Using a Web Services Architecture with Me, Myself and I

Introduction

It seems counterintuitive to write code that provides web services that will only ever run across localhost connections on a single server. However, using an internal service-oriented architecture for our Library Course Pages (LCP) application provided an extensible application structure for one of the larger locally-developed code bases at the UW-Madison Libraries. Our LCP application delivers electronic reserves materials and custom library instruction web pages targeted at specific courses. It represents one of the core applications that the UW-Madison Libraries uses to support the teaching mission of the university.

Background

The original Library Course Page system grew out of the work of talented developers within one of the UW-Madison branch libraries. After the application proved to be highly successful, and when the primary developer moved into an IT management position within the Law School, there was a push to move the application support to our central library IT department. While preparing to move application support and evaluating pending enhancements we decided to work on a complete overhaul of the code base. Since we knew that the business rules and application workflows were solid, we chose to focus on cleaning up the underlying application architecture.

Reviewing the LCP application code we ascertained that there were three distinct components in play:

1. File storage
2. Hooks into campus course metadata
3. User interfaces for managing and accessing content

While tackling the re-architecture, we decided to take a web services approach: breaking the application into three distinct applications, each designed to solve a single piece of the entire LCP system. Where the original LCP system ran on a monolithic PHP code base our re-designed system runs on a combination of a Fedora-based repository, a custom XML service for course metadata using the Restlet Java framework, and a traditional user interface web application written to use the Ruby on Rails framework. Using three applications to work together to do a single job provided several advantages:

- Process isolation - each component is able to excel at one job and one job only
- Technology choice optimization - each component uses the technology best suited to its needs
- Data model-based design - the data models for each component are distinct and therefore clean

Dividing Up the Application

The choices that we made in picking the separate services were based on the kinds of data being managed. The primary questions we posed to different data elements included: how long will it last? what exactly does it do? where does it originate? and who or what uses it? Knowing how the application should function due to the success and maturity of the original system, we used these questions to determine the three categories of data in the Library Course Page application:

1. Files that needed to be stored and retrieved by patrons (primarily for reserves);
2. Course information so we would know which reserves and instruction web pages corresponded to which courses
3. The structure and content to build human readable web pages

Storing Files

The original LCP system stored files on disk using PHP and Apache to manage the file storage and access. With our rewrite we wanted to use something slightly more sophisticated to make the management of disk resources easier. File storage is part of the core functionality of most computer systems and we were not interested in reinventing wheels for this component of our system. Therefore we looked for an existing solution to storing files. The LCP application stores files that are used for library reserves or library instruction sections. These will typically include reserves readings or the kinds of handouts library staff create when they teach bibliographic instruction sessions.

The requirements for the system started with a well-defined API. Additionally, since sharing data and making it more widely available than originally intended is a growing trend in IT, and at the core of the web services philosophy, we wanted to leave open the possibility that the instructional digital learning objects in the system could be retrieved in ways unknown to us at the present time. Our instruction librarians contribute learning resources to the curriculum at our university. They have created various digital learning objects, which may be as simple as HTML code snippets that can be re-used in different contexts. Some of the more complex resources combine multiple simple objects (images, videos, help and assistance web pages) into tutorials that are integrated into campus curriculum. As instruction staff at the library create course pages we wanted to make sure that these digital learning objecs, large and small, could be used within the LCP application and potentially in other places on our library website or other campus websites.

We chose to provide a stable store for files and to maintain a high degree of control over their use via a Fedora repository. Fedora has a well documented API and a stable community of developers. For this project we have a separate installation of Fedora that is not used by non-LCP applications. The data modeling and business rules governing persistence and versioning of this data are dramatically different from those that govern our digital collections or institutional repository. Consequently, we chose to keep this Fedora instance distinct from other uses of Fedora at the UW-Madison Libraries.

At the present time, this use of the repository is simple: librarians put things in and patrons get them out. In our application setup, Fedora only listens to localhost connections for requests to create new objects or retrieve files (specifically, datastreams in Fedora lingo). Those requests come from an interface component described in the section "Data for Presentation & Interaction" below. The interface component coordinates authorization for adding to the repository and determines who has permission to retrieve items from it.

Course Info

The second major kind of data used by our application is campus course information. By working with our university’s central IT department we can automatically deliver individual library course pages to students enrolled in the corresponding courses through our campus portal. There are three high-level stages that make this happen:

1. First we pull a local copy of course metadata from our university's central IT department
2. Next the we expose the metadata pulled from step 1 to the application that creates library course pages
3. And finally we provide a metadata feed back to our campus portal about our library course pages

Delivering our data feed back to the central campus portal is important because it is already serving essential course related content to students. The libraries can work easily with campus systems because we all share common metadata elements that originate within the campus timetable and enrollment system.

It is key to understand that the libraries do not control the data elements, their organization, or semantics. These are determined by our local campus registrar. We use remote ODBC database connections to pull a feed of the relevant course metadata from the central IT department, and cache a local copy of timetable course offerings for fast use by LCP processes. We then make this data available via a web service to the application used by librarians to build a course page. The aim of this course metadata service is to make information available to people working on library reserves or course-focused instruction at the point of need. We rebuild our cache daily due to the volatile nature of our university's course offerings; we maintain local data only for the current semester and the next semester.

Because of this highly localized data model, we chose to write our own custom data service for this piece of the LCP application. We wrote code for this component that pulls a fresh feed from campus IT servers daily, stores the information in a local database optimized for point of need exposure in the form of XML, and which responds to requests using a simple URL structure based on the campus metadata model. What we identified as particular to this component is the manner in which we wanted to expose it: by translating the elements that the UW-Madison campus uses to identify courses into resources that can be accessed via RESTful URL structures.

We chose to use the Restlet framework to write this part of the application. Restlet is a lightweight Java-based framework for writing Web applications according to the REpresentational State Transfer (REST)-style architecture. It excels at mapping a URL path into parameters that can then be easily accessed by application code. For example, in order to identify Lecture 1 of the Computer Science course Introduction to Programming, offered during the Spring 2009 semester, our campus uses 5 pieces of metadata:

- the term (semester) code,
- the session code, e.g., the standard 15 week session for spring and fall semesters,
- the campus subject (department) code,
- the course number,
- and the section number.

We access the local cache of the XML metadata for this course by visiting the following URL pattern:

http://localhost/terms/<TERM_CODE>/<SESSION_CODE>/<SUBJECT_CODE>/<COURSE>/<SECTION>

The URL for the Comp Sci course in question is:

http://localhost/terms/1094/A1/266/302/001

The resource returned is an XML view with relevant data from our course timetable:

```
<section>
  <type>LEC</type>
  <number>001</number>
  <timeOfDay>01:00 PM 02:15 PM</timeOfDay>
  <daysOfWeek>TR</daysOfWeek>
  <course>
    <number>302</number>
    <title>Introduction to Programming</title>
    <department>
      <code>266</code>
      <name>COMPUTER SCIENCES</name>
    </department>
    <session>
      <code>A1</code>
      <term>
        <code>1094</code>
        <semester>Spring</semester>
        <year>2009</year>
      </term>
    </session>
  </course>
  <instructors>
    <instructor>Skrentny,James Douglas</instructor>
  </instructors>
</section>
```

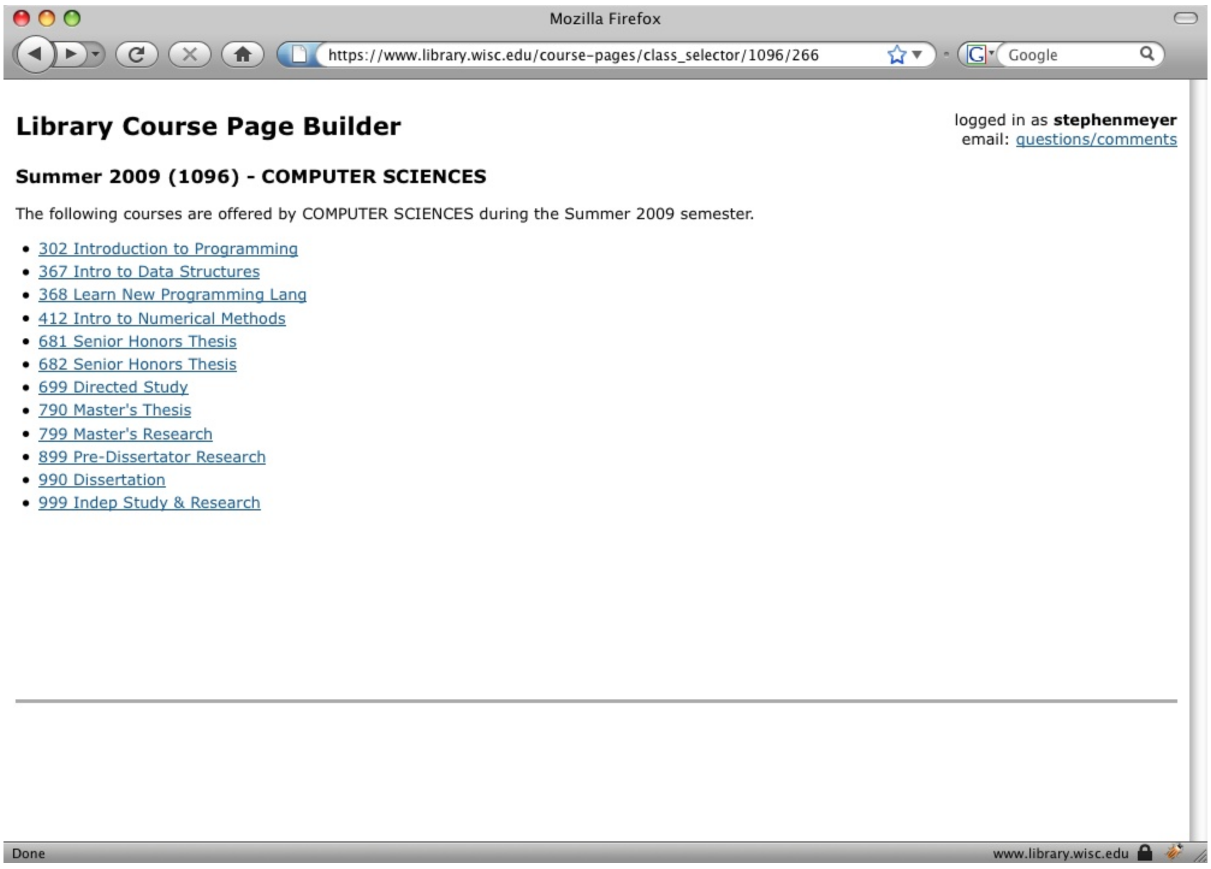
Other useful URL-defined resources in our application include things like a list of courses offered by the Computer Sciences department:

    http://localhost/terms/1094/A1/266

or a list of departments offering courses during a given session:

    http://localhost/terms/1094/A1

These kinds of resources can be used to generate navigation screens when a librarian is selecting a course to create a new library course page (see figure 1). The Restlet framework excels at mapping URL-identified resources into programmatic objects. By splitting our LCP system into three distinct applications we were free to use this technology to solve one problem extremely well.


Mozilla Firefox

https://www.library.wisc.edu/course-pages/class_selector/1096/266

Library Course Page Builder

logged in as **stephenmeyer**
email: questions/comments

Summer 2009 (1096) - COMPUTER SCIENCES

The following courses are offered by COMPUTER SCIENCES during the Summer 2009 semester.

- 302 Introduction to Programming
- 367 Intro to Data Structures
- 368 Learn New Programming Lang
- 412 Intro to Numerical Methods
- 681 Senior Honors Thesis
- 682 Senior Honors Thesis
- 699 Directed Study
- 790 Master's Thesis
- 799 Master's Research
- 899 Pre-Dissertator Research
- 990 Dissertation
- 999 Indep Study & Research

Done www.library.wisc.edu

Figure 1. Navigation screen for attaching course metadata to a Library Course Page.

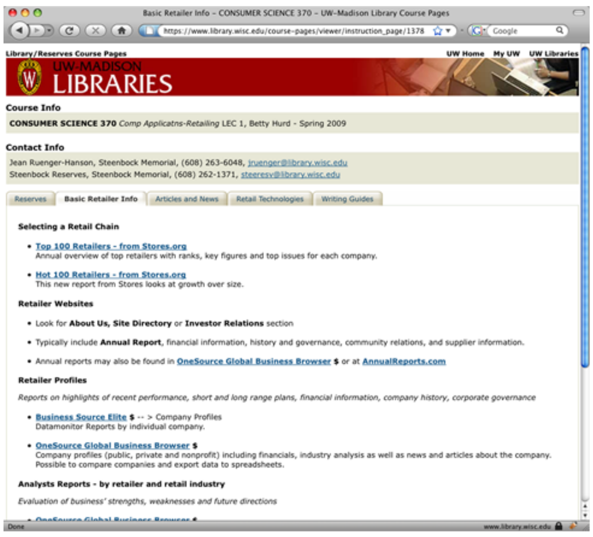
Data for Presentation & Interaction

The final component to our Library Course Page system is an interface to all the other data. This piece provides both machine and human interaction among the different components. For this part we needed a technology that is strong at both human interfaces and machine interfaces. We chose the Ruby on Rails framework because it has an MVC framework designed to solve both of these tasks.

The Human Interface

Our LCP system has a traditional web application component, a user interface for librarians to create course pages and the resulting set of web pages that students use to view the content. Librarians build course pages to deliver reserves content or custom library instruction in the form of recommended database lists and search strategies for course topics. This portion of the LCP system acts as a highly customized content management system. Common tasks for library staff building a course page might include:

- Providing the link to reserves content via a deep link into a licensed full text database,
- Uploading the course instructor's syllabus into the LCP system so that it can be downloaded from the course page,
- Providing the persistent link to a licensed electronic resource along with a description of search capabilities for the resource.


Basic Retailer Info – CONSUMER SCIENCE 370 – UW-Madison Library Course Pages

https://www.library.wisc.edu/course-pages/viewer/instruction_page/1378

Library/Reserves Course Pages   UW Home   My UW   UW Libraries

UW-MADISON LIBRARIES

Course Info

CONSUMER SCIENCE 370 *Comp Applicatns-Retailing* LEC 1, Betty Hurd - Spring 2009

Contact Info

Jean Ruenger-Hanson, Steenbock Memorial, (608) 263-6048, jruenger@library.wisc.edu
Steenbock Reserves, Steenbock Memorial, (608) 262-1371, steeresv@library.wisc.edu

Reserves | **Basic Retailer Info** | Articles and News | Retail Technologies | Writing Guides

Selecting a Retail Chain

- **Top 100 Retailers - from Stores.org**
  Annual overview of top retailers with ranks, key figures and top issues for each company.
- **Hot 100 Retailers - from Stores.org**
  This new report from Stores looks at growth over size.

Retailer Websites

- Look for **About Us, Site Directory** or **Investor Relations** section
- Typically include **Annual Report**, financial information, history and governance, community relations, and supplier information.
- Annual reports may also be found in **OneSource Global Business Browser $** or at **AnnualReports.com**

Retailer Profiles

Reports on highlights of recent performance, short and long range plans, financial information, company history, corporate governance

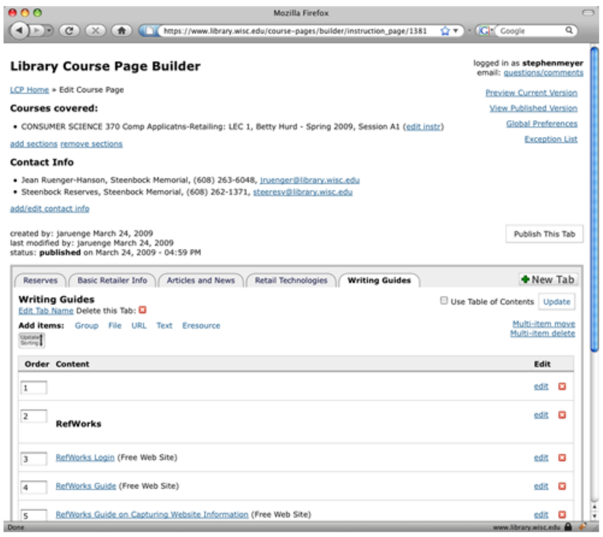
- **Business Source Elite $** -- > Company Profiles
  Datamonitor Reports by individual company.
- **OneSource Global Business Browser $**
  Company profiles (public, private and nonprofit) including financials, industry analysis as well as news and articles about the company. Possible to compare companies and export data to spreadsheets.

Analysts Reports - by retailer and retail industry

Evaluation of business' strengths, weaknesses and future directions

Done   www.library.wisc.edu

Patron View


Mozilla Firefox

https://www.library.wisc.edu/course-pages/builder/instruction_page/1381

Library Course Page Builder

logged in as **stephenmeyer**
email: questions/comments

LCP Home » Edit Course Page

Preview Current Version
View Published Version
Global Preferences
Exception List

Courses covered:

- CONSUMER SCIENCE 370 Comp Applicatns-Retailing: LEC 1, Betty Hurd - Spring 2009, Session A1 (edit instr)

add sections remove sections

Contact Info

- Jean Ruenger-Hanson, Steenbock Memorial, (608) 263-6048, jruenger@library.wisc.edu
- Steenbock Reserves, Steenbock Memorial, (608) 262-1371, steeresv@library.wisc.edu

add/edit contact info

created by: jaruenge March 24, 2009
last modified by: jaruenge March 24, 2009
status: **published** on March 24, 2009 - 04:59 PM

Publish This Tab

Reserves | Basic Retailer Info | Articles and News | Retail Technologies | **Writing Guides** | New Tab

Writing Guides
Edit Tab Name Delete this Tab: ☒

☐ Use Table of Contents Update

Add items: Group File URL Text Eresource

Multi-item move
Multi-item delete

Order	Content	Edit
1		edit ☒
2	**RefWorks**	edit ☒
3	RefWorks Login (Free Web Site)	edit ☒
4	RefWorks Guide (Free Web Site)	edit ☒
5	RefWorks Guide on Capturing Website Information (Free Web Site)	edit ☒

Done   www.library.wisc.edu

Staff View

Figure 2. End user views for patrons and staff for the LCP application

The Machine Interface

The Rails application is also the piece that coordinates the interaction among the different components. At a high level there are a few core tasks that this component performs in conjunction with the file repository and timetable service:

- Generate lists of departments and courses for a given semester,
- Store a file in the repository,
- Verify the enrollment for a given student attempting to download a file associated with a given course,
- Retrieve a file from the repository.

The machine interaction of the Ruby on Rails interface with the other components lies at the center of our web service architecture and is illustrated in figure 3.

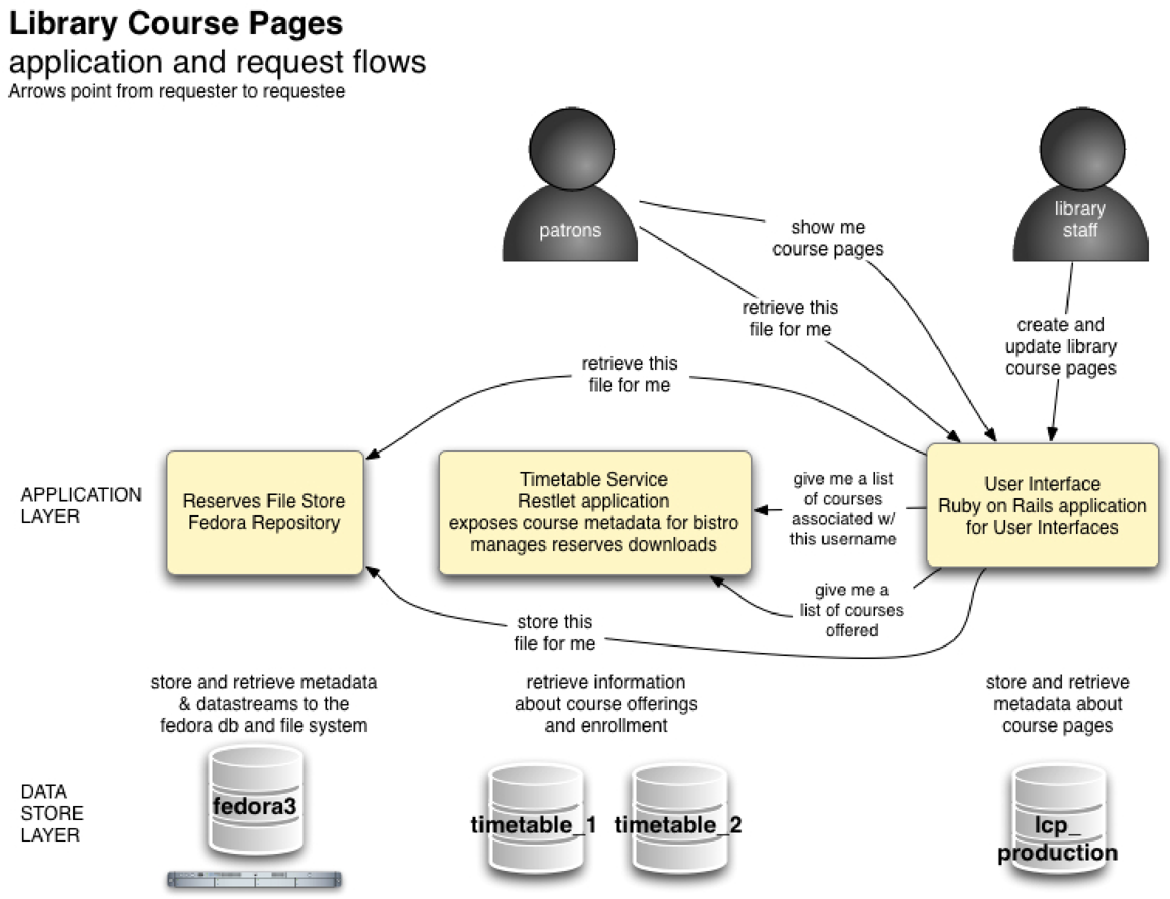

Library Course Pages
application and request flows
Arrows point from requester to requestee
patrons
library staff
show me course pages
retrieve this file for me
retrieve this file for me
create and update library course pages
APPLICATION LAYER
Reserves File Store
Fedora Repository
Timetable Service
Restlet application
exposes course metadata for bistro
manages reserves downloads
give me a list of courses associated w/ this username
User Interface
Ruby on Rails application
for User Interfaces
give me a list of courses offered
store this file for me
store and retrieve metadata & datastreams to the fedora db and file system
retrieve information about course offerings and enrollment
store and retrieve metadata about course pages
DATA STORE LAYER
fedora3
timetable_1
timetable_2
lcp_ production

Figure 3. Application request flows within the LCP system.

Pros & Cons to the Web Service Architecture

Transitioning to a web service architecture does have a few downsides in our experience. The LCP system has a slightly higher degree of complexity in that we need to manage multiple applications rather than one. Supporting the application requires a knowledge of the application components in order to quickly pinpoint where problems reside. This does make it a little more difficult for front-line support staff to troubleshoot what is going on with application problems.

Despite the up front complexity of the three for one application structure, and as mentioned above, we chose to write the new version of our application with a service-based architecture because it provides optimal technology choices. Any given application with a sufficient amount of complexity will solve many different tasks. By solving each of those functional areas individually we believe we are solving them in the most efficient manner possible.

The process of designing the locally developed application structures also benefited from the web service orientation. When designing our timetable service, for example, we approached the design with a data-centric process. Too often there is a temptation to begin designing web applications from the perspective of an end user at the expense of the data model. When we worked on the timetable service we were in the business of exposing marked up data in the form of XML. Working with the web pages that would present the data to end users was not a factor at this stage since that was left up to the design of the user interface application. The data-driven design of this service meant we were focused on ensuring our data model was well formed and optimized prior to addressing its display issues.

Something Went Very Wrong...

...but it turned out OK.

In the initial design of the LCP system the process of downloading was handled differently. Initially downloads were processed through the Restlet-based timetable service. The initial assumption was that in cases where enrollment needed to be checked for compliance with fair use copyright laws, the download process should be located as close as possible to where that enrollment data was stored. However, as the application activity picked up during the first week of classes it turned out that the initial configuration of the Restlet application was struggling under the volume and kept crashing with broken pipe exceptions.

Fortunately, Rails has native file streaming capabilities via its sendfile function. Due to the service orientation of the rest of the application, all we needed to do was rearrange the directionality of the service calls that are used to verify enrollment. To understand how this works, see the modifications to the download process illustrated in figure 4.

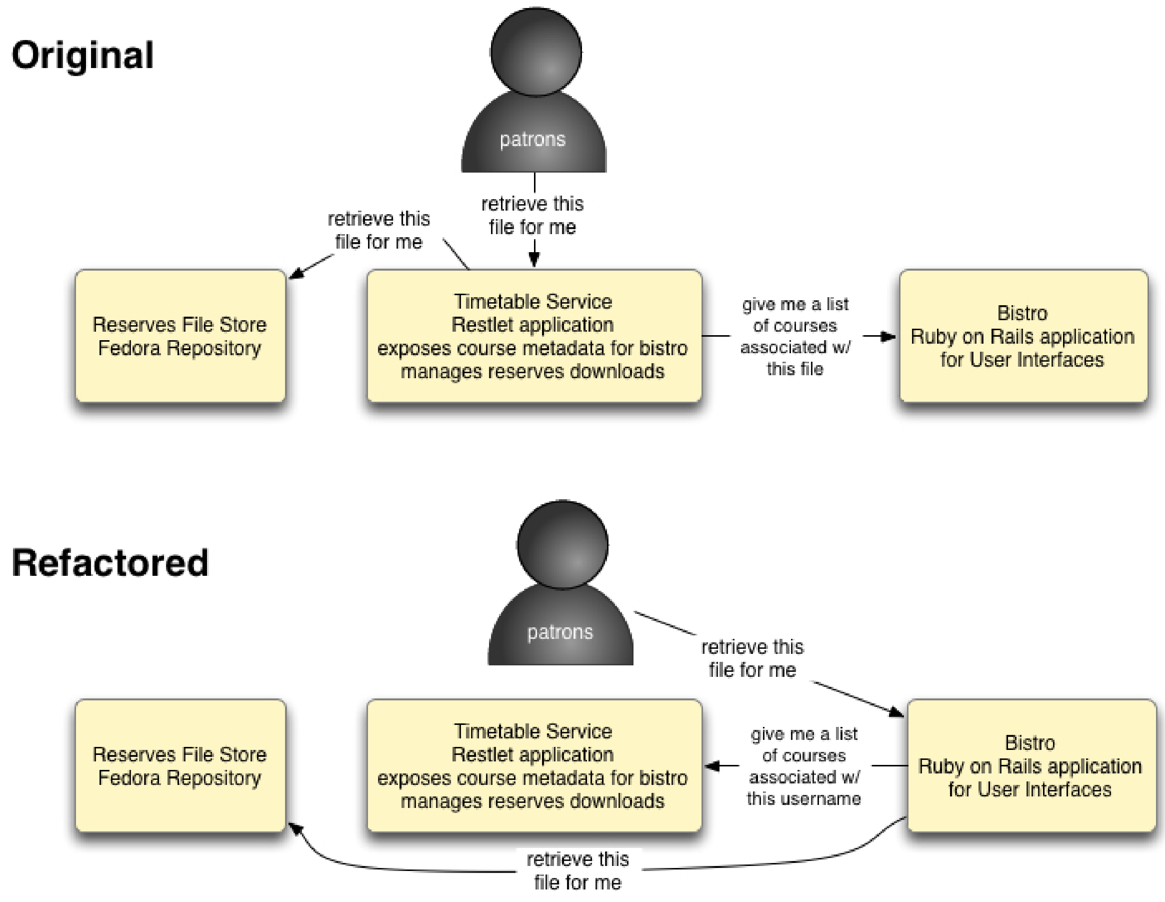

Original
patrons
retrieve this file for me
retrieve this file for me
Reserves File Store Fedora Repository
Timetable Service Restlet application exposes course metadata for bistro manages reserves downloads
give me a list of courses associated w/ this file
Bistro Ruby on Rails application for User Interfaces
Refactored
patrons
retrieve this file for me
Reserves File Store Fedora Repository
Timetable Service Restlet application exposes course metadata for bistro manages reserves downloads
give me a list of courses associated w/ this username
Bistro Ruby on Rails application for User Interfaces
retrieve this file for me

Figure 4. Application request for downloading files from the LCP system pre and post code refactoring.

Instead of the timetable service asking the Rails-based CMS which courses a particular file is associated with, the Rails application just needs to ask the timetable service which courses a given logged in student is associated with and the Rails application can gather all the data it needs to do the authorization logic. If it passes, then the Rails application can make the localhost request to Fedora for the file and send it on to the student.

Conclusion & Looking Forward

Using a web service architecture for our Library Course Page application has been a great experience for our development group. We were able to choose the best technologies to solve the different tasks at hand. We were able to design our applications based on the data models unique to each functional area. Since we have isolated each functional area within its own service the data will be easier to use in other applications. Specifically, our timetable service can be used by other applications that we would like to write, such as personalized library home pages. Because we did not couple the timetable data tightly to the LCP application exclusively, we can make API calls against it for other projects. So even though our web services only run on localhost right now, we have laid an infrastructure that can be used again and again.

For Further Exploration

- W3 Consortium definition of Web Services: http://www.w3.org/TR/ws-arch/#whatis
- Restlet framework: http://www.restlet.org/
- Ruby on Rails framework: http://rubyonrails.org/
- Fedora Repository: http://www.fedora-commons.org/

Appendx A: Timetable Service Code Example

The following code snippets demonstrate using the Restlet framework to serve up the XML needed to display a course listing for a department as illustrated in figure 1 above. These snippets are incomplete and cannot run on their own. They merely provide a demonstration of how we map a URL-based resource definition to an XML represntation of a department object.

The Base Application

The following class extends the core Restlet Application class. It is used to define individual Restlets and map them to URL patterns.

```
package edu.wisc.library.ltg.timetable.controller;

import org.restlet.Application;
import org.restlet.Restlet;
import org.restlet.Router;
import edu.wisc.library.ltg.timetable.controller.term.*;
import edu.wisc.library.ltg.timetable.model.Timetable;

public class BaseApplication extends Application {

    private Timetable timetable;

    public BaseApplication(Timetable timetable) {
        this.timetable = timetable;
    }
    @Override
    public Restlet createRoot() {

        // Create a root router
        Router router = new Router(getContext());

        /*
         * Create individual restlets for the types of responses that
         * we expect to see.
         *
         * The requests come in the form of
         *    semester:session:department:courseNumber:sectionNumber
         */
        Restlet termIndex = new TermIndex(timetable);
        Restlet term = new TermHandler(timetable);
        Restlet session = new SessionHandler(timetable);
        Restlet department = new DepartmentHandler(timetable);
        Restlet course = new CourseHandler(timetable);
        Restlet section = new SectionHandler(timetable);

        // Attach the handlers to the root router
        router.attach("/terms", termIndex);
        router.attach("/terms/{term}", term);
        router.attach("/terms/{term}/{session}", session);
        router.attach("/terms/{term}/{session}/{department}", department);
        router.attach("/terms/{term}/{session}/{department}/{course}", course);
        router.attach("/terms/{term}/{session}/{department}/{course}/{section}",
section);

        return router;
    }
}
```

The Department Handler

The following class acts as a servlet that is handed requests to represent information about academic departments. The information it returns is a list of the courses offered by the department in question for a given semester and session.

```
package edu.wisc.library.ltg.timetable.controller.term;

import org.restlet.Restlet;
import org.restlet.data.MediaType;
import org.restlet.data.Request;
import org.restlet.data.Response;
import org.restlet.data.Status;
import org.apache.log4j.Logger;
import edu.wisc.library.ltg.timetable.TimetableException;
import edu.wisc.library.ltg.timetable.model.*;

public class DepartmentHandler extends Restlet {
    protected static Logger log = Logger.getLogger(DepartmentHandler.class);
    private Department department;
    private Session session;
    private Timetable timetable;

    public DepartmentHandler(Timetable timetable) {
        // the timetable is the Hibernate-based ORM connection to the database
        this.timetable = timetable;
    }

    @Override
    public void handle(Request request, Response response) {

        try {

            // Pull the relevant params out of the URL for this handler as
            // defined by the base application.
            String termCode  = (String) request.getAttributes().get("term");
            String sessionCode = (String) request.getAttributes().get("session");
            String campusSubjectCode = (String)
request.getAttributes().get("department");

            // Get the session and department objects from the database
            session = timetable.getSession(termCode, sessionCode);
            department = timetable.getDepartment(campusSubjectCode);

            // Set the response object's entity to the XML representation of the
department
            // and apply the appropriate mime type to the response.
            response.setEntity(department.getXml(), MediaType.TEXT_XML);

        } catch (TimetableException te) {

            // Catch bad params or data from the URL and set the appropriate
response
            if( log.isInfoEnabled() ) {
                log.warn( te.getMessage() );
                log.warn( "returning response: bad client request" );
            }
            String error = "path to department is invalid";
            response.setStatus(new Status(400), error);
            response.setEntity(error, MediaType.TEXT_PLAIN);
        }
    }
}
```

As you can see, this is not a lot of code to run a servlet-style service. The Restlet website has excellent tutorials for getting started with the framework.