conforms to a set of standards. Dependencies between modules can exist, but are expressed as Web Service Application Programming Interface (WSAPI) calls between peers, and must be described in machine-readable form (see below).

There are two kinds of modules: front-end (UI) modules and back-end modules. Often an "app" consists of a UI module and an associated back-end module, the former making calls

primarily or exclusively to the latter: for example, the UI module ui-courses and the back-end module mod-courses together make up the Course Reserves app. While this example suggests a one-to-one correspondence between the front-end and back-end modules, in reality a front-end module can call many back-end modules. A back-end module can also contact other back-end modules to implement parts of its functionality.

The rules for UI and back-end modules are fundamentally different as they run in different environments. In this paper we will focus primarily on back-end modules, but before we consider these in detail, it is worth briefly considering how UI modules work.

STRIPES AND UI MODULES

Each UI module is implemented in JavaScript as an NPM package (Anonymous, 2020), using the React framework (Hunt, 2013), that meets certain requirements and can rely on certain provisions. It is run inside FOLIO's web-application framework Stripes (Taylor et al., 2017), which acts as a container for the set of UI modules in use for a specific deployment, furnishes them with facilities for access to back-end data, internationalization, etc., and renders them within a standardized layout that conforms to the FOLIO User Experience (UX) guidelines. In order to function as a Stripes module (i.e., a FOLIO UI module), an NPM package must meet the following requirements:

- Its package file must contain a stripes section.
- This section must contain an actsAs entry, an array specifying what roles this module can fulfil within Stripes: app, settings, plugin, handler, etc. Stripes uses this to know how to invoke the module.
- The stripes section must also specify what back-end interfaces, at what versions, the module uses (see below). This is a declaration of what its requirements of the back-end are.
- The package file may declare permissions that can be assigned to users of the module to enable access to aspects of its functionality. These permissions may include sub-permissions defined by back-end modules.
- A translations directory must contain files defining translation tags and their expression in at least English and usually in many other languages.
- The top-level source file must export a single React component which Stripes invokes in various different ways depending on the roles the module fulfils.

In general, UI modules do not directly depend on other UI modules (although they may indirectly depend on UI modules that have the plugin or handler role). Instead, they depend on back-end modules.

With this brief outline of UI modules in mind, from here on we will use the term "module" to mean a back-end module, and will now discuss these in more detail.

WHAT IS A FOLIO MODULE?

Just as a FOLIO UI module is a JavaScript NPM package that meets a specific set of requirements, so a FOLIO back-end module is a program that meets a different set of requirements. Unlike UI modules, back-end modules do not have to be written in any specific language: this is because, unlike UI modules, they do not run as part of a bundle, but as individual processes. FOLIO modules can be written in any language and with any tooling, and indeed modules exist written in Java, Groovy, JavaScript, Perl and other languages. At this point, the community's tooling support (see below) is strongest for Java, and so for pragmatic reasons most modules are written in that language.

All access to FOLIO back-end modules is moderated by an API gateway known as Okapi (Dickmeiss et al., 2016) — an acronym for the OK API. In typical deployments, this is enforced by blocking all network ports except the one exposed by the Okapi gateway. Among other things, Okapi manages sessions, handles user identity, maintains tenant boundaries between multiple libraries running on the same gateway, and enforces the FOLIO permissions model so that users without appropriate permissions are unable to access back-end modules: Okapi rejects their requests before the back-end modules even see them.

For a program to function as a FOLIO module, it must interact with the world by accepting and responding to WSAPI requests, and the requests that it implements must be described in machine-readable form in a module descriptor. In order to make a module available in a running instance of FOLIO, its module descriptor and some deployment information must be posted to the Okapi gateway.

The module descriptor is a JSON file (in a format defined by a JSON Schema) that describes the capabilities and limitations of the module so that Okapi can properly moderate access to it. It specifies, among other things:

- The module's unique identifier
- A list of permissions that the module defines, including which sub-permissions are included in them.
- A list of interfaces (see below) that the module requires
- A list of interfaces (see below) that the module provides, and a specification of which WSAPI operations are included in those interfaces and which permissions those operations require.

Dependencies are specified by means of interfaces rather than implementations: for example, mod-courses (which implements the back end of the Course Reserves app) depends not on the mod-inventory-storage module but on the item-storage interface. This requirement can be fulfilled by any module that implements the specified interface at a compatible version. At present this is likely to be mod-inventory-storage, but the FOLIO architecture is defined such that nothing about Course Reserves requires this, or even could. Someone else could implement an alternative inventory storage module that implements the item-storage interface, a FOLIO administrator could choose to deploy that instead of the standard implementation, and Course Reserves would happily use that instead. As Jason Skomorowski observed, "Interfaces let you replace part of the thing

without having to reinvent the whole thing so you can scratch your itch without having to start from scratch."

When a module descriptor specifies what interfaces the module provides, it includes information about what paths are supported with what HTTP methods (GET, PUT, POST, etc.), and what permissions are required by each request. Okapi uses this information to route requests to the appropriate module and to reject requests that do not have the necessary permissions. The descriptor is essentially a contract for the module.

Who can make a FOLIO module? Anyone with the technical knowhow. No-one needs to give permission. If a module meets the requirements outlined above, it can be inserted into a running FOLIO system and play its part in that ecosystem. It is a matter of policy for FOLIO operators to determine which modules they will support in their hosted FOLIO systems, but any organization hosting its own FOLIO is at liberty to add whatever modules it chooses.

Why is this important? The real value here is not in the ability of individual institutions to customize and extend their FOLIO installations. It lies in the opportunities that an open, modular architecture presents for the formation of communities and marketplaces. These communities cannot be held under the control of any individual vendor, and are at liberty to evolve the FOLIO ecosystem in whatever way best suits their needs rather than accepting whatever upgrades and extensions serve the purposes of the vendor.

TOOLING SUPPORT FOR FOLIO MODULES

In FOLIO, all communication from UI to back-end modules, and between back-end modules, is by means of HTTP-based web services. By convention, these web-services are RESTful — i.e., concerned with the state of addressable entities (Fielding, 2000) — and return responses in JSON format. These conventions are not however enforced by Okapi, which is agnostic on these issues and will happily support RPC-style web-services, XML-formatted responses, etc. (Okapi itself is controlled by RESTful JSON-based WSAPIs, which it provides for operations such as inserting or removing a module.)

The module descriptor is the only self-description that modules are required to provide: this specifies what HTTP paths and methods are supported, but says nothing about URL query parameters, the formats of requests and responses, etc. For this reason it is conventional for modules to describe the web-services in a more detailed machine-readable form. The services themselves are typically described in RAML (Heidenreich et al., 2016) or OpenAPI (Miller et al., 2020); the JSON requests and responses are described using JSON Schema (Hutton et al., 2020).

Frameworks exist to help FOLIO module developers gain maximum value from the use of these machine-readable service descriptions. The most widely used so far has been RMB (RAML Module Builder), which takes a module's RAML files and associated JSON Schemas as the source of truth, and uses them to autogenerate Java code for persisting the described objects to a PostgreSQL database. Module authors can then add business logic to the generated code. There is of course no obligation to use a framework such as RMB — for one thing, it cannot be used directly for modules written in languages other than Java — but for many modules, especially those focussed on creating, updating, fetching and searching objects of various kinds, it provides helpful support that dramatically reduces the amount of code that needs to be written.

RAML and JSON Schemas are also used by another important tool: an automatic documentation generator uses these files to create HTML developer guides at https://dev.folio.org/reference/api/. When the JSON Schemas contain useful description fields, helpful examples are provided, and introductory text is authored, the resulting autogenerated documentation can provide developers with all the information they need to work with a module: see for example https://s3.amazonaws.com/foliodocs/api/mod-courses/p/courses.html

CHALLENGES

While the modular approach taken by FOLIO has yielded many benefits, it's also true that it poses challenges. For developers coming to FOLIO from monolithic systems such as Koha, it is often frustrating that they cannot simply write a big SQL query against a single database that encompasses all the ILS's functions, and instead have to work with WSAPIs defined by the modules responsible for different areas of functionality. This is a swings-and-roundabouts situation. While the one-big-SQL-query approach is appealing for getting things done quickly, it invariably leads to maintenance difficulties in large systems that use it, with small changes to the schema rippling through code in even remotely related modules.

A more serious difficulty in working with FOLIO modules is the difficulty of cross-module queries. First consider a within-module case: a search for users whose names satisfy some criterion, in which the results returned should include the name of the patron-group that each user belongs to. The user objects returned by the low-level mod-users module include only a patron-group ID. A UI wanting to show the patron-group name must look this up in a separate patron-groups WSAPI, also provided by mod-users, to determine the name of the group with that identifier. But the business-logic module mod-users-bl presents a higher-level view of the same user data, and returns composite user objects which include an expanded patron-group object (along with other information such as the set of permissions that each user has). Because both the user register and the set of patron groups are maintained by the same underlying module, it is possible for the module to obtain information about both the user and his or her patron group in a single SQL query that joins the relevant tables.

But when a join is required across datasets that are maintained by different modules, SQL cannot be used, as this would violate the constraint that each module is responsible for its own data and its representation — indeed, in general, it cannot even be assumed that two modules use the same database or are even based on the same RDBMS. For example, consider the circulation module, mod-circulation, in which the object representing the loan of an item contains the ID of the item and the ID of the borrowing user. If we want to see the item name and user name, both of these must be looked up in their respective WSAPIs, and it is impossible for a circulation business-level module to fetch all the relevant information in a single SQL statement that joins all the tables because the circulation module knows only about its own database tables.

Of course a higher-level module can still return a loan record that includes item and user information looked up in their respective WSAPIs – and this is what mod-circulation in fact does. But there is inevitably a loss of efficiency when fetching information about a loan involves not just an SQL query, but also two calls to other modules, each of them making its own SQL query. This is a price that must be paid in order to achieve modularity. The payoff is flexibility. A new user-management module, perhaps based on LDAP or NCIP's user register facility, could be substituted

for mod-users, and the circulation module would continue to work as before with no changes required.

A more general solution to the problem of cross-module searching is provided by mod-graphql. This module has no storage of its own, but is configured by machine-readable descriptions of the links between WSAPIs offered by other modules, and knows how to respond to queries expressed in GraphQL (Facebook, 2018) by searching in multiple WSAPIs and joining up the results. This has many applications: for example, the FOLIO Z39.50 server uses mod-graphql to search for instances (bibliographic records) together with their holdings records and the items contained in those holdings, returning enriched bibliographic records that include all this information. This approach works equally for subqueries (and sub-subqueries and sub-sub-subqueries) that are implemented in the same module as the main query, or in different modules.

A related challenge is that of maintaining cross-module data integrity, given that there is no monolithic underlying RDBMS to enforce foreign-key constraints. At present, such issues are typically addressed using a publish-subscribe mechanism such as Apache Kafka to ensure that modules are notified of relevant data changes made in other modules.

Finally, it must be admitted that setting up and administering a complete FOLIO system is more challenging than running a typical monolithic ILS. This is because of the need to launch Okapi itself, load it with information about all relevant modules, and make arrangements for starting and stopping the modules themselves. This complexity, however, is the price paid in exchange for a great deal of flexibility. It allows a variety of different orchestration schemes using tools such as Kubernetes, Docker Swarm or Amazon's Elastic Container Service to allow horizontal scaling of heavily used modules, redundancy, and failover.

EXPERIENCE OF FOLIO MODULES

Given these very real challenges to FOLIO development, why persist with the modular architecture? Because the advantages of the decentralized approach reflect not just practical requirements but also the underlying philosophy of the FOLIO project. Just as libraries themselves are about democratizing access to knowledge, so modules are about democratizing library technology — about allowing its actual users to determine how and where development should be targeted, which projects they want to prioritize, what technical staff they want to employ or contract to achieve their goals, and even what interactions between modules will best serve their needs.

A small example of this is the Course Reserves module. This was required by a specific library consortium, the Fenway Library Organization, a customer of Index Data. But the FOLIO roadmap did not include a Course Reserves module, as it was not a sufficiently high priority of a large enough proportion of the subject-matter experts who feed into such decisions. In a traditional monolithic ILS, that would be the end of the story: the product vendor's development team would follow the roadmap and Course Reserves would not be added to the ILS. But in FOLIO it was a relatively simple matter for Index Data, acting alone, to design and implement a Course Reserves module (or, more precisely, a pair of modules: UI and back-end, together constituting an app). This module interacts with existing FOLIO modules' WSAPIs, most notably those provided by the Inventory module: items, locations, loan-types, etc. In doing so it requires no changes to the Inventory module, but merely consumes existing interfaces, and declares its dependence on them.

The Inventory module is unaware that Course Reserves even exists: from its perspective, Course Reserves is just a client like any other. Having proven its value, the Course Reserves module has now been accepted into the core set of FOLIO modules, making it trivial for any FOLIO-using institution to include in their installation.

A more significant example of how FOLIO's modularity opens doors unavailable to a monolithic ILS is found in the ReShare project, a system for managing Inter-Library Loan (ILL). This has recently been successfully deployed by the 50+ libraries of the Partnership for Academic Library Collaboration & Innovation (PALCI), and by the fifteen or so libraries of ConnectNY (CNY). ReShare is implemented as a set of FOLIO modules — one for managing ILL requests, one for managing fulfillment of incoming requests, one for directory management, etc. — all developed independently of the FOLIO ILS modules, but taking advantage of the FOLIO platform infrastructure. The ReShare application suite is built not only on the Stripes framework, the Okapi API gateway, and the permissions model, but also on application-level FOLIO modules include user management, tenant settings and developer settings, and tags. Additionally, a consortium deployment of ReShare includes a shared index which is implemented using the FOLIO Inventory module.

The development of the ReShare application suite has been greatly accelerated by the use of the FOLIO platform and the availability of other FOLIO modules. At the same time, the modular architecture has meant that development of ReShare has been able to advance essentially uncoupled from that of the FOLIO ILS, driven by a group of developers only some of whom are otherwise involved with FOLIO. Along the way, ReShare developers have contributed back enhancements to some parts of the FOLIO toolkit — for example, re-usable React components — indirectly helping to enrich the ILS apps as a side-effect.

We envisage the development of further application suites as well as ReShare: perhaps a content management system, an institutional repository, a procurement system, or something else altogether. Application suites may also be created to form bridges to other systems, such as ArchivesSpace or the Canvas learning management system. The exciting part of this is that the modular architecture and democratic social conventions of the FOLIO community mean that these contributions could come from anywhere — from organizations not currently linked to FOLIO, or even known to the core FOLIO developers. One measure of a software tool is the extent to which it can be put to uses that its creators did not envisage. Everything about FOLIO is calculated to encourage such uses.

CONCLUSIONS

Every aspect of the FOLIO project reflects its community origins and its democratic principles. In particular, the modular architecture is not merely a tactical choice (even though its adoption greatly contributes to eliminating what would be an exponential rise in complexity in a monolithic system). Rather, FOLIO's module system is the technical manifestation of the project's goals, attitudes and philosophy.

The choice has not been without cost. Getting an initial version of the FOLIO ILS running as a set of modules has taken some time but that cost has amply repaid itself in the ability to rapidly design, prototype and develop new additions and extensions. As a result, FOLIO is now sprouting

new functionality much faster than another ILS of comparable maturity. The current state of the project is really only the start — and where it goes from here is open for anyone to influence.

References

Anonymous. (2020). *About npm.* Retrieved September 13, 2021, from https://docs.npmjs.com/about-npm

Breeding, M. (2015). Library services platforms: A maturing genre of products. *Library Technology Reports 51*(4):1–37. https://journals.ala.org/index.php/ltr/issue/download/509/259

Dickmeiss, A., Crossley, D., Ji, H., Ladisch, J., Levanto, H., McNally, C., Skoczen, J., & Taylor, M. P. (2016). *Okapi guide and reference*. Retrieved August 24, 2021, from https://github.com/folio-org/okapi/blob/master/doc/guide.md

Facebook. (2018). *GraphQL*, June 2018 Edition. Retrieved August 27, 2021, from https://spec.graphql.org/June2018/

Fielding, R. T. (2000). Chapter 5: Representational state transfer (REST). In: *Architectural styles and the design of network-based software architectures* (Doctoral dissertation). University of California, Irvine.

FOLIO. (2021). *Our community*. Retrieved September 14, 2021, from https://www.folio.org/community/

Heidenreich, C. et al. (2016). *RAML Version 1.0: RESTful API modeling language*. Retrieved August 26, 2021, from https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/

Hunt, P. (2013). *Why did we build React?* Retrieved September 13, 2021, from https://reactjs.org/blog/2013/06/05/why-react.html

Hutton, B., Wright, A., Andrews, H., & Dennis, G. (2020). *JSON schema: A media type for describing JSON documents*, draft-bhutton-json-schema-00. Retrieved August 26, 2021, from https://json-schema.org/draft/2020-12/json-schema-core.html

Miller, D. et al. (2020). *OpenAPI specification*, version 3.1.0. 2020. Retrieved August 26, 2021, from https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md

Taylor, M. P., Burke, Z., Crossley, D., Deutsch, M. & Skomorowski, J. (2017). *The stripes module developer's guide*. Retrieved August 24, 2021, from https://github.com/folio-org/stripes/blob/master/doc/dev-guide.md

---

About the author

Michael Taylor is a Software Guy at Index Data, a boutique software house that has been creating open-source software for libraries since the early 1990s. He is also a Research Associate in dinosaur palaeontology at the University of Bristol, UK. He was one of the engineers on the core

team that laid out the technical design of FOLIO and has been deeply involved in implementing FOLIO's user-interface tools, its GraphQL service, and its Z39.50 server, as well as the ReShare application suite among other FOLIO-platform applications. In his spare time, he has described and named multiple sauropod dinosaurs, including *Xenoposeidon* ("alien earthquake god") and *Brontomerus* ("thunder-thighs"). He blogs at The Reinvigorated Programmer (https://reprog.wordpress.com/) and Sauropod Vertebra Picture of the Week (https://svpow.com/).