The Yogo Framework: An Open Source Platform For Scientific Data Management Applications

Authors and Affiliations

Ryan C. Heimbuch
Robert R. Lamb
Pol M. Llovet
Sean B. Cleveland
Ivan R. Judson

Gwen A. Jacobs (corresponding Author)
Center for Computational Biology, Montana State University, Bozeman, MT 59717

Abstract

Write This part last......

Introduction

One of the biggest challenges in scientific data sharing is the development of robust data management applications that make the data within them discoverable and interoperable. These applications should make the process of capturing the meta-data about the experimental process an easy and natural part of data management in the lab.

The burden of data management falls to the individual investigator who must find an appropriate tool. Electronic lab notebooks(ELN) and laboratory information management systems(LIMS) have been developed to fill this niche and over 30 companies offer commercial tools [Goddard 2009][Rubacha 2010][Goddard 2003][Li H Gennari 2006]. Only a few of these commercial products are appropriate for the individual investigator, due to the licensing costs, support overhead and user training costs. Many tools are expressly designed to support the discovery process in bio-tech and pharmaceutical companies, exlduing the needs of single investigators [Rubacha 2010]. Two recent additions, E-Cat [Goddard 2009] and iLabber (ilabber.com) are aimed at single investigators offering light-weight or hosted solutions for data management.

Currently no software or frameworks fulfill the requirements recently identified for successful scientific data management[Jacobs 2011-paper1]. Based on previous work and observations, we conclude the problem space across all the scientific domains is too different and complex for a "one-size-fits-all' solution. Therefore, a software framework to support the creation of flexible, domain-specific data-management application is the most appropriate solution.

This framework must encapsulate the requirements of "Eventually Perfect Data & Schema", Curation, Data Colloboration and Data Sharing. This framework needs to be open-source with a permissive license that allow commercial and open source development thus bypassing a cost-prohibitive and non-customizable hurdle. Further, this open source environment encourages the participation of a larger community of developers and scientists that can share and re-use tools as standards are established.

Eventually Perfect Data

To most researchers, the process of scientific inquiry does not work well with existing database technologies.[Jagadish 2004] Scientific discovery is not static, experimental designs change, new technologies emerge. Both the structure and content of a scientists' data changes as the investigation progresses. Most database technologies do not support an evolving data model or database schema. Thus, we introduced the concept of ***eventually perfect data***, a process by which the investigator can manage their data over time with a data management tool that allows the data model and schema to evolve without corrupting the system[Jacobs 2011].

To support the "Eventually Perfect Data" paradigm a framework must allow users to modify the schema without losing data or requiring database expertise. An evolvable schema provides the user the ability to modify how they capture the information about their experiments as they develop a new understanding of their research or identify important aspects not realized with the initial conception of the project. This iterative process, which will work with the scientific process will lead to a better, more robust, model and organization of the data. These modifications will impact the entire system, so the framework must be designed to accommodate these changes gracefully.

Curation

In order for published data to be useful, it must be curated. The required degree of curation is constantly debated topic, but it is unanimous that the more curated a published dataset is, the more useful it is[Howe 2008 Nature][Goble 2008]. On the other hand, if the curation requirements are too demanding, the curation process will inhibit data management or be ignored[Howe 2008]. A related challenge is that ontological curation of data is a discipline in itself, most experimental scientists are not ontologists (nor should they be expected to be).

To solve the curation issue a framework should support varying degrees of data annotation from relatively lightweight annotation to very comprehensive annotation. At the most basic level, a system of tagging can be implemented. This enables users to tag data elements with annotations. The next level of curation would be the usage of controlled vocabularies. These vocabularies would be used to constrain both tags and data values to managed vocabularies of terms. The most advanced level of curation would be the ability to associate data types and values with ontological terms that exist in community databases like OBI[Brinkman 2011].

A part of curation that is often overlooked is the full provenance of an end data product[Buneman 2000]. Full data provenance is a record of all changes to data that have happened since it was first detected by an instrument, observation, or sensor. In order to provide full data provenance data needs to be versioned, each version having some meta-data associated with it describing the trans-formative operation. A framework should incorporate a method to version and track provenance of data.

Collaboration & Sharing (API)

Collaboration and data sharing go hand in hand. It is obvious that if researchers are to work together they must share data and results if rapid progress is to made[Willams 2008]. And in order to collaborate with or provide data to others a data management system must include functionality that allows multiple users to access the data. However, all users are not usually given the same access to the data -- some users might be allowed to create, modify and delete data, while other users are only allowed to browse and search for data. To provide these controls all users must present credentials for authentication. Once authenticated to the system their account authorizes what operations, processes, and actions are allowed or disallowed. Most systems today implement Role-Based Access Controls (RBAC) to provide authorization functionality and a framework should provide a RBAC.

Data sharing solutions can take many forms: web based download of data, programmatic access to data using API's, and publishing data to public repositories are the most common solutions in use today[Harel 2011]. Web based download of data is easily implemented by most systems by automatically exporting specific data and putting it in a web-accessible location. The downside of this solution is having to have multiple copies of data in multiple places. A slightly better solution is to provide a read-only user interface that allows anonymous users to download the data directly from the data storage system which a framework should provide.

Providing programmatic access to data through API's is a way to allow other developers to write tools that can ingest, process and share your data. Modern web services, such as Google's map service, provide RESTful API's that allow programmers to retrieve data quickly and easily in multiple formats. The most common web data formats are JSON and XML. A framework should supply an API for easy extension and data access.

Lastly, data storage systems should be configured to publish all or some of their data to external systems.. These publication features can be used for external access to data, to backup data, or to provide data sharing through the population of external data warehouses manged by the different scientific domains.

Presented here is the initial implementation of a data management framework for building flexible scientific data management software application. The Yogo Framework (Yogo) was developed to address the needs and requirements listed above. Yogo aims to specifically address and support the use cases for "Eventually Perfect Data", Curation, Collaboration and Sharing to allow developers to rapidly create data management software that address scientists needs.

Technologies and Tools

Language and supported frameworks

Yogo's core components were developed using Ruby, a dynamic object oriented language. The Yogo framework libraries may be easily integrated into applications built with Ruby web-development tools such as Ruby-on-Rails, Sinatra and RACK.

Database/Datastore communication technology

Yogo libraries support interaction with relational database through the DataMapper libary. DataMapper is a Object/Relational Mapper (ORM) with plugable architecture that interfaces with a multitude of different datastores. Adapters exist for standard RDBMS, NoSQL stores, various file formats and even some webservices. Thus, Yogo can use any datastore that has a DataMapper adapter, the number of which is ever growing. The use of DataMapper allows the same operations to be applied to any datastore without the need to change any of the Yogo code to deal with datastore specific syntax.

Gems and Packaging

Yogo component libraries are organized within a single ruby gem named the 'yogo-framework'. Gems are the packaging and distribution technology used by all Ruby applications and libraries. The yogo-framework gem provides a meta-package wrapping multiple Yogo components, making it simpler for application developers to leverage the features contained in Yogo. The yogo-framework gem integrates each major piece of functionality: yogo-db, yogo-project, yogo-auth. It is possible to pick and choose which functionality should be included within an application if the entire framework is not required by simply including the desired gem.

Support and Documentation Libraries

Yogo was developed using many software engineering best practices such as unit testing, revision control continous integration testing and metrics and automated documentation to make use and adoption of Yogo as easy as possible. Unit testing was implemented using rspec, a ruby specification functionality testing plugin. Revision controls used were git and and the corresponding code repository is hosted at github.com/yogo/yogo. Continuous integration testing was performed using Hudson. Documentation of Yogo is automated via Yard and Yardstick was used to measure documentation coverage.

Design and Implementation

Yogo Framework Architecture Overview

Within the context of a Yogo application data is organized into collections, called Projects, which encapsulate data for access control. Project data is stored in separate databases, one per project, and where the data is versioned. The data storage component supports non-destructive schema evolution during runtime and supports multiple backend storage solutions that include file-systems and both relational and non-relational databases, including: SQLite, MySQL, PostgreSQL and Persevere (a Document DB). An RBAC component regulates user access rights to project data. All project data can be queried through the Resource Query Language (RQL)[Karvounarakis 2002] middleware component, allowing for manual and API querying across the entire application data-set.

To support the development of Yogo applications the system was designed as a set of components that can be used to build data management applications. The end result of an application developed using the Yogo components is that data managed within can be non-destructively evolved and shared with collaborators using fine-grained access controls and the availability of a rich querying API allowing programmatic access to all of the data. The genesis of such an application is made possible by the combination of the eight primary Yogo components: Yogo DB, Yogo Operation, Yogo Support, Yogo DataMapper, Yogo Project and Yogo Auth, Yogo Version and Yogo RQL (Fig 1). Many of these component can be used alone as individual gems or as an entire package, the yogo-framework gem. This modularization supports maximum flexibility for developers and enables the construction of a data management application streamlined to meet end user needs.

In practice the Yogo framework components have been combined with a Ruby web framework, such as Sinatra or Ruby on Rails. In this configuration, only the user interface needs to be customized in order to have a fully functional data management application. The Yogo applications that have been built to date have all been built in this manner. However, the framework components are also suitable for use in headless data server applications or other web services. For example, an RQL web service that processed queries against a database that was not otherwise web accessible. Or, alternatively, a searchable versioned database that spanned multiple data repositories for use with any Ruby application (or Ruby compatible runtime).

Yogo Component Decriptions

Yogo DB, Yogo Operations, Yogo Support, Yogo Datamapper and Yogo Version

The YogoDB directly uses Yogo DataMapper and Yogo Operation to implement dynamic schema and database generation[YogoDB Paper3 REF]. (Fig1).   Yogo DB works in conjunction with Yogo Operation to store sets of Yogo DataMapper operations and then uses them to produce a DataMapper model(Fig 2). Yogo Operation provides the most novel parts of Yogo by providing the constructs for writing schema transformation operations. An operation is composable with another operation when the result of an operation is of the same type as the input of the operation. This makes it possible to create more advanced operations by composing simpler operations.

Yogo DataMapper provides the system with a set of composable operations specifically for manipulating DataMapper models. A composable operation within Yogo DataMapper describes a specific modification, such as adding a field or removing a field. Composed together, they define a data model. Groups of operations can be evaluated to produce the appropriate schema for the backend storage system in use. These sets of operations are versioned, and because they are composable, operations can be combined and applied to replay all schema operations performed in the Yogo DataMapper sub-system (Fig 2). This allows rollback to any previous version of the schema. Additionally, Yogo DataMapper provides operations for relationship types including one-to-one, one-to-many, many-to-one and many-to-many allowing the capture of complex models. The generated DataMapper models contain a set of default internal properties. These properties are a unique identifier and timestamps indicating the time created and updated. These defaults are implemented so the user of the system is not required to have a unique identifier for their data. It also does not prevent the user from having their own unique identifier if they so choose.

Data manipulation is handled by Yogo DB in combination with Yogo Version. When a datum is changed, the previous version is not destroyed but stored, and the updated item becomes the primary one used by the system (Fig 3) - this is typically referred to as a “Copy-On-Write”. Optionally, the Yogo Version can require a user to annotate the update with a comment. A user tis allowed to roll-back to previous “versions” of data if an update is unwanted and provides a path of provenance from an original datum to the final end datum product. This enables end-users to implicitly retain the provenance of their data with minimal effort. Additionally, Yogo DB does not allow a traditional delete and instead marks the record as removed but still maintains it in the system so data is never lost.

Yogo Project, Yogo RQL and Yogo Auth

In Yogo a Project is a generic container for local and global data models and the data associated with them. A Project can also be thought of as an experiment, study, or collection as defined by a Yogo Application. Projects are attached to their own database, thus the generation of a new Project generates a new database. Within a Yogo Application, each Project has access to its own Models and data as well as globally defined Models and data (Fig 4).

Global models are models used by all all projects to provide shared services, such as controlled vocabularies or user models (Fig 4). The second model type is local models or models that are directly stored within a project making the data and schema only accessible to users of that particular project. Both the global and local models can be manipulated based on user roles allowing any user with proper credentials to update both the schema or the data stored within either model type.

To handle the access to data and models Yogo Auth was created. It wraps the targeted resource objects produced by the Yogo DB component with a proxy object. The proxy object looks and acts just like the original object, except that it also verifies that the current user of the system is allowed to perform the action on the object. This is known as the Facet pattern[Amborn 2004]. If the current user is not allowed to access the action, an error is raised that can be handled at a different layer of the application. For example, when an authorization error is raised, the module the error was raised in decides what action should be taken next. With the default setup, if the user has not authenticated to the system, they are prompted to login to the system and the action is attempted again. If the current user has already logged in, a <some kind of nice sounding denied error> is presented to the user, and the action is not allowed to complete.

Yogo Auth can be used to create various roles that can limit users’ abilities to create, modify, or delete schemas and data within the global or local models. In order to allow authenticated users to interact with Yogo programmatically, the framework supports security through application programmatic interface (API) keys, which users provide when they make calls to the system. These keys are unique and associated with an individual user so the Yogo-based application retrieves the user associated with the

API key and continues to process the request as if it originated from the user interface. Developers (or Users) using the RESTful APIs are limited to the same set of privileges they are authorized to perform from the web-based user interface.

The RESTful API interface exposed by Yogo is enhanced by Yogo RQL. Yogo RQL allows all HTTP-accesbile resources to respond to user defined queries. This allows developers and users to perform advanced data queries and save them as URLs. Additional benefits of the RQL interface is that it ties in nicely to current Javascript UI tools such as Dojo which aid in rapid development of user interfaces.

Discussion

Yogo Applications

The Yogo components provide powerful functionality for developers creating data management applications. Two successful data management applications have been built using the Yogo Framework for two distinctly different science domains.

The Crux Experimental Management System (Crux) was a proof of concept developed for the Michael J. Fox Foundation in collaboration with Dr. Gully Burns at the University of Southern California's Information Sciences Institute, and Dr. Alan Ruttenburg at Science Commons. The purpose of Crux was to improve the process of managing data produced by investigators in order to track the progress of funded research and outcomes of the experiments. Research funding organizations face difficult data management challenges, many that stem from the challenge of getting a comprehensive view of the all the research being supported. Crux explored making this problem more tractable by using a formalized syntax for graphically describing experimental protocols, producing a data repository from that protocol, and annotating the data directly with ontological terms to allow for data analysis of both the resulting research data and the protocols that were used to generate it. Requirements for Crux included data modeling, curation with ontological terms and import and export of data in spreadsheet format.

The Virtual Observatory and Ecological Informatics System (VOEIS) was designed in collaboration with the University of Montana, the University of Kentucky, Montana State University, the University of Louisville and Eastern Kentucky University to provide data-management infrastructure for ecologists. VOEIS provides an infrastructure that supports data acquisition, analysis, model integration, and display of data products from completed workflows including geo-spatially explicit models, graphs from statistical analyses, GIS displays of classified ecological attributes on the landscape, and 3-D visualization models of waterscape and landscape processes. At the core of the VOEIS is the VOEIS Data Hub, a Yogo application that provides coherent data management for data collected from streaming sensor networks, generated from laboratory experiments on samples taken in the field, and recorded from direct field observations. The VOEIS Data Hub required rigorous authentication and authorization requirements for granular data access, static data models, and the importing and exporting data to external data sources.

These application provided large scale application of the core Yogo components, illuminating performance and scale issues that emerge when used in a real-time, high load application. VOEIS continues to inform and direct the development of the Yogo framework.

Yogo Fulfills Data Management Requirements

The Yogo Framework was designed to meet and exceed the minimal requirements for scientific data management. As a set of functional components the framework modules provide more benefit than the sum of their individual parts, however each component is discussed in detail to illuminate the value provided to the framework and the applications produced using the framework.

Evolvable Schema and Data

The Yogo DB, Yogo Operations, Yogo DataMapper and Yogo Version modules directly support the “Evolvable Schema and Data” paradigm and satisfy one of the core requirements of a data management system for scientists -- the ability to adapt the data model quickly, painlessly, and without the possibility of losing data. These modules together allow schemas to evolve, specifically, users can modify data models without losing data or requiring database expertise. Users can add, edit, or remove schema components and because the schemas are version controlled, if a mistake is made the user can revert to the previous version of the schema. Yogo is flexible enough to allow users to store data without any schema, then incrementally improve the schema as their understanding of the data improves. This provides the lowest possible barrier to entry for users who need to adopt data management technology. Additionally, users are able to incremental change, hopefully improving, the data models they use as their conceptualization of their research changes over time.

Curation

Ideally, a Yogo application would allow the investigators to use whatever vernacular terminology that they are accustomed to, and the ontologist would link the vernacular terms to ontologically precise terms to increase the utility of the published data. CRUX a successful implementation of this curation system (Fig 5). Yogo offers a solution to the curation problem is three parts, first the evolvable schemas allow scientists to apply the eventually perfect data process leveraging the existing three curation levels.

The first level enable users to easily add initial annotations or tags in a lightweight manner that can become more rich as need arises. The second level of curation involves the use of controlled vocabularies to constrain data that is entered to a certain set of predefined values. This is implemented using additional data models in the application. The third and most rigorous form of curation is to incorporate existing community ontology's into the data by having a curator attach terms to data values and schema components. The curation process can take place subsequent to or in parallel with the activities of data gathering which prevents curation from inhibiting the experimental research process. This also allows the project to acquire the services of an ontologist later on in the process or have an ontologist on-staff that is tasked with curating the data without becoming an impediment to the research process.

Collaboration and Data Sharing

One of the key requirements for collaborative systems is to provide the ability to define access controls for individual users of the system and for unauthenticated or anonymous users. This ability to control access to the various pieces of functionality and data within a Yogo Framework application is provided by the Yogo Authentication (Yogo Auth) module. Yogo Auth provides a role based access control (RBAC) implementation that is leveraged by the rest of the Yogo Framework components. This allows different types of users to be endowed with different levels of access, disallowing anonymous or public users from being able to delete models or data for example. Yogo Auth supports multiple roles for each users within each data collection so that varying levels of data manipulation can be easily assigned such as in the VOEIS application (Fig 6) where multiple levels of access are required.

The Yogo framework enables data sharing in multiple ways, first, since the Yogo framework is implemented as a set of Rack compatible middleware components it can be used with any Ruby/Rack web framework. This makes all Yogo framework applications able to directly expose their data through a set of RESTful API's. Additionally, to expose different formats of data Yogo has the ability to parse and provide data in common formats such as XML, YAML, JSON, and CSV. Further, Yogo RQL provides a queriable interface enhancing RESTful API access that users, application, widgets or even websites can utilize to gather the data. The combination of these features provide a flexible set of data sharing solutions. For example, the VOEIS application developed using Yogo has the ability to not only publish and share data through the web-application but also to push data into the hydrological community CUAHSI HydroServer infra-structure[Hooper 2004][Tarboton 2009].

Conclusion

The use of a flexible and feature rich framework for developing data management application shows many benefits to both developers but also the researchers who are the end users. The fact that Yogo is open source with the most permissive license available will help establish an environment for the open collaboration of data-management tool creation and re-use and data-management applications that will assist the entire research community. The current successful creation of two vastly different data management application that use the Yogo Framework illustrate the current utility of the system and the possibilities for the future.

Availability and Future Directions

The currently available version of the Yogo Framework is available for download from the Yogo Project web site (http://yogo.msu.montana.edu/), source code is available from a public github repository (http://github.com/yogo/yogo). Yogo is also available through standard Ruby gem distribution mechanisms. Contributions to the framework -- bug reports, feature requests, and suggestions -- can be made through the issue interface of the github repository (http://github.com/yogo/yogo/issues).

Future development tasks include the creation of an electronic notebook based data management application, called Sapphire, the creation of an ontology curation tool, a bundled experimental protocol designer to enhance the process of designing and creating data models, a new content addressable data storage back-end allowing data to be branched and merged similar to the git source code revision control system, and integrated graphical representations of application data models and workflows.

Additional features that are driven by our applications include: finer grained authorization -- per data element permissions, reusable dashboard components, a mapping tool to map free-form tags to controlled vocabulary terms and a mapping tool to map legacy database schemas to dynamic Yogo managed schemas.

Acknowledgments

References

Amborn, M. (2004). Facet-Oriented Program Design.

Archer, D. W., Delcambre, L. M. L., & Maier, D. (2009). A framework for fine-grained data integration and curation, with provenance, in a dataspace, 8. USENIX Association.

Brinkman, R. R., Courtot, M., Derom, D., Fostel, J. M., He, Y., Lord, P., Malone, J., et al. (2010). Modeling biomedical experimental processes with OBI Journal of biomedical semantics, 1 Suppl 1, S7. doi:10.1186/2041-1480-1-S1-S7

Buneman, P., Khanna, S., & Tan, W. (2000). Lecture Notes in Computer Science. (S. Kapoor & S. Prasad, Eds.)Lecture Notes in Computer Science (Vol. 1974, pp. 87-93). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-44450-5_6

Goble, C., Stevens, R., Hull, D., Wolstencroft, K., & Lopez, R. (2008). Data curation + process curation=data integration + science. Briefings in Bioinformatics, 9(6), 506-517. doi:10.1093/bib/bbn034

Harel, A., Dalah, I., Pietrokovski, S., & Safran, M. (2011). SpringerLink - Abstract. Methods in molecular ….

Hooper, R., Maidment, D., Helly, J., & Kumar, P. (2004). CUAHSI Hydrologic Information Systems. … and Hydrology.

Howe, D., Costanzo, M., Fey, P., Gojobori, T., Hannick, L., Hide, W., Hill, D. P., et al. (2008). Big data: The future of biocuration. Nature, 455(7209), 47-50. NIH Public Access. doi:10.1038/455047a

Jagadish, H. (2004). Database management for life sciences research. ACM SIGMOD Record.

Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., & Scholl, M. (2002). Proceedings of the eleventh international conference on World Wide Web - WWW '02. In the eleventh international conference, a declarative query language for RDF (p. 592). Presented at the the eleventh international conference, New York, New York, USA: ACM Press. doi:10.1145/511446.511524

Tarboton, D., Horsburgh, J., & Maidment, D. (2009). Development of a Community Hydrologic Information System. 18th World IMACS ….

WILLIAMS, A. (2008). Internet-based tools for communication and collaboration in chemistry. Drug Discovery Today, 13(11-12), 502-506. doi:10.1016/j.drudis.2008.03.015

Figure Legends

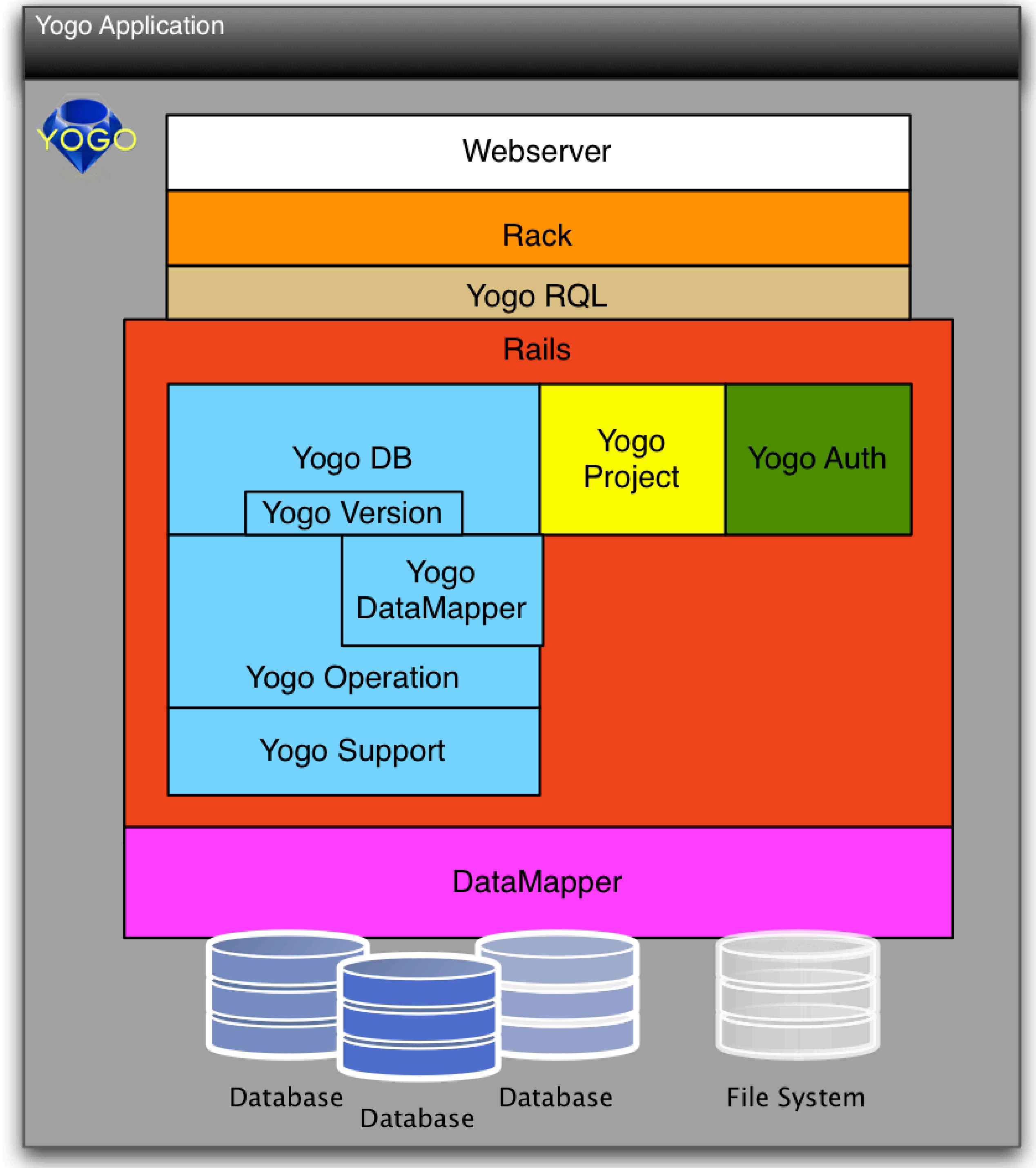


Figure 1: The Yogo Framework Implement in Rails Stack

The Yogo Framwork components are shown in the context of a Yogo application that uses a Rails software stack. The gray box indicates the Yogo Application which contains the entire software stack. The webserver is indicated by the white box and site on top of the Rack layer. The Rack layer is indicated in orange and is the layer that communicates between the Webserver and both Rails and the Yogo RQL component. The Yogo RQL component indicated in tan is an adapter that supports RQL queries passed over a REST API, supplied by Rails in this example, from the Webserver and Rack layers and translate the queries into something DataMapper can understand to retrieve the requested data. Rails,indicated in red, is the ruby MVC framework encapsulating the other Yogo componenets. Yogo DB,indicated in blue, is an implementation of a dynamic schema management tool that uses operations (provided by the Yogo Operation component also in blue) to define the core of the dynamic data definition system. With these features, Yogo enables evolvable, versioned data storage. Yogo Support, indicated in blue, provides a Ruby implementation of the required meta-programming patterns used to implement Yogo Operation. Yogo DataMapper, indicated in blue, is a set of enhancements and operations for the DataMapper object-relational-mapper. Yogo Version, indicated in blue, provides the support for versioning data and tracking data provenance. Yogo Project, indicated in yellow, is a set of flexible tools to provide dynamic project models for applications.This allows project-specific models and/or globally shared models. Yogo Auth, indicated in green, is a rich set of authorization tools that allow applications to finely tune the roles and privileges of all users in the system. DataMapper, indicated in purple, is a data-store agnostic interface between the Ruby programming language and the underlying data-stores which are indicated in shades of blue and grey.

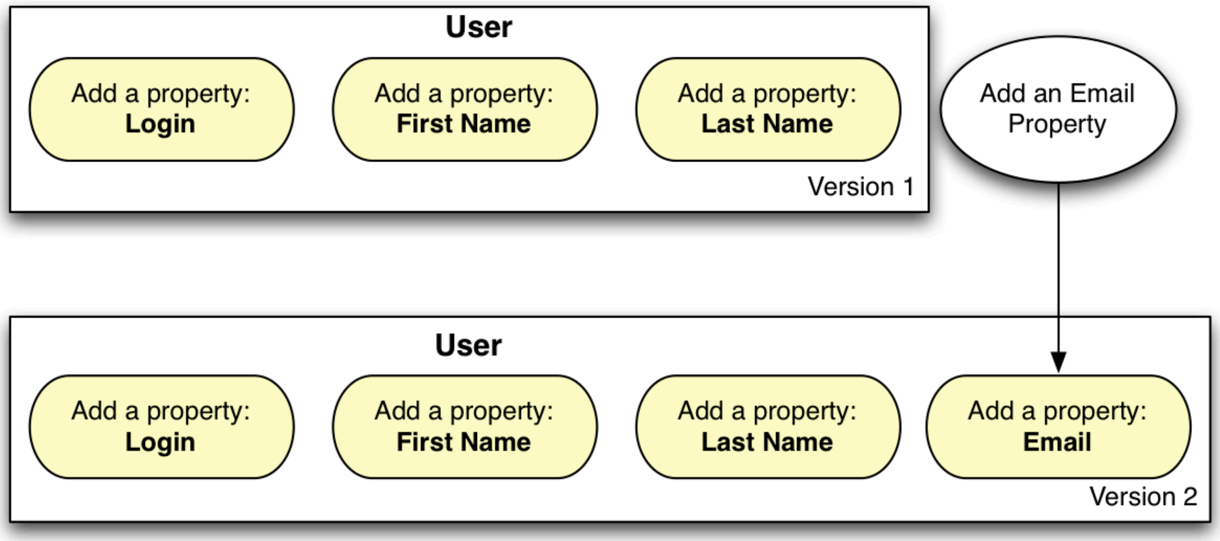
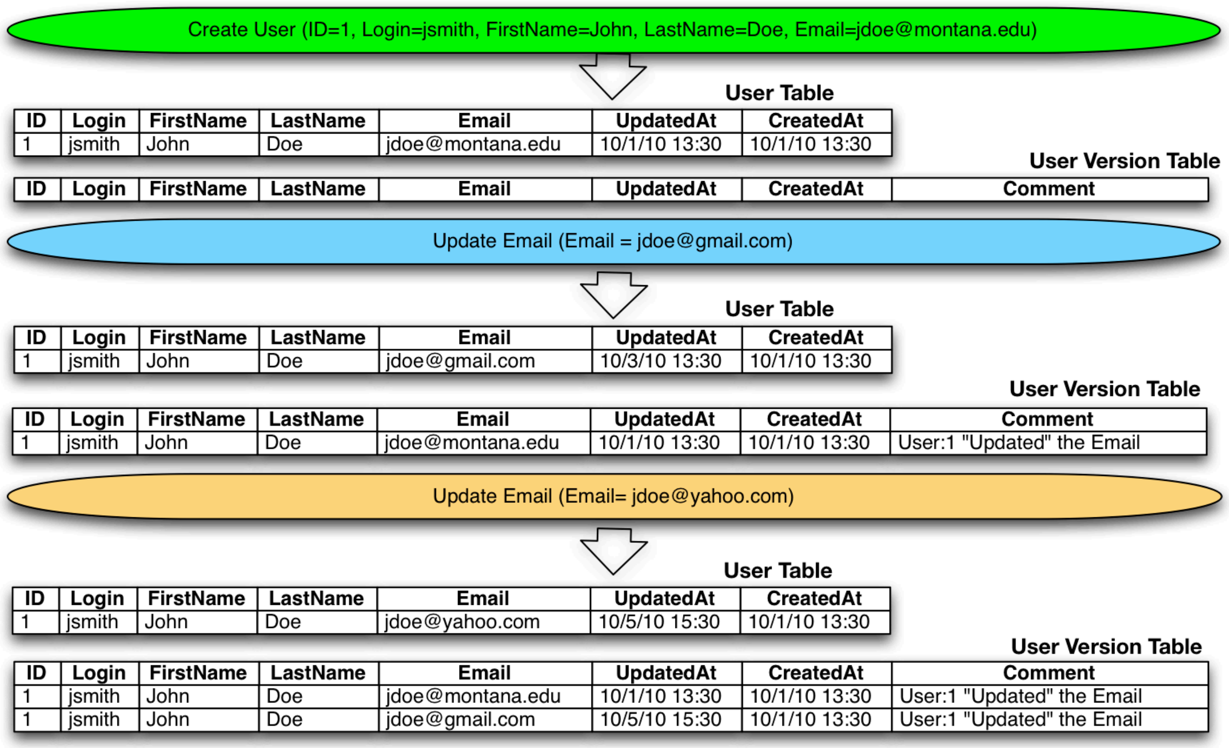


Figure 2: Composable operation: User schema evolution
This is an example of how the YogoDB would store the composable operations necessary to generate the "User" schema which would then be propogated as a table or similar object within the chosen data-storage. The white boxes represent the versioned, grouped composable operations and the yellow ovals represent individual operations and in this example all operations are "adding a property". From schema version1 to version2 the addition of the "Email" property is shown.


Create User (ID=1, Login=jsmith, FirstName=John, LastName=Doe, Email=jdoe@montana.edu)
User Table
User Version Table
Update Email (Email = jdoe@gmail.com)
User Table
User Version Table
Update Email (Email= jdoe@yahoo.com)
User Table
User Version Table

User Table

ID	Login	FirstName	LastName	Email	UpdatedAt	CreatedAt
1	jsmith	John	Doe	jdoe@montana.edu	10/1/10 13:30	10/1/10 13:30

User Version Table

ID	Login	FirstName	LastName	Email	UpdatedAt	CreatedAt	Comment

User Table

ID	Login	FirstName	LastName	Email	UpdatedAt	CreatedAt
1	jsmith	John	Doe	jdoe@gmail.com	10/3/10 13:30	10/1/10 13:30

User Version Table

ID	Login	FirstName	LastName	Email	UpdatedAt	CreatedAt	Comment
1	jsmith	John	Doe	jdoe@montana.edu	10/1/10 13:30	10/1/10 13:30	User:1 "Updated" the Email

User Table

ID	Login	FirstName	LastName	Email	UpdatedAt	CreatedAt
1	jsmith	John	Doe	jdoe@yahoo.com	10/5/10 15:30	10/1/10 13:30

User Version Table

ID	Login	FirstName	LastName	Email	UpdatedAt	CreatedAt	Comment
1	jsmith	John	Doe	jdoe@montana.edu	10/1/10 13:30	10/1/10 13:30	User:1 "Updated" the Email
1	jsmith	John	Doe	jdoe@gmail.com	10/5/10 15:30	10/1/10 13:30	User:1 "Updated" the Email

Figure 3: Versioning Figure- User Email update

As user information is updated the previous version of the user record is stored with a “user_version” table or equivalent. From the top of the diagram the initial action “Create User” of takes place, the operation is indicated in the green oval. The current user data or record that is stored within the User Table is shown where the user properties are indicated in bold and the corresponding values shown directly below them. The User Version Table is empty at this point. Upon the “Update Email” action, shown in the blue oval, where the user’s “Email” property is changed from jdoe@montana.edu to jdoe@gmail.com Yogo Version does four things. First, prior to the User table updating the user’s properties are copied to the User Version Table. Second, the User Version Table’s “Comment” for the record is populated with the User id who changed the and the action (The also supports a non-automated comment that could have more details for better provenance descriptions). Third, the User Table property “UpdatedAt” is changed to the current system date and time(10/3/10 13:30). Lastly, the “Email” property is updated to jdoe@gmail.com. At this point in the process the User Version Table has one record. Another “Update Email” action, shown in the orange oval, now occurs. All four processes occur again and the result is the updated User Table record and and two records now stored within the User Version Table.

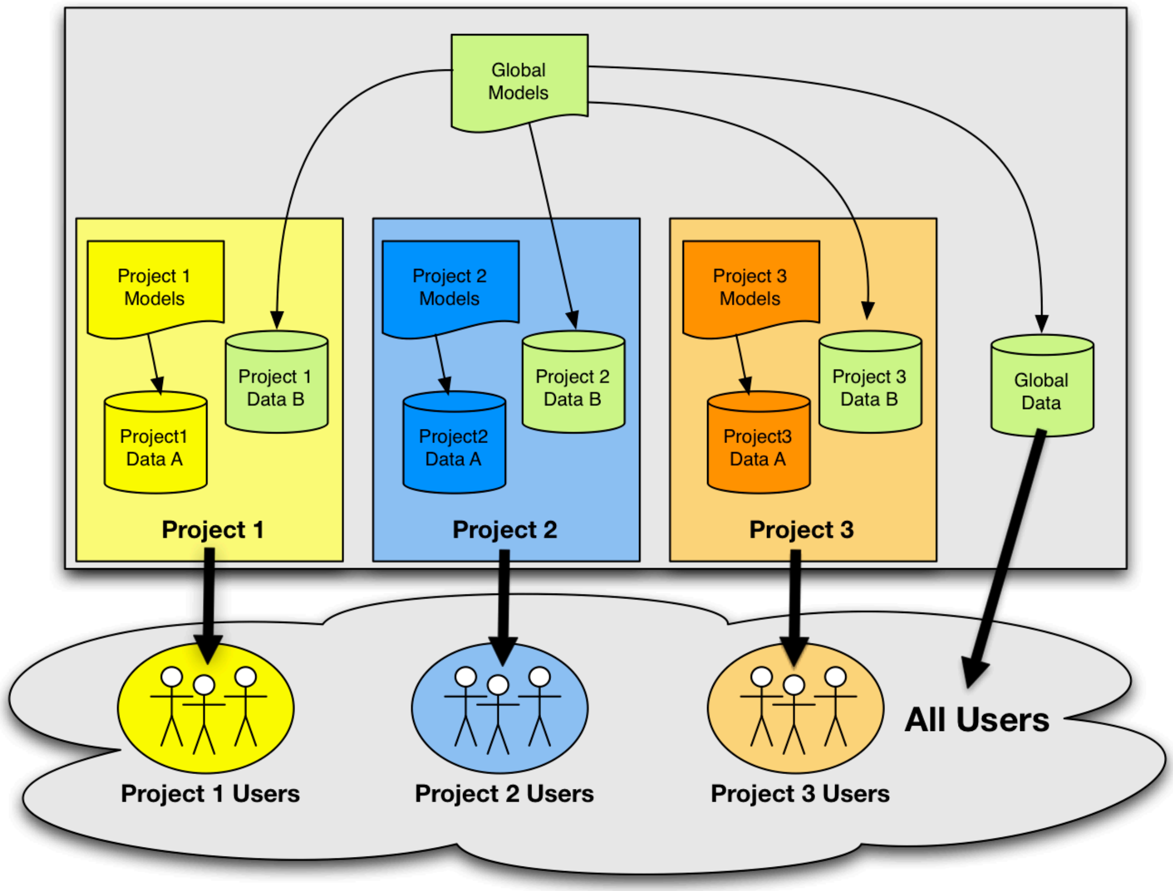
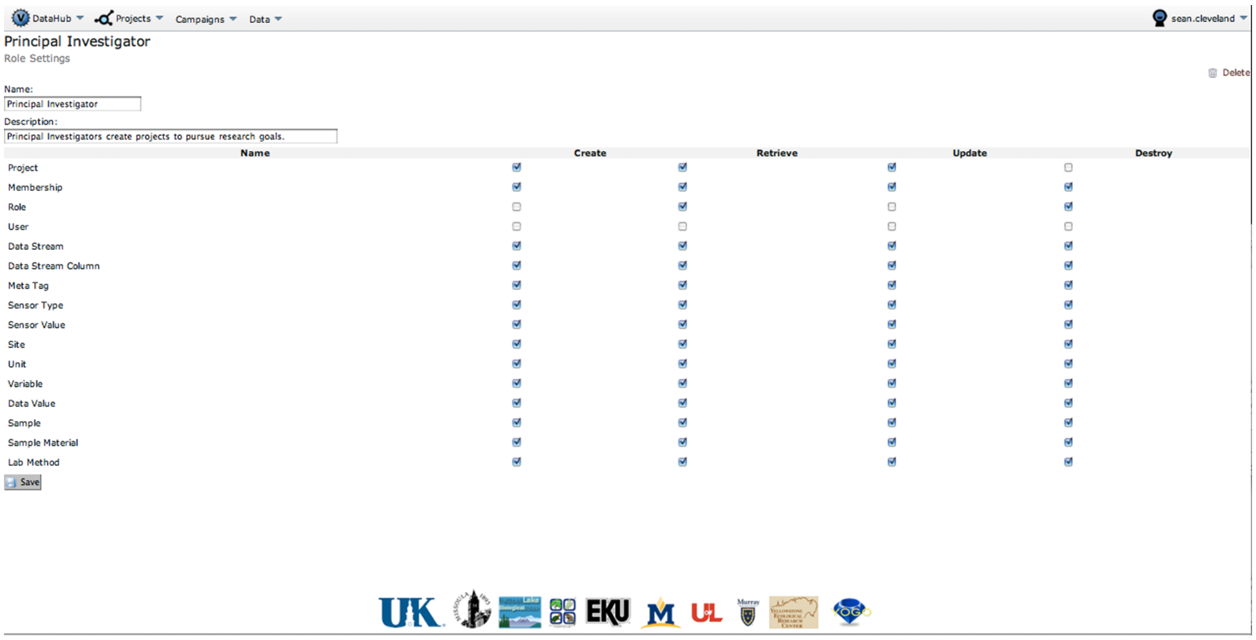


Figure 4: Yogo Project example- Global and Project Models and Data with users
Yogo applications can store both Project specific models and data and Global models and data. This application example illustrates a data management system that has three projects (Project 1 in yellow, Project 2 in blue and Project 3 in orange) and Global models, in green. This example can illustrate a use case such as a shared Controlled Vocabulary. Each project has it's own private models and data (Data A) but also has data (Data B, in green) that in the Controlled Vocabulary context would be vocabulary terms specific to that Project that used the Global model schema for organization. The Global Data is data in the same schema as the Project Data B but this data is accesbile to any user while each project's Data B is only accessible to the authorized user group indicated by the circles. This organization allows a Project to create unique terms just for the Project or re-use or share terms with the entire system.

Figure 5: CRUX curation
This is a screenshot example of the CRUX application developed using Yogo and how the Yogo componenets enabled robust data curation.



Principal Investigator

Role Settings

Name: Principal Investigator

Description: Principal Investigators create projects to pursue research goals.

Name	Create	Retrieve	Update	Destroy
Project	✓	✓	✓	☐
Membership	✓	✓	✓	✓
Role	☐	✓	☐	✓
User	☐	☐	☐	☐
Data Stream	✓	✓	✓	✓
Data Stream Column	✓	✓	✓	✓
Meta Tag	✓	✓	✓	✓
Sensor Type	✓	✓	✓	✓
Sensor Value	✓	✓	✓	✓
Site	✓	✓	✓	✓
Unit	✓	✓	✓	✓
Variable	✓	✓	✓	✓
Data Value	✓	✓	✓	✓
Sample	✓	✓	✓	✓
Sample Material	✓	✓	✓	✓
Lab Method	✓	✓	✓	✓

Figure 6: Role Based Access in VOEIS

This is an example of a UI implementation that uses the Yogo Auth funcationality to manage user roles. This user administration screen in VOEIS is for editing the system role principal investigator. The table shown is displaying the current tables or models that this principal investigator role has access to and the permissions. The Name column indicates the table name. The Create column indicates if the role can create new records in the corresponding table. The Retrieve column indicates if the role can read the records from the corresponding table. The Update column indicated if the role can update the records in the corresponing table. The Destroy table indicates if the role can delete records in the corresponding table (it should be noted that a delete only tags a record as removed and in actuality the data is never destroyed).