Administering Evergreen through the Command Line

Documentation Interest Group

Administering Evergreen through the Command Line

Documentation Interest Group

Table of Contents

I. Introduction 7
1. About This Documentation 9
2. About Evergreen 10
II. Installing Evergreen 11
3. System Requirements 14
Server Minimum Requirements 14
Web Client Requirements 14
Staff Client Requirements 14
4. Installing the Evergreen server 15
Preamble: referenced user accounts 15
Preamble: developer instructions 15
Installing prerequisites 15
Extra steps for web staff client 16
Configuration and compilation instructions 17
Installation instructions 17
Change ownership of the Evergreen files 17
Run ldconfig 18
Additional Instructions for Developers 18
Configure the Apache Web server 18
Configure OpenSRF for the Evergreen application 19
Configure action triggers for the Evergreen application 20
Creating the Evergreen database 20
Starting Evergreen 21
Testing connections to Evergreen 22
Getting help 23
License 23
5. Upgrading the Evergreen Server 24
Software Prerequisites 24
Upgrade the Evergreen code 24
Upgrade the Evergreen database schema 26
Restart Evergreen and Test 27
Review Release Notes 28
6. Setting Up EDI Acquisitions 29
Introduction 29
Installation 29
Configuration 30
Troubleshooting 31
7. Migrating from a legacy system 33
Introduction 33
Making electronic resources visible in the catalog 33
Migrating your bibliographic records 34
Migrating your call numbers, items, and parts 35
Migrating Patron Data 37
III. Individual Evergreen Components 42
8. Easing gently into OpenSRF 44
Abstract 44
Introducing OpenSRF 44
Enough jibber-jabber: writing an OpenSRF service 48
Getting under the covers with OpenSRF 60
Evergreen-specific OpenSRF services 62
Evergreen after one year: reflections on OpenSRF 64
Summary 65
Appendix: Python client 65

9. Support Scripts 67
authority_control_fields: Connecting Bibliographic and Authority records 68
marc_export: Exporting Bibliographic Records into MARC files 68
Parallel Ingest with pingest.pl 70
Importing Authority Records from Command Line 71
Juvenile-to-adult batch script 72
MARC Stream Importer 72
Processing Action Triggers 73
10. Daemons and services 75
Starting and Stopping the Reporter Daemon 75
ebook_api service 76
hold-targeter service 76
QStore service 76
11. Developing with pgTAP tests 77
Setting up pgTAP on your development server 77
Running pgTAP tests 77
IV. System Configuration 78
12. Describing your people 80
Setting the staff user's working location 80
Comparing approaches for managing permissions 81
Managing permissions in the staff client 81
Managing role-based permission groups in the staff client 83
Managing role-based permission groups in the database 86
Authentication Proxy 87
Patron Address City/State/County Pre-Populate by ZIP Code 89
Apache Rewrite Tricks 92
Apache Access Handler Perl Module 94
13. Updating translations using Launchpad 97
Prerequisites 97
Updating the translations 97
V. Cataloging Administration 98
14. Cataloging Staff Interface 100
Administering the Physical Characteristics Wizard 100
15. Cataloging timesavers and shortcuts 101
MARC Templates 101
16. Notes about the Bibliographic Schema in the Database 103
Bibliographic fingerprint 103
VI. Managing Staff from the Command Line 104
17. Changing passwords 106
VII. Patron Data 107
18. Aging Circulations 109
Global Flags 109
What Data is Aged? 109
How Circulations are Aged 110
Impacts on Billing Data 110
19. Purging holds 111
20. Purge User Activity 112
VIII. Backing up your Evergreen System 113
21. Database backups 115
Creating logical database backups 115
Restoring from logical database backups 115
Creating physical database backups with support for point-in-time recovery 116
Creating a replicated database 117
IX. UX Administration 119
22. TPac Configuration and Customization 122

Template toolkit documentation ... 122
TPAC URL ... 122
Perl modules used directly by TPAC ... 122
Default templates ... 122
Apache configuration files ... 122
TPAC CSS and media files ... 123
Mapping templates to URLs ... 123
How to override templates ... 123
Changing some text in the TPAC ... 125
Troubleshooting ... 126
23. Designing your catalog ... 127
Configuring and customizing the public interface ... 127
Setting the default physical location for your library environment ... 131
Setting a default language and adding optional languages ... 131
Change Date Format in Patron Account View ... 132
Including External Content in Your Public Interface ... 133
Including Locally Hosted Content in Your Public Interface ... 137
Styling the searchbar on the homepage ... 138
24. Designing the patron search experience ... 139
Editing the formats select box options in the search interface ... 139
Adding and removing search fields in advanced search ... 139
Changing the display of facets and facet groups ... 140
Facilitating search scope changes ... 141
Sitemap generator ... 141
Troubleshooting TPAC errors ... 142
25. Ebook API integration ... 143
Ebook API service configuration ... 143
OverDrive API integration ... 143
OneClickdigital API integration ... 144
Additional configuration ... 145
26. Managing audio alerts ... 146
Globally silencing sounds ... 146
Self-check interface ... 146
X. Creating a New Skin: the Bare Minimum ... 147
27. Introduction ... 149
28. Apache directives ... 150
29. Customizing templates ... 151
XI. Keeping Evergreen Current and Secure ... 153
30. Introduction ... 155
31. Upgrading the Evergreen software ... 156
32. Securing the server(s) on which your Evergreen installation runs ... 157
A. Attributions ... 158
B. Admonitions ... 160
C. Licensing ... 161
Index ... 162

List of Tables

7.1. 856 field for electronic resources: indicators and subfields 33

Part I. Introduction

Table of Contents

1. About This Documentation 9
2. About Evergreen 10

Chapter 1. About This Documentation

This guide was produced by the Evergreen Documentation Interest Group (DIG), consisting of numerous volunteers from many different organizations. The DIG has drawn together, edited, and supplemented pre-existing documentation contributed by libraries and consortia running Evergreen that were kind enough to release their documentation into the creative commons. Please see the Attributions section for a full list of authors and contributing organizations. Just like the software it describes, this guide is a work in progress, continually revised to meet the needs of its users, so if you find errors or omissions, please let us know, by contacting the DIG facilitators at docs@evergreen-ils.org.

This guide to Evergreen is designed for system administrators who can access their Evergreen server using a command line. It is organized into Parts, Chapters, and Sections addressing key aspects of the software.

Copies of this guide can be accessed in PDF and HTML formats from http://docs.evergreen-ils.org/.

Chapter 2. About Evergreen

Evergreen is an open source library automation software designed to meet the needs of the very smallest to the very largest libraries and consortia. Through its staff interface, it facilitates the management, cataloging, and circulation of library materials, and through its online public access interface it helps patrons find those materials.

The Evergreen software is freely licensed under the GNU General Public License, meaning that it is free to download, use, view, modify, and share. It has an active development and user community, as well as several companies offering migration, support, hosting, and development services.

The community's development requirements state that Evergreen must be:

- Stable, even under extreme load.
- Robust, and capable of handling a high volume of transactions and simultaneous users.
- Flexible, to accommodate the varied needs of libraries.
- Secure, to protect our patrons' privacy and data.
- User-friendly, to facilitate patron and staff use of the system.

Evergreen, which first launched in 2006 now powers over 544 libraries of every type – public, academic, special, school, and even tribal and home libraries – in over a dozen countries worldwide.

Part II. Installing Evergreen

Table of Contents

3. System Requirements 14
    Server Minimum Requirements 14
    Web Client Requirements 14
    Staff Client Requirements 14
4. Installing the Evergreen server 15
    Preamble: referenced user accounts 15
    Preamble: developer instructions 15
    Installing prerequisites 15
    Extra steps for web staff client 16
        Install dependencies for web staff client 16
        Install AngularJS files for web staff client 17
        Install Angular files for web staff client 17
    Configuration and compilation instructions 17
    Installation instructions 17
    Change ownership of the Evergreen files 17
    Run ldconfig 18
    Additional Instructions for Developers 18
    Configure the Apache Web server 18
    Configure OpenSRF for the Evergreen application 19
    Configure action triggers for the Evergreen application 20
    Creating the Evergreen database 20
        Setting up the PostgreSQL server 20
        Creating the Evergreen database and schema 21
        Loading sample data 21
        Creating the database on a remote server 21
    Starting Evergreen 21
    Testing connections to Evergreen 22
    Getting help 23
    License 23
5. Upgrading the Evergreen Server 24
    Software Prerequisites 24
    Upgrade the Evergreen code 24
    Upgrade the Evergreen database schema 26
    Restart Evergreen and Test 27
    Review Release Notes 28
6. Setting Up EDI Acquisitions 29
    Introduction 29
    Installation 29
        Install EDI Translator 29
        Install EDI Scripts 30
    Configuration 30
        Configuring Providers 30
        Configuring EDI Accounts 31
        Configuring Organizational Unit SAN code 31
    Troubleshooting 31
        PO JEDI Template Issues 31
7. Migrating from a legacy system 33
    Introduction 33
    Making electronic resources visible in the catalog 33
    Migrating your bibliographic records 34
    Migrating your call numbers, items, and parts 35
    Migrating Patron Data 37
        Introduction 37

Creating an sql Script for Importing Patrons 39
Batch Updating Patron Data 41

Chapter 3. System Requirements

Server Minimum Requirements

The following are the base requirements setting Evergreen up on a test server:

- An available desktop, server or virtual image
- 4GB RAM, or more if your server also runs a graphical desktop
- Linux Operating System (community supports Debian, Ubuntu, or Fedora)
- Ports 80 and 443 should be opened in your firewall for TCP connections to allow OPAC and staff client connections to the Evergreen server.

Web Client Requirements

The current stable release of Firefox or Chrome is required to run the web client in a browser.

Staff Client Requirements

Staff terminals connect to the central database using the Evergreen staff client, available for download from The Evergreen download page. The staff client must be installed on each staff workstation and requires at minimum:

- Windows, Mac OS X, or Linux operating system
- a reliable high speed Internet connection
- 2GB RAM
- The staff client uses the TCP protocol on ports 80 and 443 to communicate with the Evergreen server.

Barcode Scanners

Evergreen will work with virtually any barcode scanner – if it worked with your legacy system it should work on Evergreen.

Printers

Evergreen can use any printer configured for your terminal to print receipts, check-out slips, holds lists, etc. The single exception is spine label printing, which is still under development. Evergreen currently formats spine labels for output to a label roll printer. If you do not have a roll printer manual formatting may be required.

Chapter 4. Installing the Evergreen server

Preamble: referenced user accounts

In subsequent sections, we will refer to a number of different accounts, as follows:

- Linux user accounts:
  - The **user** Linux account is the account that you use to log onto the Linux system as a regular user.
  - The **root** Linux account is an account that has system administrator privileges. On Debian you can switch to this account from your **user** account by issuing the `su -` command and entering the password for the **root** account when prompted. On Ubuntu you can switch to this account from your **user** account using the `sudo su -` command and entering the password for your **user** account when prompted.
  - The **opensrf** Linux account is an account that you create when installing OpenSRF. You can switch to this account from the **root** account by issuing the `su - opensrf` command.
  - The **postgres** Linux account is created automatically when you install the PostgreSQL database server. You can switch to this account from the **root** account by issuing the `su - postgres` command.
- PostgreSQL user accounts:
  - The **evergreen** PostgreSQL account is a superuser account that you will create to connect to the PostgreSQL database server.
- Evergreen administrator account:
  - The **egadmin** Evergreen account is an administrator account for Evergreen that you will use to test connectivity and configure your Evergreen instance.

Preamble: developer instructions

Skip this section if you are using an official release tarball downloaded from http://evergreen-ils.org/egdownloads

Developers working directly with the source code from the Git repository, rather than an official release tarball, must perform one step before they can proceed with the `./configure` step.

As the **user** Linux account, issue the following command in the Evergreen source directory to generate the configure script and Makefiles:

```
autoreconf -i
```

Installing prerequisites

- **PostgreSQL**: The minimum supported version is 9.4.

- **Linux**: Evergreen has been tested on Debian Stretch (9), Debian Jessie (8), Ubuntu Bionic Beaver (18.04), and Ubuntu Xenial Xerus (16.04). If you are running an older version of these distributions, you may want to upgrade before upgrading Evergreen. For instructions on upgrading these distributions, visit the Debian or Ubuntu websites.

- **OpenSRF**: The minimum supported version of OpenSRF is 3.0.0.

Evergreen has a number of prerequisite packages that must be installed before you can successfully configure, compile, and install Evergreen.

1. Begin by installing the most recent version of OpenSRF (3.0.0 or later). You can download OpenSRF releases from http://evergreen-ils.org/opensrf-downloads/

2. Issue the following commands as the **root** Linux account to install prerequisites using the `Makefile.install` prerequisite installer, substituting `debian-stretch`,`debian-jessie`,`ubuntu-bionic`, or `ubuntu-xenial` for <osname> below:

   ```
   make -f Open-ILS/src/extras/Makefile.install <osname>
   ```

3. OPTIONAL: Developer additions

   To perform certain developer tasks from a Git source code checkout, additional packages may be required. As the **root** Linux account:

   - To install packages needed for retrieving and managing web dependencies, use the <osname>-developer Makefile.install target. Currently, this is only needed for building and installing the web staff client.

     ```
     make -f Open-ILS/src/extras/Makefile.install <osname>-developer
     ```

   - To install packages required for building Evergreen translations, use the <osname>-translator Makefile.install target.

     ```
     make -f Open-ILS/src/extras/Makefile.install <osname>-translator
     ```

   - To install packages required for building Evergreen release bundles, use the <osname>-packager Makefile.install target.

     ```
     make -f Open-ILS/src/extras/Makefile.install <osname>-packager
     ```

Extra steps for web staff client

Skip this entire section if you are using an official release tarball downloaded from http://evergreen-ils.org/downloads

Install dependencies for web staff client

You may skip this section if you have installed the optional developer additions. You will still need to do the following steps in Install files for web staff client.

1. Install the long-term stability (LTS) release of Node.js. Add the Node.js `/bin` directory to your environment variable `PATH`.

Install AngularJS files for web staff client

1. Building, Testing, Minification: The remaining steps all take place within the staff JS web root:

   ```
   cd $EVERGREEN_ROOT/Open-ILS/web/js/ui/default/staff/
   ```

2. Install Project-local Dependencies. npm inspects the *package.json* file for dependencies and fetches them from the Node package network.

   ```
   npm install   # fetch JS dependencies
   ```

3. Run the build script.

   ```
   # build, run tests, concat+minify
   npm run build-prod
   npm run test
   ```

Install Angular files for web staff client

1. Building, Testing, Minification: The remaining steps all take place within the Angular staff root:

   ```
   cd $EVERGREEN_ROOT/Open-ILS/src/eg2/
   ```

2. Install Project-local Dependencies. npm inspects the *package.json* file for dependencies and fetches them from the Node package network.

   ```
   npm install   # fetch JS dependencies
   ```

3. Run the build script.

   ```
   # build and run tests
   ng build --prod
   npm run test
   ```

Configuration and compilation instructions

For the time being, we are still installing everything in the `/openils/` directory. From the Evergreen source directory, issue the following commands as the **user** Linux account to configure and build Evergreen:

```
PATH=/openils/bin:$PATH ./configure --prefix=/openils --sysconfdir=/openils/conf
make
```

These instructions assume that you have also installed OpenSRF under `/openils/`. If not, please adjust PATH as needed so that the Evergreen `configure` script can find `osrf_config`.

Installation instructions

1. Once you have configured and compiled Evergreen, issue the following command as the **root** Linux account to install Evergreen and copy example configuration files to `/openils/conf`.

   ```
   make install
   ```

Change ownership of the Evergreen files

All files in the `/openils/` directory and subdirectories must be owned by the `opensrf` user. Issue the following command as the **root** Linux account to change the ownership on the files:

```
chown -R opensrf:opensrf /openils
```

Run ldconfig

On Debian Stretch, run the following command as the root user:

```
ldconfig
```

Additional Instructions for Developers

Skip this section if you are using an official release tarball downloaded from http://evergreen-ils.org/egdownloads

Developers working directly with the source code from the Git repository, rather than an official release tarball, need to install the Dojo Toolkit set of JavaScript libraries. The appropriate version of Dojo is included in Evergreen release tarballs. Developers should install the Dojo 1.3.3 version of Dojo by issuing the following commands as the **opensrf** Linux account:

```
wget http://download.dojotoolkit.org/release-1.3.3/dojo-release-1.3.3.tar.gz
tar -C /openils/var/web/js -xzf dojo-release-1.3.3.tar.gz
cp -r /openils/var/web/js/dojo-release-1.3.3/* /openils/var/web/js/dojo/.
```

Configure the Apache Web server

1. Use the example configuration files to configure your Web server for the Evergreen catalog, web staff client, Web services, and administration interfaces. Issue the following commands as the **root** Linux account:

   ```
   cp Open-ILS/examples/apache_24/eg_24.conf       /etc/apache2/sites-available/eg.conf
   cp Open-ILS/examples/apache_24/eg_vhost_24.conf /etc/apache2/eg_vhost.conf
   cp Open-ILS/examples/apache_24/eg_startup       /etc/apache2/
   # Now set up SSL
   mkdir /etc/apache2/ssl
   cd /etc/apache2/ssl
   ```

2. The `openssl` command cuts a new SSL key for your Apache server. For a production server, you should purchase a signed SSL certificate, but you can just use a self-signed certificate and accept the warnings in the and browser during testing and development. Create an SSL key for the Apache server by issuing the following command as the **root** Linux account:

   ```
   openssl req -new -x509 -days 365 -nodes -out server.crt -keyout server.key
   ```

3. As the **root** Linux account, edit the `eg.conf` file that you copied into place.

   a. To enable access to the offline upload / execute interface from any workstation on any network, make the following change (and note that you **must** secure this for a production instance):

      - Replace `Require host 10.0.0.0/8` with `Require all granted`

4. Change the user for the Apache server.

   - As the **root** Linux account, edit `/etc/apache2/envvars`. Change `export APACHE_RUN_USER=www-data` to `export APACHE_RUN_USER=opensrf`.

5. As the **root** Linux account, configure Apache with KeepAlive settings appropriate for Evergreen. Higher values can improve the performance of a single client by allowing multiple requests to be sent over the same TCP connection, but increase the risk of using up all available Apache child processes and memory.

   - Edit `/etc/apache2/apache2.conf`.

     a. Change `KeepAliveTimeout` to `1`.

     b. Change `MaxKeepAliveRequests` to `100`.

6. As the **root** Linux account, configure the prefork module to start and keep enough Apache servers available to provide quick responses to clients without running out of memory. The following settings are a good starting point for a site that exposes the default Evergreen catalogue to the web:

   `/etc/apache2/mods-available/mpm_prefork.conf`.

   ```
   <IfModule mpm_prefork_module>
      StartServers             15
      MinSpareServers           5
      MaxSpareServers          15
      MaxRequestWorkers        75
      MaxConnectionsPerChild 500
   </IfModule>
   ```

7. As the **root** user, enable the mpm_prefork module:

   ```
   a2dismod mpm_event
   a2enmod mpm_prefork
   ```

8. As the **root** Linux account, enable the Evergreen site:

   ```
   a2dissite 000-default  # OPTIONAL: disable the default site (the "It Works" page)
   a2ensite eg.conf
   ```

9. As the **root** Linux account, enable Apache to write to the lock directory; this is currently necessary because Apache is running as the `opensrf` user:

   ```
   chown opensrf /var/lock/apache2
   ```

Learn more about additional Apache options in the following sections:

- Apache Rewrite Tricks
- Apache Access Handler Perl Module

Configure OpenSRF for the Evergreen application

There are a number of example OpenSRF configuration files in `/openils/conf/` that you can use as a template for your Evergreen installation. Issue the following commands as the **opensrf** Linux account:

```
cp -b /openils/conf/opensrf_core.xml.example /openils/conf/opensrf_core.xml
cp -b /openils/conf/opensrf.xml.example /openils/conf/opensrf.xml
```

When you installed OpenSRF, you created four Jabber users on two separate domains and edited the `opensrf_core.xml` file accordingly. Please refer back to the OpenSRF README and, as the **opensrf** Linux account, edit the Evergreen version of the `opensrf_core.xml` file using the same Jabber users and domains as you used while installing and testing OpenSRF.

The `-b` flag tells the `cp` command to create a backup version of the destination file. The backup version of the destination file has a tilde (~) appended to the file name, so if you have forgotten the Jabber users and domains, you can retrieve the settings from the backup version of the files.

`eg_db_config`, described in Creating the Evergreen database, sets the database connection information in `opensrf.xml` for you.

Configure action triggers for the Evergreen application

Action Triggers provide hooks for the system to perform actions when a given event occurs; for example, to generate reminder or overdue notices, the `checkout.due` hook is processed and events are triggered for potential actions if there is no checkin time.

To enable the default set of hooks, issue the following command as the **opensrf** Linux account:

```
cp -b /openils/conf/action_trigger_filters.json.example /openils/conf/action_trigger_filters.json
```

For more information about configuring and running action triggers, see Notifications / Action Triggers.

Creating the Evergreen database

Setting up the PostgreSQL server

For production use, most libraries install the PostgreSQL database server on a dedicated machine. Therefore, by default, the `Makefile.install` prerequisite installer does **not** install the PostgreSQL 9 database server that is required by every Evergreen system. You can install the packages required by Debian or Ubuntu on the machine of your choice using the following commands as the **root** Linux account:

1. Installing PostgreSQL server packages

Each OS build target provides the postgres server installation packages required for each operating system. To install Postgres server packages, use the make target *postgres-server-*<*OSTYPE*>. Choose the most appropriate command below based on your operating system.

```
make -f Open-ILS/src/extras/Makefile.install postgres-server-debian-stretch
make -f Open-ILS/src/extras/Makefile.install postgres-server-debian-jessie
make -f Open-ILS/src/extras/Makefile.install postgres-server-ubuntu-xenial
make -f Open-ILS/src/extras/Makefile.install postgres-server-ubuntu-bionic
```

For a standalone PostgreSQL server, install the following Perl modules for your distribution as the **root** Linux account:

(Debian and Ubuntu). No extra modules required for these distributions.

You need to create a PostgreSQL superuser to create and access the database. Issue the following command as the **postgres** Linux account to create a new PostgreSQL superuser named `evergreen`. When prompted, enter the new user's password:

```
createuser -s -P evergreen
```

Enabling connections to the PostgreSQL database. Your PostgreSQL database may be configured by default to prevent connections, for example, it might reject attempts to connect via TCP/IP or from other servers. To enable

TCP/IP connections from localhost, check your `pg_hba.conf` file, found in the `/etc/postgresql/` directory on Debian and Ubuntu. A simple way to enable TCP/IP connections from localhost to all databases with password authentication, which would be suitable for a test install of Evergreen on a single server, is to ensure the file contains the following entries *before* any "host … ident" entries:

```
host    all             all             ::1/128                 md5
host    all             all             127.0.0.1/32            md5
```

When you change the `pg_hba.conf` file, you will need to reload PostgreSQL to make the changes take effect. For more information on configuring connectivity to PostgreSQL, see http://www.postgresql.org/docs/devel/static/auth-pg-hba-conf.html

Creating the Evergreen database and schema

Once you have created the **evergreen** PostgreSQL account, you also need to create the database and schema, and configure your configuration files to point at the database server. Issue the following command as the **root** Linux account from inside the Evergreen source directory, replacing <user>, <password>, <hostname>, <port>, and <dbname> with the appropriate values for your PostgreSQL database (where <user> and <password> are for the **evergreen** PostgreSQL account you just created), and replace <admin-user> and <admin-pass> with the values you want for the **egadmin** Evergreen administrator account:

```
perl Open-ILS/src/support-scripts/eg_db_config --update-config \
       --service all --create-database --create-schema --create-offline \
       --user <user> --password <password> --hostname <hostname> --port <port> \
       --database <dbname> --admin-user <admin-user> --admin-pass <admin-pass>
```

This creates the database and schema and configures all of the services in your `/openils/conf/opensrf.xml` configuration file to point to that database. It also creates the configuration files required by the Evergreen `cgi-bin` administration scripts, and sets the user name and password for the **egadmin** Evergreen administrator account to your requested values.

You can get a complete set of options for `eg_db_config` by passing the `--help` parameter.

Loading sample data

If you add the `--load-all-sample` parameter to the `eg_db_config` command, a set of authority and bibliographic records, call numbers, copies, staff and regular users, and transactions will be loaded into your target database. This sample dataset is commonly referred to as the *concerto* sample data, and can be useful for testing out Evergreen functionality and for creating problem reports that developers can easily recreate with their own copy of the *concerto* sample data.

Creating the database on a remote server

In a production instance of Evergreen, your PostgreSQL server should be installed on a dedicated server.

PostgreSQL 9.4 and later

To create the database instance on a remote database server running PostgreSQL 9.4 or later, simply use the `--create-database` flag on `eg_db_config`.

Starting Evergreen

1. As the **root** Linux account, start the `memcached` and `ejabberd` services (if they aren't already running):

```
/etc/init.d/ejabberd start
/etc/init.d/memcached start
```

2. As the **opensrf** Linux account, start Evergreen. The `-l` flag in the following command is only necessary if you want to force Evergreen to treat the hostname as `localhost`; if you configured `opensrf.xml` using the real hostname of your machine as returned by `perl -ENet::Domain 'print Net::Domain::hostfqdn() . "\n";'`, you should not use the `-l` flag.

   ```
   osrf_control -l --start-all
   ```

   - If you receive the error message `bash: osrf_control: command not found`, then your environment variable `PATH` does not include the `/openils/bin` directory; this should have been set in the **opensrf** Linux account's `.bashrc` configuration file. To manually set the `PATH` variable, edit the configuration file `~/.bashrc` as the **opensrf** Linux account and add the following line:

     ```
     export PATH=$PATH:/openils/bin
     ```

3. As the **opensrf** Linux account, generate the Web files needed by the web staff client and catalogue and update the organization unit proximity (you need to do this the first time you start Evergreen, and after that each time you change the library org unit configuration.):

   ```
   autogen.sh
   ```

4. As the **root** Linux account, restart the Apache Web server:

   ```
   /etc/init.d/apache2 restart
   ```

   If the Apache Web server was running when you started the OpenSRF services, you might not be able to successfully log in to the OPAC or web staff client until the Apache Web server is restarted.

Testing connections to Evergreen

Once you have installed and started Evergreen, test your connection to Evergreen via `srfsh`. As the **opensrf** Linux account, issue the following commands to start `srfsh` and try to log onto the Evergreen server using the **egadmin** Evergreen administrator user name and password that you set using the `eg_db_config` command:

```
/openils/bin/srfsh
srfsh% login <admin-user> <admin-pass>
```

You should see a result like:

```
Received Data: "250bf1518c7527a03249858687714376"
------------------------------------
Request Completed Successfully
Request Time in seconds: 0.045286
------------------------------------

Received Data: {
   "ilsevent":0,
   "textcode":"SUCCESS",
   "desc":" ",
   "pid":21616,
   "stacktrace":"oils_auth.c:304",
   "payload":{
      "authtoken":"e5f9827cc0f93b503a1cc66bee6bdd1a",
      "authtime":420
   }

}
```

```
----------------------------------
Request Completed Successfully
Request Time in seconds: 1.336568
----------------------------------
```

If this does not work, it's time to do some troubleshooting.

- As the **opensrf** Linux account, run the `settings-tester.pl` script to see if it finds any system configuration problems. The script is found at `Open-ILS/src/support-scripts/settings-tester.pl` in the Evergreen source tree.
- Follow the steps in the troubleshooting guide.
- If you have faithfully followed the entire set of installation steps listed here, you are probably extremely close to a working system. Gather your configuration files and log files and contact the Evergreen development mailing list for assistance before making any drastic changes to your system configuration.

Getting help

Need help installing or using Evergreen? Join the mailing lists at http://evergreen-ils.org/communicate/mailing-lists/ or contact us on the Freenode IRC network on the #evergreen channel.

License

Chapter 5. Upgrading the Evergreen Server

Before upgrading, it is important to carefully plan an upgrade strategy to minimize system downtime and service interruptions. All of the steps in this chapter are to be completed from the command line.

Software Prerequisites

- **PostgreSQL**: The minimum supported version is 9.4.
- **Linux**: Evergreen 3.X.X has been tested on Debian Stretch (9.0), Debian Jessie (8.0), Ubuntu Xenial Xerus (16.04), and Ubuntu Bionic Beaver (18.04). If you are running an older version of these distributions, you may want to upgrade before upgrading Evergreen. For instructions on upgrading these distributions, visit the Debian or Ubuntu websites.
- **OpenSRF**: The minimum supported version of OpenSRF is 3.0.0.

In the following instructions, you are asked to perform certain steps as either the **root** or **opensrf** user.

- **Debian**: To become the **root** user, issue the `su` command and enter the password of the root user.
- **Ubuntu**: To become the **root** user, issue the `sudo su` command and enter the password of your current user.

To switch from the **root** user to a different user, issue the `su - [user]` command; for example, `su - opensrf`. Once you have become a non-root user, to become the **root** user again simply issue the `exit` command.

Upgrade the Evergreen code

The following steps guide you through a simplistic upgrade of a production server. You must adjust these steps to accommodate your customizations such as catalogue skins.

1. Stop Evergreen and back up your data:
   a. As **root**, stop the Apache web server.
   b. As the **opensrf** user, stop all Evergreen and OpenSRF services:
      ```
      osrf_control --localhost --stop-all
      ```
   c. Back up the /openils directory.
2. Upgrade OpenSRF. Download and install the latest version of OpenSRF from the OpenSRF download page.
3. As the **opensrf** user, download and extract Evergreen 3.X.X:
   ```
   wget https://evergreen-ils.org/downloads/Evergreen-ILS-3.X.X.tar.gz
   tar xzf Evergreen-ILS-3.X.X.tar.gz
   ```

   For the latest edition of Evergreen, check the Evergreen download page and adjust upgrading instructions accordingly.
4. As the **root** user, install the prerequisites:

```
cd /home/opensrf/Evergreen-ILS-3.X.X
```

On the next command, replace `[distribution]` with one of these values for your distribution of Debian or Ubuntu:

- `debian-stretch` for Debian Stretch (9.0) (EDI compatibility in progress)
- `debian-jessie` for Debian Jessie (8.0) (See Bug 134222 if you want to use EDI)
- `ubuntu-xenial` for Ubuntu Xenial Xerus (16.04) (EDI compatibility in progress)

```
make -f Open-ILS/src/extras/Makefile.install [distribution]
```

5. As the **opensrf** user, configure and compile Evergreen:

```
cd /home/opensrf/Evergreen-ILS-3.X.X
PATH=/openils/bin:$PATH ./configure --prefix=/openils --sysconfdir=/openils/conf
make
```

These instructions assume that you have also installed OpenSRF under /openils/. If not, please adjust PATH as needed so that the Evergreen configure script can find osrf_config.

6. As the **root** user, install Evergreen:

```
cd /home/opensrf/Evergreen-ILS-3.X.X
make install
```

Note that this version of Evergreen does not use the legacy XUL staff client by default, but if you wish to use a versioned XUL staff client, you can supply `STAFF_CLIENT_STAMP` during the `make install` step like this:

```
cd /home/opensrf/Evergreen-ILS-3.X.X
make STAFF_CLIENT_STAMP_ID=rel_3_x_x install
```

7. As the **root** user, change all files to be owned by the opensrf user and group:

```
chown -R opensrf:opensrf /openils
```

8. (Optional, only if you are using the legacy staff client) As the **opensrf** user, update the server symlink in /openils/var/web/xul/:

```
cd /openils/var/web/xul/
rm server
ln -sf rel_3_x_x/server server
```

9. As the **opensrf** user, update opensrf_core.xml and opensrf.xml by copying the new example files (/openils/conf/opensrf_core.xml.example and /openils/conf/opensrf.xml). The *-b* option creates a backup copy of the old file.

```
cp -b /openils/conf/opensrf_core.xml.example /openils/conf/opensrf_core.xml
cp -b /openils/conf/opensrf.xml.example /openils/conf/opensrf.xml
```

Copying these configuration files will remove any customizations you have made to them. Remember to redo your customizations after copying them.

10. As the **opensrf** user, update the configuration files:

```
cd /home/opensrf/Evergreen-ILS-3.X.X
perl Open-ILS/src/support-scripts/eg_db_config --update-config --service all \
--create-offline --database evergreen --host localhost --user evergreen --password evergreen
```

11.As the **root** user, update the Apache files:

Use the example configuration files in `Open-ILS/examples/apache/` (for Apache versions below 2.4) or `Open-ILS/examples/apache_24/` (for Apache versions 2.4 or greater) to configure your Web server for the Evergreen catalog, staff client, Web services, and administration interfaces. Issue the following commands as the **root** Linux account:

Copying these Apache configuration files will remove any customizations you have made to them. Remember to redo your customizations after copying them. For example, if you purchased an SSL certificate, you will need to edit eg.conf to point to the appropriate SSL certificate files. The diff command can be used to show the differences between the distribution version and your customized version. `diff <customized file> <dist file>`

a. Update */etc/apache2/eg_startup* by copying the example from *Open-ILS/examples/apache/eg_startup*.

```
cp /home/opensrf/Evergreen-ILS-3.X.X/Open-ILS/examples/apache/eg_startup /etc/apache2/eg_startup
```

b. Update /etc/apache2/eg_vhost.conf by copying the example from Open-ILS/examples/apache/eg_vhost.conf.

```
cp /home/opensrf/Evergreen-ILS-3.X.X/Open-ILS/examples/apache/eg_vhost.conf /etc/apache2/eg_vhost.conf
```

c. Update /etc/apache2/sites-available/eg.conf by copying the example from Open-ILS/examples/apache/eg.conf.

```
cp /home/opensrf/Evergreen-ILS-3.X.X/Open-ILS/examples/apache/eg.conf /etc/apache2/sites-available/eg.conf
```

Upgrade the Evergreen database schema

The upgrade of the Evergreen database schema is the lengthiest part of the upgrade process for sites with a significant amount of production data.

Before running the upgrade script against your production Evergreen database, back up your database, restore it to a test server, and run the upgrade script against the test server. This enables you to determine how long the upgrade will take and whether any local customizations present problems for the stock upgrade script that require further tailoring of the upgrade script. The backup also enables you to cleanly restore your production data if anything goes wrong during the upgrade.

Evergreen provides incremental upgrade scripts that allow you to upgrade from one minor version to the next until you have the current version of the schema. For example, if you want to upgrade from 2.9.0 to 2.11.0, you would run the following upgrade scripts:

- 2.9.0-2.9.1-upgrade-db.sql
- 2.9.1-2.9.2-upgrade-db.sql
- 2.9.2-2.9.3-upgrade-db.sql
- 2.9.3-2.10.0-upgrade-db.sql (this is a major version upgrade)
- 2.10.0-2.10.1-upgrade-db.sql

- 2.10.1-2.10.2-upgrade-db.sql
- 2.10.2-2.10.3-upgrade-db.sql
- 2.10.3-2.10.4-upgrade-db.sql
- 2.10.4-2.10.5-upgrade-db.sql
- 2.10.5-2.10.6-upgrade-db.sql
- 2.10.6-2.10.7-upgrade-db.sql
- 2.10.7-2.11.0-upgrade-db.sql (this is a major version upgrade)

Note that you do **not** necessarily want to run additional upgrade scripts to upgrade to the newest version, since currently there is no automated way, for example to upgrade from 2.9.4+ to 2.10. Only upgrade as far as necessary to reach the major version upgrade script (in this example, as far as 2.9.3).

Pay attention to error output as you run the upgrade scripts. If you encounter errors that you cannot resolve yourself through additional troubleshooting, please report the errors to the Evergreen Technical Discussion List.

Run the following steps (including other upgrade scripts, as noted above) as a user with the ability to connect to the database server.

```
cd /home/opensrf/Evergreen-ILS-3.X.X/Open-ILS/src/sql/Pg
psql -U evergreen -h localhost -f version-upgrade/3.X.W-3.X.X-upgrade-db.sql evergreen
```

After the some database upgrade scripts finish, you may see a note on how to reingest your bib records. You may run this after you have completed the entire upgrade and tested your system. Reingesting records may take a long time depending on the number of bib records in your system.

Restart Evergreen and Test

1. As the **root** user, restart memcached to clear out all old user sessions.

   ```
   service memcached restart
   ```

2. As the **opensrf** user, start all Evergreen and OpenSRF services:

   ```
   osrf_control --localhost --start-all
   ```

3. As the **opensrf** user, run autogen to refresh the static organizational data files:

   ```
   cd /openils/bin
   ./autogen.sh
   ```

4. Start srfsh and try logging in using your Evergreen username and password:

   ```
   /openils/bin/srfsh
   srfsh% login username password
   ```

   You should see a result like:

```
Received Data: "250bf1518c7527a03249858687714376"
    ------------------------------------
    Request Completed Successfully
    Request Time in seconds: 0.045286
    ------------------------------------

    Received Data: {
       "ilsevent":0,
       "textcode":"SUCCESS",
       "desc":" ",
       "pid":21616,
       "stacktrace":"oils_auth.c:304",
       "payload":{
          "authtoken":"e5f9827cc0f93b503a1cc66bee6bdd1a",
          "authtime":420
       }

    }

    ------------------------------------
    Request Completed Successfully
    Request Time in seconds: 1.336568
    ------------------------------------
```

If this does not work, it's time to do some troubleshooting.

5. As the **root** user, start the Apache web server.

   If you encounter errors, refer to the troubleshooting section of this documentation for tips on finding solutions and seeking further assistance from the Evergreen community.

Review Release Notes

Review this version's release notes for other tasks that need to be done after upgrading. If you have upgraded over several major versions, you will need to review the release notes for each version also.

Chapter 6. Setting Up EDI Acquisitions

Introduction

Electronic Data Interchange (EDI) is used to exchange information between participating vendors and Evergreen. This chapter contains technical information for installation and configuration of the components necessary to run EDI Acquisitions for Evergreen.

Installation

Install EDI Translator

The EDI Translator is used to convert data into EDI format. It runs on localhost and listens on port 9191 by default. This is controlled via the edi_webrick.cnf file located in the edi_translator directory. It should not be necessary to edit this configuration if you install EDI Translator on the same server used for running Action/Triggers events.

If you are running Evergreen with a multi-server configuration, make sure to install EDI Translator on the same server used for Action/Trigger event generation.

Steps for Installing

1. As the **opensrf** user, copy the EDI Translator code found in Open-ILS/src/edi_translator to somewhere accessible (for example, /openils/var/edi):

   ```
   cp -r Open-ILS/src/edi_translator /openils/var/edi
   ```

2. Navigate to where you have saved the code to begin next step:

   ```
   cd /openils/var/edi
   ```

3. Next, as the **root** user (or a user with sudo rights), install the dependencies, via "install.sh". This will perform some apt-get routines to install the code needed for the EDI translator to function. (Note: subversion must be installed first)

   ```
   ./install.sh
   ```

4. Now, we're ready to start "edi_webrick.bash" which is the script that calls the "Ruby" code to translate EDI. This script needs to be started in order for EDI to function so please take appropriate measures to ensure this starts following reboots/upgrades/etc. As the **opensrf** user:

   ```
   ./edi_webrick.bash
   ```

5. You can check to see if EDI translator is running.

   - Using the command "ps aux | grep edi" should show you something similar if the script is running properly:

     ```
     root    30349  0.8  0.1  52620 10824 pts/0      S       13:04   0:00 ruby ./edi_webrick.rb
     ```

   - To shutdown EDI Translator you can use something like pkill (assuming no other ruby processes are running on that server):

     ```
     kill -INT $(pgrep ruby)
     ```

Install EDI Scripts

The EDI scripts are "edi_pusher.pl" and "edi_fetcher.pl" and are used to "push" and "fetch" EDI messages for configured EDI accounts.

1. As the **opensrf** user, copy edi_pusher.pl and edi_fetcher.pl from Open-ILS/src/support-scripts into /openils/bin:

```
cp Open-ILS/src/support-scripts/edi_pusher.pl /openils/bin
cp Open-ILS/src/support-scripts/edi_fetcher.pl /openils/bin
```

2. Setup the edi_pusher.pl and edi_fetcher.pl scripts to run as cron jobs in order to regularly push and receive EDI messages.

   - Add to the opensrf user's crontab the following entries:

```
10 * * * * cd /openils/bin && /usr/bin/perl ./edi_pusher.pl > /dev/null
0 1 * * * cd /openils/bin && /usr/bin/perl ./edi_fetcher.pl > /dev/null
```

   - The example for edi_pusher.pl sets the script to run at 10 minutes past the hour, every hour.
   - The example for edi_fetcher.pl sets the script to run at 1 AM every night.

You may choose to run the EDI scripts more or less frequently based on the necessary response times from your vendors.

Configuration

Configuring Providers

Look in Administration → Acquisitions Administration → Providers

Column	Description/Notes
Provider Name	A unique name to identify the provider
Code	A unique code to identify the provider
Owner	The org unit who will "own" the provider.
Currency	The currency format the provider accepts
Active	Whether or not the Provider is "active" for use
Default Claim Policy	??
EDI Default	The default "EDI Account" to use (see EDI Accounts Configuration)
Email	The email address for the provider
Fax Phone	A fax number for the provider
Holdings Tag	The holdings tag to be utilized (usually 852, for Evergreen)
Phone	A phone number for the provider
Prepayment Required	Whether or not prepayment is required
SAN	The vendor provided, org unit specific SAN code
URL	The vendor website

Configuring EDI Accounts

Look in Administration → Acquisitions Administration → EDI Accounts

Column	Description/Notes
Label	A unique name to identify the provider
Host	FTP/SFTP/SSH hostname - vendor assigned
Username	FTP/SFTP/SSH username - vendor assigned
Password	FTP/SFTP/SSH password - vendor assigned
Account	Vendor assigned account number associated with your organization
Owner	The organizational unit who owns the EDI account
Last Activity	The date of last activity for the account
Provider	This is a link to one of the "codes" in the "Providers" interface
Path	The path on the vendor's server where Evergreen will send it's outgoing .epo files
Incoming Directory	The path on the vendor's server where "incoming" .epo files are stored
Vendor Account Number	Vendor assigned account number.
Vendor Assigned Code	Usually a sub-account designation. Can be used with or without the Vendor Account Number.

Configuring Organizational Unit SAN code

Look in Administration → Server Administration → Organizational Units

This interface allows a library to configure their SAN, alongside their address, phone, etc.

Troubleshooting

PO JEDI Template Issues

Some libraries may run into issues with the action/trigger (PO JEDI). The template has to be modified to handle different vendor codes that may be used. For instance, if you use "ingra" instead of INGRAM this may cause a problem because they are hardcoded in the template. The following is an example of one modification that seems to work.

Original template has:

```
"buyer":[
    [%   IF   target.provider.edi_default.vendcode && (target.provider.code == 'BT' || target.provider.name.match('(
        {"id-qualifier": 91, "id":"[% target.ordering_agency.mailing_address.san _ ' ' _ target.provider.edi_default
    [%- ELSIF target.provider.edi_default.vendcode && target.provider.code == 'INGRAM' -%]
        {"id":"[% target.ordering_agency.mailing_address.san %]"},
        {"id-qualifier": 91, "id":"[% target.provider.edi_default.vendcode %]"}
    [%- ELSE -%]
        {"id":"[% target.ordering_agency.mailing_address.san %]"}
    [%- END -%]
],
```

Modified template has the following where it matches on provider SAN instead of code:

```
"buyer":[
    [%   IF   target.provider.edi_default.vendcode && (target.provider.san == '1556150')  -%]
        {"id-qualifier": 91, "id":"[% target.ordering_agency.mailing_address.san _ ' ' _ target.provider.edi_default
        {"id-qualifier": 91, "id":"[% target.ordering_agency.mailing_address.san _ ' ' _ target.provider.edi_default
    [%- ELSIF target.provider.edi_default.vendcode && (target.provider.san == '1697978')  -%]
        {"id":"[% target.ordering_agency.mailing_address.san %]"},
        {"id-qualifier": 91, "id":"[% target.provider.edi_default.vendcode %]"}
    [%- ELSE -%]
        {"id":"[% target.ordering_agency.mailing_address.san %]"}
    [%- END -%]
],
```

Chapter 7. Migrating from a legacy system

Introduction

When you migrate to Evergreen, you generally want to migrate the bibliographic records and item information that existed in your previous library system. For anything more than a few thousand records, you should import the data directly into the database rather than use the tools in the staff client. While the data that you can extract from your legacy system varies widely, this section assumes that you or members of your team have the ability to write scripts and are comfortable working with SQL to manipulate data within PostgreSQL. If so, then the following section will guide you towards a method of generating common data formats so that you can then load the data into the database in bulk.

Making electronic resources visible in the catalog

Electronic resources generally do not have any call number or item information associated with them, and Evergreen enables you to easily make bibliographic records visible in the public catalog within sections of the organizational unit hierarchy. For example, you can make a set of bibliographic records visible only to specific branches that have purchased licenses for the corresponding resources, or you can make records representing publicly available electronic resources visible to the entire consortium.

Therefore, to make a record visible in the public catalog, modify the records using your preferred MARC editing approach to ensure the 856 field contains the following information before loading records for electronic resources into Evergreen:

Table 7.1. 856 field for electronic resources: indicators and subfields

Attribute	Value	Note
Indicator 1	4	
Indicator 2	0 or 1	
Subfield u	URL for the electronic resource	
Subfield y	Text content of the link	
Subfield z	Public note	Normally displayed after the link
Subfield 9	Organizational unit short name	The record will be visible when a search is performed specifying this organizational unit or one of its children. You can repeat this subfield as many times as you need.

Once your electronic resource bibliographic records have the required indicators and subfields for each 856 field in the record, you can proceed to load the records using either the command-line bulk import method or the MARC Batch Importer in the staff client.

Migrating your bibliographic records

Convert your MARC21 binary records into the MARCXML format, with one record per line. You can use the following Python script to achieve this goal; just install the *pymarc* library first, and adjust the values of the *input* and *output* variables as needed.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import codecs
import pymarc

input = 'records_in.mrc'
output = 'records_out.xml'

reader = pymarc.MARCReader(open(input, 'rb'), to_unicode=True)
writer = codecs.open(output, 'w', 'utf-8')
for record in reader:
    record.leader = record.leader[:9] + 'a' + record.leader[10:]
    writer.write(pymarc.record_to_xml(record) + "\n")
```

Once you have a MARCXML file with one record per line, you can load the records into your Evergreen system via a staging table in your database.

1. Connect to the PostgreSQL database using the *psql* command. For example:

   ```
   psql -U <user-name> -h <hostname> -d <database>
   ```

2. Create a staging table in the database. The staging table is a temporary location for the raw data that you will load into the production table or tables. Issue the following SQL statement from the *psql* command line, adjusting the name of the table from *staging_records_import*, if desired:

   ```
   CREATE TABLE staging_records_import (id BIGSERIAL, dest BIGINT, marc TEXT);
   ```

3. Create a function that will insert the new records into the production table and update the *dest* column of the staging table. Adjust "staging_records_import" to match the name of the staging table that you plan to create when you issue the following SQL statement:

   ```
   CREATE OR REPLACE FUNCTION staging_importer() RETURNS VOID AS $$
   DECLARE stage RECORD;
   BEGIN
   FOR stage IN SELECT * FROM staging_records_import ORDER BY id LOOP
         INSERT INTO biblio.record_entry (marc, last_xact_id) VALUES (stage.marc, 'IMPORT');
         UPDATE staging_records_import SET dest = currval('biblio.record_entry_id_seq')
          WHERE id = stage.id;
      END LOOP;
     END;
     $$ LANGUAGE plpgsql;
   ```

4. Load the data from your MARCXML file into the staging table using the COPY statement, adjusting for the name of the staging table and the location of your MARCXML file:

   ```
   COPY staging_records_import (marc) FROM '/tmp/records_out.xml';
   ```

5. Load the data from your staging table into the production table by invoking your staging function:

   ```
   SELECT staging_importer();
   ```

When you leave out the *id* value for a *BIGSERIAL* column, the value in the column automatically increments for each new record that you add to the table.

Once you have loaded the records into your Evergreen system, you can search for some known records using the staff client to confirm that the import was successful.

Migrating your call numbers, items, and parts

Holdings, comprised of call numbers, items, and parts, are the set of objects that enable users to locate and potentially acquire materials from your library system.

Call numbers connect libraries to bibliographic records. Each call number has a *label* associated with a classification scheme such as a the Library of Congress or Dewey Decimal systems, and can optionally have either or both a label prefix and a label suffix. Label prefixes and suffixes do not affect the sort order of the label.

Copies connect call numbers to particular instances of that resource at a particular library. Each item has a barcode and must exist in a particular item location. Other optional attributes of items include circulation modifier, which may affect whether that item can circulate or for how long it can circulate, and OPAC visibility, which controls whether that particular item should be visible in the public catalog.

Parts provide more granularity for items, primarily to enable patrons to place holds on individual parts of a set of items. For example, an encyclopedia might be represented by a single bibliographic record, with a single call number representing the label for that encyclopedia at a given library, with 26 items representing each letter of the alphabet, with each item mapped to a different part such as *A*, *B*, *C*, … *Z*.

To migrate this data into your Evergreen system, you will create another staging table in the database to hold the raw data for your materials from which the actual call numbers, items, and parts will be generated.

Begin by connecting to the PostgreSQL database using the *psql* command. For example:

```
psql -U <user-name> -h <hostname> -d <database>
```

Create the staging materials table by issuing the following SQL statement:

```
CREATE TABLE staging_materials (
  bibkey BIGINT,  -- biblio.record_entry_id
  callnum TEXT, -- call number label
  callnum_prefix TEXT, -- call number prefix
  callnum_suffix TEXT, -- call number suffix
  callnum_class TEXT, -- classification scheme
  create_date DATE,
  location TEXT, -- shelving location code
  item_type TEXT, -- circulation modifier code
  owning_lib TEXT, -- org unit code
  barcode TEXT, -- copy barcode
  part TEXT
);
```

For the purposes of this example migration of call numbers, items, and parts, we assume that you are able to create a tab-delimited file containing values that map to the staging table properties, with one item per line. For example, the following 5 lines demonstrate how the file could look for 5 different items, with non-applicable attribute values represented by *\N*, and 3 of the items connected to a single call number and bibliographic record via parts:

```
1   QA 76.76 A3 \N  \N  LC  2012-12-05  STACKS  BOOK    BR1 30007001122620  \N
2   GV 161 V8   Ref.    Juv.    LC  2010-11-11  KIDS    DVD BR2 30007005197073  \N
3   AE 5 E363 1984  \N  \N      LC  1984-01-10  REFERENCE   BOOK    BR1 30007006853385  A
3   AE 5 E363 1984  \N  \N      LC  1984-01-10  REFERENCE   BOOK    BR1 30007006853393  B
3   AE 5 E363 1984  \N  \N      LC  1984-01-10  REFERENCE   BOOK    BR1 30007006853344  C
```

Once your holdings are in a tab-delimited format—which, for the purposes of this example, we will name *holdings.tsv*--you can import the holdings file into your staging table. Copy the contents of the holdings file into the staging table using the *COPY* SQL statement:

```
COPY staging_items (bibkey, callnum, callnum_prefix,
  callnum_suffix, callnum_class, create_date, location,
  item_type, owning_lib, barcode, part) FROM 'holdings.tsv';
```

Generate the item locations you need to represent your holdings:

```
INSERT INTO asset.copy_location (name, owning_lib)
  SELECT DISTINCT location, 1 FROM staging_materials
  WHERE NOT EXISTS (
    SELECT 1 FROM asset.copy_location
    WHERE name = location
  );
```

Generate the circulation modifiers you need to represent your holdings:

```
INSERT INTO config.circ_modifier (code, name, description, sip2_media_type)
  SELECT DISTINCT circmod, circmod, circmod, '001'
  FROM staging_materials
  WHERE NOT EXISTS (
    SELECT 1 FROM config.circ_modifier
    WHERE circmod = code
  );
```

Generate the call number prefixes and suffixes you need to represent your holdings:

```
INSERT INTO asset.call_number_prefix (owning_lib, label)
  SELECT DISTINCT aou.id, callnum_prefix
  FROM staging_materials sm
    INNER JOIN actor.org_unit aou
      ON aou.shortname = sm.owning_lib
  WHERE NOT EXISTS (
    SELECT 1 FROM asset.call_number_prefix acnp
    WHERE callnum_prefix = acnp.label
      AND aou.id = acnp.owning_lib
  ) AND callnum_prefix IS NOT NULL;

INSERT INTO asset.call_number_suffix (owning_lib, label)
  SELECT DISTINCT aou.id, callnum_suffix
  FROM staging_materials sm
    INNER JOIN actor.org_unit aou
      ON aou.shortname = sm.owning_lib
  WHERE NOT EXISTS (
    SELECT 1 FROM asset.call_number_suffix acns
    WHERE callnum_suffix = acns.label
      AND aou.id = acns.owning_lib
  ) AND callnum_suffix IS NOT NULL;
```

Generate the call numbers for your holdings:

```
INSERT INTO asset.call_number (
  creator, editor, record, owning_lib, label, prefix, suffix, label_class
)
  SELECT DISTINCT 1, 1, bibkey, aou.id, callnum, acnp.id, acns.id,
  CASE WHEN callnum_class = 'LC' THEN 1
             WHEN callnum_class = 'DEWEY' THEN 2
  END
  FROM staging_materials sm
    INNER JOIN actor.org_unit aou
      ON aou.shortname = owning_lib
    INNER JOIN asset.call_number_prefix acnp
      ON COALESCE(acnp.label, '') = COALESCE(callnum_prefix, '')
    INNER JOIN asset.call_number_suffix acns
      ON COALESCE(acns.label, '') = COALESCE(callnum_suffix, '')
;
```

Generate the items for your holdings:

```
INSERT INTO asset.copy (
  circ_lib, creator, editor, call_number, location,
 loan_duration, fine_level, barcode
)
  SELECT DISTINCT aou.id, 1, 1, acn.id, acl.id, 2, 2, barcode
  FROM staging_materials sm
    INNER JOIN actor.org_unit aou
      ON aou.shortname = sm.owning_lib
    INNER JOIN asset.copy_location acl
      ON acl.name = sm.location
    INNER JOIN asset.call_number acn
      ON acn.label = sm.callnum
  WHERE acn.deleted IS FALSE
;
```

Generate the parts for your holdings. First, create the set of parts that are required for each record based on your staging materials table:

```
INSERT INTO biblio.monograph_part (record, label)
  SELECT DISTINCT bibkey, part
  FROM staging_materials sm
  WHERE part IS NOT NULL AND NOT EXISTS (
    SELECT 1 FROM biblio.monograph_part bmp
    WHERE sm.part = bmp.label
      AND sm.bibkey = bmp.record
  );
```

Now map the parts for each record to the specific items that you added:

```
INSERT INTO asset.copy_part_map (target_copy, part)
  SELECT DISTINCT acp.id, bmp.id
  FROM staging_materials sm
    INNER JOIN asset.copy acp
      ON acp.barcode = sm.barcode
    INNER JOIN biblio.monograph_part bmp
      ON bmp.record = sm.bibkey
  WHERE part IS NOT NULL
    AND part = bmp.label
    AND acp.deleted IS FALSE
    AND NOT EXISTS (
    SELECT 1 FROM asset.copy_part_map
    WHERE target_copy = acp.id
      AND part = bmp.id
  );
```

At this point, you have loaded your bibliographic records, call numbers, call number prefixes and suffixes, items, and parts, and your records should be visible to searches in the public catalog within the appropriate organization unit scope.

Migrating Patron Data

Introduction

This section will explain the task of migrating your patron data from comma delimited files into Evergreen. It does not deal with the process of exporting from the non-Evergreen system since this process may vary depending on where you are extracting your patron records. Patron could come from an ILS or it could come from a student database in the case of academic records.

When importing records into Evergreen you will need to populate 3 tables in your Evergreen database:

- actor.usr - The main table for user data
- actor.card - Stores the barcode for users; Users can have more than 1 card but only 1 can be active at a given time;

- actor.usr_address - Used for storing address information; A user can have more than one address.

Before following the procedures below to import patron data into Evergreen, it is a good idea to examine the fields in these tables in order to decide on a strategy for data to include in your import. It is important to understand the data types and constraints on each field.

1. Export the patron data from your existing ILS or from another source into a comma delimited file. The comma delimited file used for importing the records should use Unicode (UTF8) character encoding.

2. Create a staging table. A staging table will allow you to tweak the data before importing. Here is an example sql statement:

```
CREATE TABLE students (
     student_id int, barcode text, last_name text, first_name text, email text,
             address_type text, street1 text, street2 text,
    city text, province text, country text, postal_code text, phone text, profile
    int DEFAULT 2, ident_type int, home_ou int, claims_returned_count int DEFAULT
            0, usrname text, net_access_level int DEFAULT 2, password text
);
```

The *default* variables allow you to set default for your library or to populate required fields in Evergreen if your data includes NULL values.

   The data field profile in the above SQL script refers to the user group and should be an integer referencing the id field in permission.grp_tree. Setting this value will affect the permissions for the user. See the values in permission.grp_tree for possibilities.

   ident_type is the identification type used for identifying users. This is a integer value referencing config.identification_type and should match the id values of that table. The default values are 1 for Drivers License, 2 for SSN or 3 for other.

   home_ou is the home organizational unit for the user. This value needs to match the corresponding id in the actor.org_unit table.

3. Copy records into staging table from a comma delimited file.

```
COPY students (student_id, last_name, first_name, email, address_type, street1, street2,
    city, province, country, postal_code, phone)
    FROM '/home/opensrf/patrons.csv'
            WITH CSV HEADER;
```

   The script will vary depending on the format of your patron load file (patrons.csv).

4. Formatting of some fields to fit Evergreen filed formatting may be required. Here is an example of sql to adjust phone numbers in the staging table to fit the evergreen field:

```
UPDATE students phone = replace(replace(replace(rpad(substring(phone from 1 for 9), 10, '-') ||
substring(phone from 10), '(', ''), ')', ''), ' ', '-');
```

   Data “massaging” will be required to fit formats used in Evergreen.

5. Insert records from the staging table into the actor.usr Evergreen table:

```
 INSERT INTO actor.usr (
    profile, usrname, email, passwd, ident_type, ident_value, first_given_name,
    family_name, day_phone, home_ou, claims_returned_count, net_access_level)
    SELECT profile, students.usrname, email, password, ident_type, student_id,
    first_name, last_name, phone, home_ou, claims_returned_count, net_access_level
    FROM students;
```

6. Insert records into actor.card from actor.usr .

```
INSERT INTO actor.card (usr, barcode)
    SELECT actor.usr.id, students.barcode
    FROM students
            INNER JOIN actor.usr
                    ON students.usrname = actor.usr.usrname;
```

   This assumes a one to one card patron relationship. If your patron data import has multiple cards assigned to one patron more complex import scripts may be required which look for inactive or active flags.

7. Update actor.usr.card field with actor.card.id to associate active card with the user:

```
UPDATE actor.usr
    SET card = actor.card.id
    FROM actor.card
    WHERE actor.card.usr = actor.usr.id;
```

8. Insert records into actor.usr_address to add address information for users:

```
INSERT INTO actor.usr_address (usr, street1, street2, city, state, country, post_code)
    SELECT actor.usr.id, students.street1, students.street2, students.city, students.province,
    students.country, students.postal_code
    FROM students
    INNER JOIN actor.usr ON students.usrname = actor.usr.usrname;
```

9. Update actor.usr.address with address id from address table.

```
UPDATE actor.usr
    SET mailing_address = actor.usr_address.id, billing_address = actor.usr_address.id
    FROM actor.usr_address
    WHERE actor.usr.id = actor.usr_address.usr;
```

This assumes 1 address per patron. More complex scenarios may require more sophisticated SQL.

Creating an sql Script for Importing Patrons

The procedure for importing patron can be automated with the help of an sql script. Follow these steps to create an import script:

1. Create an new file and name it import.sql

2. Edit the file to look similar to this:

```
BEGIN;

-- Remove any old staging table.
DROP TABLE IF EXISTS students;

-- Create staging table.
CREATE TABLE students (
    student_id text, barcode text, last_name text, first_name text, email text, address_type text,
            street1 text, street2 text, city text, province text, country text, postal_code text, phone
            text, profile int, ident_type int, home_ou int, claims_returned_count int DEFAULT 0, usrname text,
    net_access_level int DEFAULT 2, password text, already_exists boolean DEFAULT FALSE
);

--Copy records from your import text file
COPY students (student_id, last_name, first_name, email, address_type, street1, street2, city, province,
    country, postal_code, phone, password)
    FROM '/home/opensrf/patrons.csv' WITH CSV HEADER;

--Determine which records are new, and which are merely updates of existing patrons
--You may with to also add a check on the home_ou column here, so that you don't
--accidentally overwrite the data of another library in your consortium.
--You may also use a different matchpoint than actor.usr.ident_value.
UPDATE students
    SET already_exists = TRUE
    FROM actor.usr
    WHERE students.student_id = actor.usr.ident_value;

--Update the names of existing patrons, in case they have changed their name
UPDATE actor.usr
    SET first_given_name = students.first_name, family_name=students.last_name
    FROM students
    WHERE actor.usr.ident_value=students.student_id
    AND (first_given_name != students.first_name OR family_name != students.last_name)
    AND students.already_exists;

--Update email addresses of existing patrons
--You may wish to update other fields as well, while preserving others
--actor.usr.passwd is an example of a field you may not wish to update,
--since patrons may have set the password to something other than the
--default.
UPDATE actor.usr
    SET email=students.email
    FROM students
    WHERE actor.usr.ident_value=students.student_id
    AND students.email != ''
    AND actor.usr.email != students.email
    AND students.already_exists;

--Insert records from the staging table into the actor.usr table.
INSERT INTO actor.usr (
    profile, usrname, email, passwd, ident_type, ident_value, first_given_name, family_name,
    day_phone, home_ou, claims_returned_count, net_access_level)
    SELECT profile, students.usrname, email, password, ident_type, student_id, first_name,
    last_name, phone, home_ou, claims_returned_count, net_access_level
    FROM students WHERE NOT already_exists;

--Insert records from the staging table into the actor.card table.
INSERT INTO actor.card (usr, barcode)
    SELECT actor.usr.id, students.barcode
    FROM students
            INNER JOIN actor.usr
                    ON students.usrname = actor.usr.usrname
    WHERE NOT students.already_exists;

--Update actor.usr.card field with actor.card.id to associate active card with the user:
UPDATE actor.usr
    SET card = actor.card.id
    FROM actor.card
    WHERE actor.card.usr = actor.usr.id;

--INSERT records INTO actor.usr_address from staging table.
INSERT INTO actor.usr_address (usr, street1, street2, city, state, country, post_code)
    SELECT actor.usr.id, students.street1, students.street2, students.city, students.province,
    students.country, students.postal_code
    FROM students
    INNER JOIN actor.usr ON students.usrname = actor.usr.usrname
    WHERE NOT students.already_exists;
```

Placing the sql statements between BEGIN; and COMMIT; creates a transaction block so that if any sql statements fail, the entire process is canceled and the database is rolled back to its original state. Lines beginning with — are comments to let you you what each sql statement is doing and are not processed.

Batch Updating Patron Data

For academic libraries, doing batch updates to add new patrons to the Evergreen database is a critical task. The above procedures and import script can be easily adapted to create an update script for importing new patrons from external databases. If the data import file contains only new patrons, then, the above procedures will work well to insert those patrons. However, if the data load contains all patrons, a second staging table and a procedure to remove existing patrons from that second staging table may be required before importing the new patrons. Moreover, additional steps to update address information and perhaps delete inactive patrons may also be desired depending on the requirements of the institution.

After developing the scripts to import and update patrons have been created, another important task for library staff is to develop an import strategy and schedule which suits the needs of the library. This could be determined by registration dates of your institution in the case of academic libraries. It is important to balance the convenience of patron loads and the cost of processing these loads vs staff adding patrons manually.

Part III. Individual Evergreen Components

Table of Contents

8. Easing gently into OpenSRF 44
    Abstract 44
    Introducing OpenSRF 44
        Programming language support 44
        OpenSRF communication flows over XMPP 45
        OpenSRF communication flows over HTTP 45
        Stateless and stateful connections 47
    Enough jibber-jabber: writing an OpenSRF service 48
        Registering a service with the OpenSRF configuration files 49
        Calling an OpenSRF method 51
        Accepting and returning more interesting data types 54
        Accepting and returning Evergreen objects 54
        Returning streaming results 56
        Error! Warning! Info! Debug! 57
        Caching results: one secret of scalability 58
        Initializing the service and its children: child labour 59
        Retrieving configuration settings 59
    Getting under the covers with OpenSRF 60
        Get on the messaging bus - safely 60
        Message body format 61
        Registering OpenSRF methods in depth 62
    Evergreen-specific OpenSRF services 62
    Evergreen after one year: reflections on OpenSRF 64
        Strengths of OpenSRF 64
        Weaknesses 65
    Summary 65
    Appendix: Python client 65
9. Support Scripts 67
    authority_control_fields: Connecting Bibliographic and Authority records 68
    marc_export: Exporting Bibliographic Records into MARC files 68
        Options 69
    Parallel Ingest with pingest.pl 70
        Command Line Options 71
    Importing Authority Records from Command Line 71
    Juvenile-to-adult batch script 72
    MARC Stream Importer 72
    Processing Action Triggers 73
10. Daemons and services 75
    Starting and Stopping the Reporter Daemon 75
        Starting the Reporter Daemon 75
        Stopping the Reporter Daemon 75
    ebook_api service 76
    hold-targeter service 76
    QStore service 76
11. Developing with pgTAP tests 77
    Setting up pgTAP on your development server 77
    Running pgTAP tests 77

Chapter 8. Easing gently into OpenSRF

Abstract

The Evergreen open-source library system serves library consortia composed of hundreds of branches with millions of patrons - for example, the Georgia Public Library Service PINES system. One of the claimed advantages of Evergreen over alternative integrated library systems is the underlying Open Service Request Framework (OpenSRF, pronounced "open surf") architecture. This article introduces OpenSRF, demonstrates how to build OpenSRF services through simple code examples, and explains the technical foundations on which OpenSRF is built.

Introducing OpenSRF

OpenSRF is a message routing network that offers scalability and failover support for individual services and entire servers with minimal development and deployment overhead. You can use OpenSRF to build loosely-coupled applications that can be deployed on a single server or on clusters of geographically distributed servers using the same code and minimal configuration changes. Although copyright statements on some of the OpenSRF code date back to Mike Rylander's original explorations in 2000, Evergreen was the first major application to be developed with, and to take full advantage of, the OpenSRF architecture starting in 2004. The first official release of OpenSRF was 0.1 in February 2005 (http://evergreen-ils.org/blog/?p=21), but OpenSRF's development continues a steady pace of enhancement and refinement, with the release of 1.0.0 in October 2008 and the most recent release of 1.2.2 in February 2010.

OpenSRF is a distinct break from the architectural approach used by previous library systems and has more in common with modern Web applications. The traditional "scale-up" approach to serve more transactions is to purchase a server with more CPUs and more RAM, possibly splitting the load between a Web server, a database server, and a business logic server. Evergreen, however, is built on the Open Service Request Framework (OpenSRF) architecture, which firmly embraces the "scale-out" approach of spreading transaction load over cheap commodity servers. The initial GPLS PINES hardware cluster, while certainly impressive, may have offered the misleading impression that Evergreen is complex and requires a lot of hardware to run.

This article hopes to correct any such lingering impression by demonstrating that OpenSRF itself is an extremely simple architecture on which one can easily build applications of many kinds – not just library applications – and that you can use a number of different languages to call and implement OpenSRF methods with a minimal learning curve. With an application built on OpenSRF, when you identify a bottleneck in your application's business logic layer, you can adjust the number of the processes serving that particular bottleneck on each of your servers; or if the problem is that your service is resource-hungry, you could add an inexpensive server to your cluster and dedicate it to running that resource-hungry service.

Programming language support

If you need to develop an entirely new OpenSRF service, you can choose from a number of different languages in which to implement that service. OpenSRF client language bindings have been written for C, Java, JavaScript, Perl, and Python, and server language bindings have been written for C, Perl, and Python. This article uses Perl examples as a lowest common denominator programming language. Writing an OpenSRF binding for another language is a relatively small task if that language offers libraries that support the core technologies on which OpenSRF depends:

- Extensible Messaging and Presence Protocol (XMPP, sometimes referred to as Jabber) - provides the base messaging infrastructure between OpenSRF clients and servers

- JavaScript Object Notation (JSON) - serializes the content of each XMPP message in a standardized and concise format
- memcached - provides the caching service
- syslog - the standard UNIX logging service

Unfortunately, the OpenSRF reference documentation, although augmented by the OpenSRF glossary, blog posts like the description of OpenSRF and Jabber, and even this article, is not a sufficient substitute for a complete specification on which one could implement a language binding. The recommended option for would-be developers of another language binding is to use the Python implementation as the cleanest basis for a port to another language.

OpenSRF communication flows over XMPP

The XMPP messaging service underpins OpenSRF, requiring an XMPP server such as ejabberd. When you start OpenSRF, the first XMPP clients that connect to the XMPP server are the OpenSRF public and private *routers*. OpenSRF routers maintain a list of available services and connect clients to available services. When an OpenSRF service starts, it establishes a connection to the XMPP server and registers itself with the private router. The OpenSRF configuration contains a list of public OpenSRF services, each of which must also register with the public router. Services and clients connect to the XMPP server using a single set of XMPP client credentials (for example, `opensrf@private.localhost`), but use XMPP resource identifiers to differentiate themselves in the Jabber ID (JID) for each connection. For example, the JID for a copy of the `opensrf.simple-text` service with process ID `6285` that has connected to the `private.localhost` domain using the `opensrf` XMPP client credentials could be `opensrf@private.localhost/opensrf.simple-text_drone_at_localhost_6285`.

OpenSRF communication flows over HTTP

Any OpenSRF service registered with the public router is accessible via the OpenSRF HTTP Translator. The OpenSRF HTTP Translator implements the OpenSRF-over-HTTP proposed specification as an Apache module that translates HTTP requests into OpenSRF requests and returns OpenSRF results as HTTP results to the initiating HTTP client.

Issuing an HTTP POST request to an OpenSRF method via the OpenSRF HTTP Translator.

```
# curl request broken up over multiple lines for legibility
curl -H "X-OpenSRF-service: opensrf.simple-text"                                \ #
    --data 'osrf-msg=[                                                          \ #
        {"__c":"osrfMessage","__p":{"threadTrace":0,"locale":"en-CA",           \ #
            "type":"REQUEST","payload": {"__c":"osrfMethod","__p":              \
                {"method":"opensrf.simple-text.reverse","params":["foobar"]}    \
            }}                                                                  \
        }]'                                                                     \
http://localhost/osrf-http-translator                                           \ #
```

The `X-OpenSRF-service` header identifies the OpenSRF service of interest.
The POST request consists of a single parameter, the `osrf-msg` value, which contains a JSON array.
The first object is an OpenSRF message (`"__c":"osrfMessage"`) with a set of parameters (`"__p":{}`) containing:

- the identifier for the request (`"threadTrace":0`); this value is echoed back in the result
- the message type (`"type":"REQUEST"`)

- the locale for the message; if the OpenSRF method is locale-sensitive, it can check the locale for each OpenSRF request and return different information depending on the locale

- the payload of the message (`"payload":{}`) containing the OpenSRF method request (`"__c":"osrfMethod"`) and its parameters (`"__p":{}`), which in turn contains:

  - the method name for the request (`"method":"opensrf.simple-text.reverse"`)

  - a set of JSON parameters to pass to the method (`"params":["foobar"]`); in this case, a single string `"foobar"`

The URL on which the OpenSRF HTTP translator is listening, `/osrf-http-translator` is the default location in the Apache example configuration files shipped with the OpenSRF source, but this is configurable.

Results from an HTTP POST request to an OpenSRF method via the OpenSRF HTTP Translator.

```
# HTTP response broken up over multiple lines for legibility
[{"__c":"osrfMessage","__p":                                                \ #
    {"threadTrace":0, "payload":                                            \ #
        {"__c":"osrfResult","__p":                                          \ #
            {"status":"OK","content":"raboof","statusCode":200}             \ #
        },"type":"RESULT","locale":"en-CA"                                  \ #
    }
},
{"__c":"osrfMessage","__p":                                                \ #
    {"threadTrace":0,"payload":                                             \ #
        {"__c":"osrfConnectStatus","__p":                                   \ #
            {"status":"Request Complete","statusCode":205}                  \ #
        },"type":"STATUS","locale":"en-CA"                                  \ #
    }
}]
```

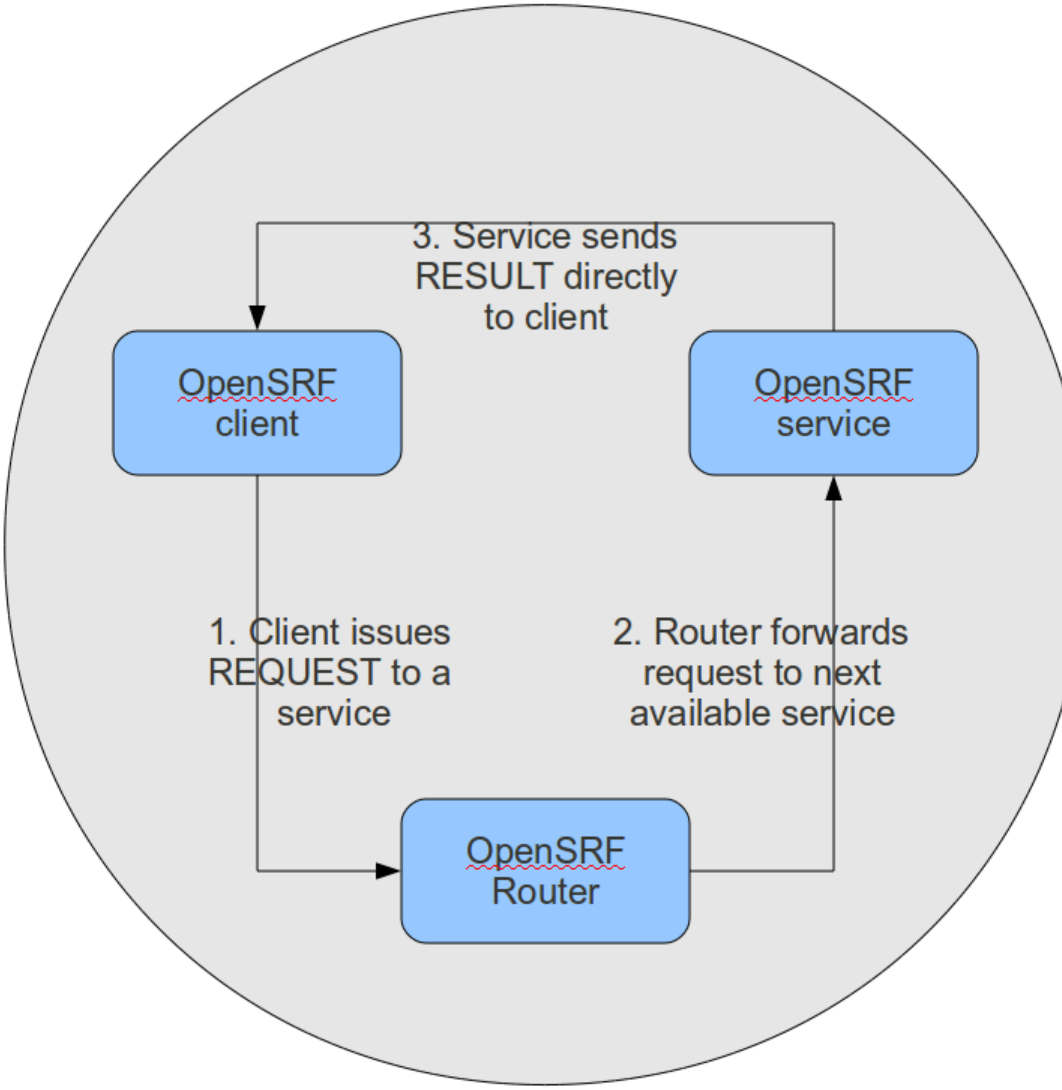
The OpenSRF HTTP Translator returns an array of JSON objects in its response. Each object in the response is an OpenSRF message (`"__c":"osrfMessage"`) with a collection of response parameters (`"__p":`).
The OpenSRF message identifier (`"threadTrace":0`) confirms that this message is in response to the request matching the same identifier.
The message includes a payload JSON object (`"payload":`) with an OpenSRF result for the request (`"__c":"osrfResult"`).
The result includes a status indicator string (`"status":"OK"`), the content of the result response - in this case, a single string "raboof" (`"content":"raboof"`) - and an integer status code for the request (`"statusCode":200`).
The message also includes the message type (`"type":"RESULT"`) and the message locale (`"locale":"en-CA"`).
The second message in the set of results from the response.
Again, the message identifier confirms that this message is in response to a particular request.
The payload of the message denotes that this message is an OpenSRF connection status message (`"__c":"osrfConnectStatus"`), with some information about the particular OpenSRF connection that was used for this request.
The response parameters for an OpenSRF connection status message include a verbose status (`"status":"Request Complete"`) and an integer status code for the connection status (`"statusCode":205).
The message also includes the message type (`"type":"RESULT"`) and the message locale (`"locale":"en-CA"`).

Before adding a new public OpenSRF service, ensure that it does not introduce privilege escalation or unchecked access to data. For example, the Evergreen `open-ils.cstore` private service is an object-relational mapper that provides read and write access to the entire Evergreen database, so it would be catastrophic to expose that service publicly. In comparison, the Evergreen `open-ils.pcrud` public service offers the same functionality as `open-ils.cstore` to any connected HTTP client or OpenSRF client, but the additional authentication and authorization layer in `open-ils.pcrud` prevents unchecked access to Evergreen's data.

Stateless and stateful connections

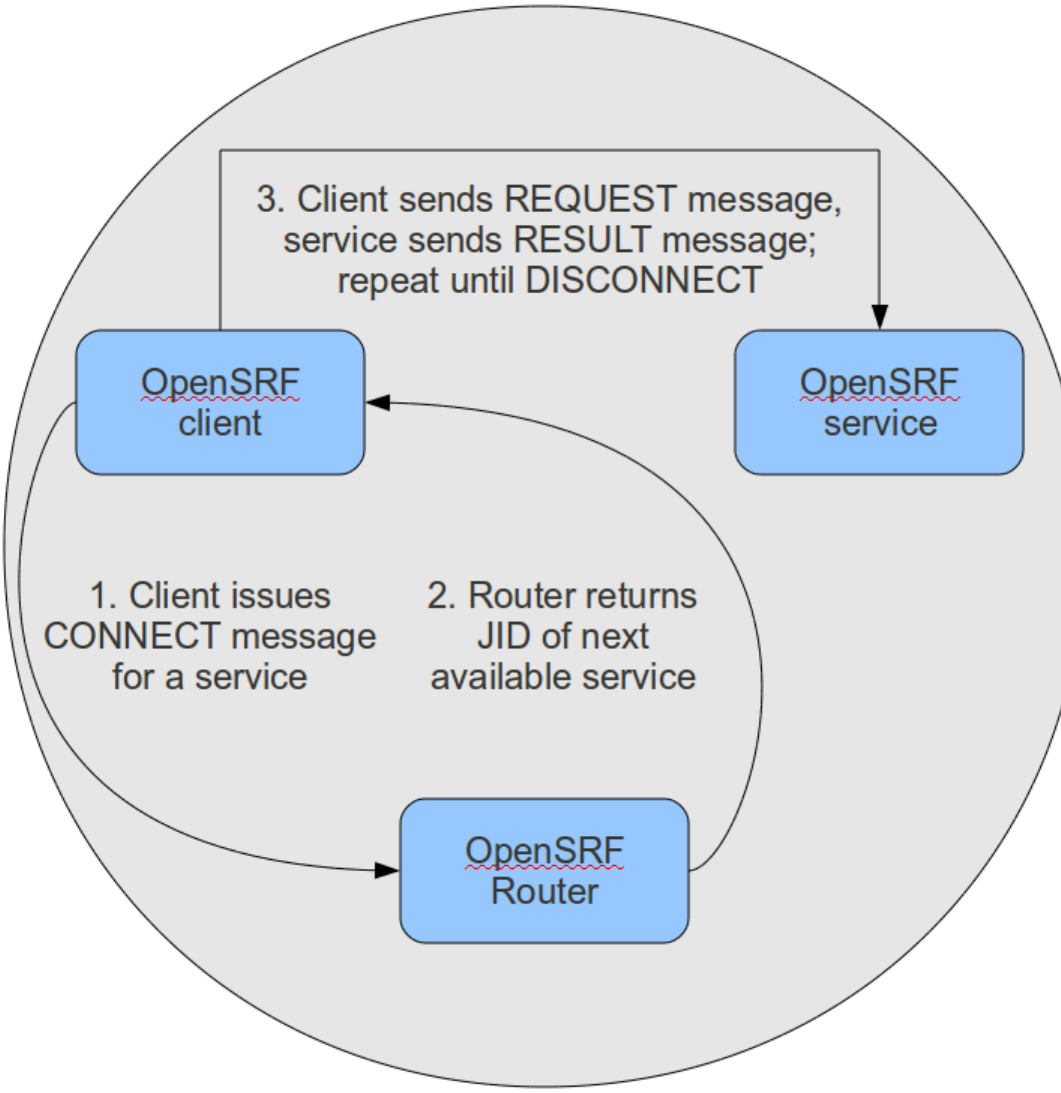
OpenSRF supports both *stateless* and *stateful* connections. When an OpenSRF client issues a `REQUEST` message in a *stateless* connection, the router forwards the request to the next available service and the service returns the result directly to the client.

REQUEST flow in a stateless connection.



When an OpenSRF client issues a `CONNECT` message to create a *stateful* connection, the router returns the Jabber ID of the next available service to the client so that the client can issue one or more `REQUEST` message directly to that particular service and the service will return corresponding `RESULT` messages directly to the client. Until the client issues a `DISCONNECT` message, that particular service is only available to the requesting client. Stateful connections are useful for clients that need to make many requests from a particular service, as it avoids the intermediary step of contacting the router for each request, as well as for operations that require a controlled sequence of commands, such as a set of database INSERT, UPDATE, and DELETE statements within a transaction.

CONNECT, REQUEST, and DISCONNECT flow in a stateful connection.



Enough jibber-jabber: writing an OpenSRF service

Imagine an application architecture in which 10 lines of Perl or Python, using the data types native to each language, are enough to implement a method that can then be deployed and invoked seamlessly across hundreds

of servers. You have just imagined developing with OpenSRF – it is truly that simple. Under the covers, of course, the OpenSRF language bindings do an incredible amount of work on behalf of the developer. An OpenSRF application consists of one or more OpenSRF services that expose methods: for example, the `opensrf.simple-text` demonstration service exposes the `opensrf.simple-text.split()` and `opensrf.simple-text.reverse()` methods. Each method accepts zero or more arguments and returns zero or one results. The data types supported by OpenSRF arguments and results are typical core language data types: strings, numbers, booleans, arrays, and hashes.

To implement a new OpenSRF service, perform the following steps:

1. Include the base OpenSRF support libraries
2. Write the code for each of your OpenSRF methods as separate procedures
3. Register each method
4. Add the service definition to the OpenSRF configuration files

For example, the following code implements an OpenSRF service. The service includes one method named `opensrf.simple-text.reverse()` that accepts one string as input and returns the reversed version of that string:

```
#!/usr/bin/perl

package OpenSRF::Application::Demo::SimpleText;

use strict;

use OpenSRF::Application;
use parent qw/OpenSRF::Application/;

sub text_reverse {
    my ($self , $conn, $text) = @_;
    my $reversed_text = scalar reverse($text);
    return $reversed_text;
}

__PACKAGE__->register_method(
    method    => 'text_reverse',
    api_name  => 'opensrf.simple-text.reverse'
);
```

Ten lines of code, and we have a complete OpenSRF service that exposes a single method and could be deployed quickly on a cluster of servers to meet your application's ravenous demand for reversed strings! If you're unfamiliar with Perl, the `use OpenSRF::Application; use parent qw/OpenSRF::Application/;` lines tell this package to inherit methods and properties from the `OpenSRF::Application` module. For example, the call to `__PACKAGE__->register_method()` is defined in `OpenSRF::Application` but due to inheritance is available in this package (named by the special Perl symbol `__PACKAGE__` that contains the current package name). The `register_method()` procedure is how we introduce a method to the rest of the OpenSRF world.

Registering a service with the OpenSRF configuration files

Two files control most of the configuration for OpenSRF:

- `opensrf.xml` contains the configuration for the service itself as well as a list of which application servers in your OpenSRF cluster should start the service

- `opensrf_core.xml` (often referred to as the "bootstrap configuration" file) contains the OpenSRF networking information, including the XMPP server connection credentials for the public and private routers; you only need to touch this for a new service if the new service needs to be accessible via the public router

Begin by defining the service itself in `opensrf.xml`. To register the `opensrf.simple-text` service, add the following section to the `<apps>` element (corresponding to the XPath `/opensrf/default/apps/`):

```
<apps>
  <opensrf.simple-text>                                                  <!--    -->
    <keepalive>3</keepalive>                                             <!--    -->
    <stateless>1</stateless>                                             <!--    -->
    <language>perl</language>                                            <!--    -->
    <implementation>OpenSRF::Application::Demo::SimpleText</implementation>  <!--    -->
    <max_requests>100</max_requests>                                     <!--    -->
    <unix_config>
      <max_requests>1000</max_requests>                                  <!--    -->
      <unix_log>opensrf.simple-text_unix.log</unix_log>                  <!--    -->
      <unix_sock>opensrf.simple-text_unix.sock</unix_sock>               <!--    -->
      <unix_pid>opensrf.simple-text_unix.pid</unix_pid>                 <!--    -->
      <min_children>5</min_children>                                     <!--    -->
      <max_children>15</max_children>                                    <!--    -->
      <min_spare_children>2</min_spare_children>                         <!--    -->
      <max_spare_children>5</max_spare_children>                         <!--    -->
    </unix_config>
  </opensrf.simple-text>

  <!-- other OpenSRF services registered here... -->
</apps>
```

The element name is the name that the OpenSRF control scripts use to refer to the service.

Specifies the interval (in seconds) between checks to determine if the service is still running.

Specifies whether OpenSRF clients can call methods from this service without first having to create a connection to a specific service backend process for that service. If the value is `1`, then the client can simply issue a request and the router will forward the request to an available service and the result will be returned directly to the client.

Specifies the programming language in which the service is implemented

Specifies the name of the library or module in which the service is implemented

(C implementations): Specifies the maximum number of requests a process serves before it is killed and replaced by a new process.

(Perl implementations): Specifies the maximum number of requests a process serves before it is killed and replaced by a new process.

The name of the log file for language-specific log messages such as syntax warnings.

The name of the UNIX socket used for inter-process communications.

The name of the PID file for the master process for the service.

The minimum number of child processes that should be running at any given time.

The maximum number of child processes that should be running at any given time.

The minimum number of child processes that should be available to handle incoming requests. If there are fewer than this number of spare child processes, new processes will be spawned.

The maximum number of child processes that should be available to handle incoming requests. If there are more than this number of spare child processes, the extra processes will be killed.

To make the service accessible via the public router, you must also edit the `opensrf_core.xml` configuration file to add the service to the list of publicly accessible services:

Making a service publicly accessible in `opensrf_core.xml`.

```
<router>                                                                    <!--    -->
    <!-- This is the public router. On this router, we only register applications
     which should be accessible to everyone on the opensrf network -->
    <name>router</name>
    <domain>public.localhost</domain>                                      <!--    -->
    <services>
        <service>opensrf.math</service>
        <service>opensrf.simple-text</service>                             <!--    -->
    </services>
</router>
```

This section of the `opensrf_core.xml` file is located at XPath `/config/opensrf/routers/`. `public.localhost` is the canonical public router domain in the OpenSRF installation instructions. Each `<service>` element contained in the `<services>` element offers their services via the public router as well as the private router.

Once you have defined the new service, you must restart the OpenSRF Router to retrieve the new configuration and start or restart the service itself.

Calling an OpenSRF method

OpenSRF clients in any supported language can invoke OpenSRF services in any supported language. So let's see a few examples of how we can call our fancy new `opensrf.simple-text.reverse()` method:

Calling OpenSRF methods from the srfsh client

`srfsh` is a command-line tool installed with OpenSRF that you can use to call OpenSRF methods. To call an OpenSRF method, issue the `request` command and pass the OpenSRF service and method name as the first two arguments; then pass a list of JSON objects as the arguments to the method being invoked.

The following example calls the `opensrf.simple-text.reverse` method of the `opensrf.simple-text` OpenSRF service, passing the string `"foobar"` as the only method argument:

```
$ srfsh
srfsh # request opensrf.simple-text opensrf.simple-text.reverse "foobar"

Received Data: "raboof"

=------------------------------------
Request Completed Successfully
Request Time in seconds: 0.016718
=------------------------------------
```

Getting documentation for OpenSRF methods from the srfsh client

The `srfsh` client also gives you command-line access to retrieving metadata about OpenSRF services and methods. For a given OpenSRF method, for example, you can retrieve information such as the minimum number of required arguments, the data type and a description of each argument, the package or library in which the method is implemented, and a description of the method. To retrieve the documentation for an opensrf method from `srfsh`, issue the `introspect` command, followed by the name of the OpenSRF service and (optionally) the name of the OpenSRF method. If you do not pass a method name to the `introspect` command, `srfsh` lists all of the methods offered by the service. If you pass a partial method name, `srfsh` lists all of the methods that match that portion of the method name.

The quality and availability of the descriptive information for each method depends on the developer to register the method with complete and accurate information. The quality varies across the set of OpenSRF and Evergreen APIs, although some effort is being put towards improving the state of the internal documentation.

```
srfsh# introspect opensrf.simple-text "opensrf.simple-text.reverse"
--> opensrf.simple-text

Received Data: {
  "__c":"opensrf.simple-text",
  "__p":{
    "api_level":1,
    "stream":0,                                                    \ #
    "object_hint":"OpenSRF_Application_Demo_SimpleText",
    "remote":0,
    "package":"OpenSRF::Application::Demo::SimpleText",            \ #
    "api_name":"opensrf.simple-text.reverse",                      \ #
    "server_class":"opensrf.simple-text",
    "signature":{                                                  \ #
      "params":[                                                   \ #
        {
          "desc":"The string to reverse",
          "name":"text",
          "type":"string"
        }
      ],
      "desc":"Returns the input string in reverse order\n",        \ #
      "return":{                                                   \ #
        "desc":"Returns the input string in reverse order",
        "type":"string"
      }
    },
    "method":"text_reverse",                                       \ #
    "argc":1                                                       \ #
  }
}
```

`stream` denotes whether the method supports streaming responses or not.
`package` identifies which package or library implements the method.
`api_name` identifies the name of the OpenSRF method.
`signature` is a hash that describes the parameters for the method.
`params` is an array of hashes describing each parameter in the method; each parameter has a description (`desc`), name (`name`), and type (`type`).
`desc` is a string that describes the method itself.
`return` is a hash that describes the return value for the method; it contains a description of the return value (`desc`) and the type of the returned value (`type`).
`method` identifies the name of the function or method in the source implementation.
`argc` is an integer describing the minimum number of arguments that must be passed to this method.

Calling OpenSRF methods from Perl applications

To call an OpenSRF method from Perl, you must connect to the OpenSRF service, issue the request to the method, and then retrieve the results.

```
#/usr/bin/perl
use strict;
use OpenSRF::AppSession;
use OpenSRF::System;

OpenSRF::System->bootstrap_client(config_file => '/openils/conf/opensrf_core.xml'); #

my $session = OpenSRF::AppSession->create("opensrf.simple-text");                    #

print "substring: Accepts a string and a number as input, returns a string\n";
my $result = $session->request("opensrf.simple-text.substring", "foobar", 3);       #
my $request = $result->gather();                                                     #
print "Substring: $request\n\n";

print "split: Accepts two strings as input, returns an array of strings\n";
$request = $session->request("opensrf.simple-text.split", "This is a test", " ");   #
my $output = "Split: [";
my $element;
while ($element = $request->recv()) {                                                #
    $output .= $element->content . ", ";                                             #
}
$output =~ s/, $/]/;
print $output . "\n\n";

print "statistics: Accepts an array of strings as input, returns a hash\n";
my @many_strings = [
    "First I think I'll have breakfast",
    "Then I think that lunch would be nice",
    "And then seventy desserts to finish off the day"
];

$result = $session->request("opensrf.simple-text.statistics", @many_strings);       #
$request = $result->gather();                                                        #
print "Length: " . $result->{'length'} . "\n";
print "Word count: " . $result->{'word_count'} . "\n";

$session->disconnect();                                                              #
```

The `OpenSRF::System->bootstrap_client()` method reads the OpenSRF configuration information from the indicated file and creates an XMPP client connection based on that information.

The `OpenSRF::AppSession->create()` method accepts one argument - the name of the OpenSRF service to which you want to want to make one or more requests - and returns an object prepared to use the client connection to make those requests.

The `OpenSRF::AppSession->request()` method accepts a minimum of one argument - the name of the OpenSRF method to which you want to make a request - followed by zero or more arguments to pass to the OpenSRF method as input values. This example passes a string and an integer to the `opensrf.simple-text.substring` method defined by the `opensrf.simple-text` OpenSRF service.

The `gather()` method, called on the result object returned by the `request()` method, iterates over all of the possible results from the result object and returns a single variable.

This `request()` call passes two strings to the `opensrf.simple-text.split` method defined by the `opensrf.simple-text` OpenSRF service and returns (via `gather()`) a reference to an array of results.

The `opensrf.simple-text.split()` method is a streaming method that returns an array of results with one element per `recv()` call on the result object. We could use the `gather()` method to retrieve all of the results in a single array reference, but instead we simply iterate over the result variable until there are no more results to retrieve.

While the `gather()` convenience method returns only the content of the complete set of results for a given request, the `recv()` method returns an OpenSRF result object with `status`, `statusCode`, and `content` fields as we saw in the HTTP results example.

This `request()` call passes an array to the `opensrf.simple-text.statistics` method defined by the `opensrf.simple-text` OpenSRF service.

The result object returns a hash reference via `gather()`. The hash contains the `length` and `word_count` keys we defined in the method.

The `OpenSRF::AppSession->disconnect()` method closes the XMPP client connection and cleans up resources associated with the session.

Accepting and returning more interesting data types

Of course, the example of accepting a single string and returning a single string is not very interesting. In real life, our applications tend to pass around multiple arguments, including arrays and hashes. Fortunately, OpenSRF makes that easy to deal with; in Perl, for example, returning a reference to the data type does the right thing. In the following example of a method that returns a list, we accept two arguments of type string: the string to be split, and the delimiter that should be used to split the string.

Text splitting method - streaming mode.

```
sub text_split {
    my $self = shift;
    my $conn = shift;
    my $text = shift;
    my $delimiter = shift || ' ';

    my @split_text = split $delimiter, $text;
    return \@split_text;
}

__PACKAGE__->register_method(
    method    => 'text_split',
    api_name  => 'opensrf.simple-text.split'
);
```

We simply return a reference to the list, and OpenSRF does the rest of the work for us to convert the data into the language-independent format that is then returned to the caller. As a caller of a given method, you must rely on the documentation used to register to determine the data structures - if the developer has added the appropriate documentation.

Accepting and returning Evergreen objects

OpenSRF is agnostic about objects; its role is to pass JSON back and forth between OpenSRF clients and services, and it allows the specific clients and services to define their own semantics for the JSON structures. On top of that infrastructure, Evergreen offers the fieldmapper: an object-relational mapper that provides a complete definition of all objects, their properties, their relationships to other objects, the permissions required to create, read, update, or delete objects of that type, and the database table or view on which they are based.

The Evergreen fieldmapper offers a great deal of convenience for working with complex system objects beyond the basic mapping of classes to database schemas. Although the result is passed over the wire as a JSON object containing the indicated fields, fieldmapper-aware clients then turn those JSON objects into native objects with setter / getter methods for each field.

All of this metadata about Evergreen objects is defined in the fieldmapper configuration file (`/openils/conf/fm_IDL.xml`), and access to these classes is provided by the `open-ils.cstore`, `open-ils.pcrud`, and `open-ils.reporter-store` OpenSRF services which parse the fieldmapper configuration file and dynamically register OpenSRF methods for creating, reading, updating, and deleting all of the defined classes.

Example fieldmapper class definition for "Open User Summary".

```
<class id="mous" controller="open-ils.cstore open-ils.pcrud"
 oils_obj:fieldmapper="money::open_user_summary"
 oils_persist:tablename="money.open_usr_summary"
 reporter:label="Open User Summary">                                        <!--    -->
    <fields oils_persist:primary="usr" oils_persist:sequence="">            <!--    -->
        <field name="balance_owed" reporter:datatype="money" />             <!--    -->
        <field name="total_owed" reporter:datatype="money" />
        <field name="total_paid" reporter:datatype="money" />
        <field name="usr" reporter:datatype="link"/>
    </fields>
    <links>
        <link field="usr" reltype="has_a" key="id" map="" class="au"/>      <!--    -->
    </links>
    <permacrud xmlns="http://open-ils.org/spec/opensrf/IDL/permacrud/v1">  <!--    -->
        <actions>
            <retrieve permission="VIEW_USER">                               <!--    -->
                <context link="usr" field="home_ou"/>                       <!--    -->
            </retrieve>
        </actions>
    </permacrud>
</class>
```

The `<class>` element defines the class:

- The `id` attribute defines the *class hint* that identifies the class both elsewhere in the fieldmapper configuration file, such as in the value of the `field` attribute of the `<link>` element, and in the JSON object itself when it is instantiated. For example, an "Open User Summary" JSON object would have the top level property of `"__c":"mous"`.

- The `controller` attribute identifies the services that have direct access to this class. If `open-ils.pcrud` is not listed, for example, then there is no means to directly access members of this class through a public service.

- The `oils_obj:fieldmapper` attribute defines the name of the Perl fieldmapper class that will be dynamically generated to provide setter and getter methods for instances of the class.

- The `oils_persist:tablename` attribute identifies the schema name and table name of the database table that stores the data that represents the instances of this class. In this case, the schema is `money` and the table is `open_usr_summary`.

- The `reporter:label` attribute defines a human-readable name for the class used in the reporting interface to identify the class. These names are defined in English in the fieldmapper configuration file; however, they are extracted so that they can be translated and served in the user's language of choice.

The `<fields>` element lists all of the fields that belong to the object.

- The `oils_persist:primary` attribute identifies the field that acts as the primary key for the object; in this case, the field with the name `usr`.

- The `oils_persist:sequence` attribute identifies the sequence object (if any) in this database provides values for new instances of this class. In this case, the primary key is defined by a field that is linked to a different table, so no sequence is used to populate these instances.

Each `<field>` element defines a single field with the following attributes:

- The `name` attribute identifies the column name of the field in the underlying database table as well as providing a name for the setter / getter method that can be invoked in the JSON or native version of the object.

- The `reporter:datatype` attribute defines how the reporter should treat the contents of the field for the purposes of querying and display.

- The `reporter:label` attribute can be used to provide a human-readable name for each field; without it, the reporter falls back to the value of the `name` attribute.

The `<links>` element contains a set of zero or more `<link>` elements, each of which defines a relationship between the class being described and another class.

- The `field` attribute identifies the field named in this class that links to the external class.

- The `reltype` attribute identifies the kind of relationship between the classes; in the case of `has_a`, each value in the `usr` field is guaranteed to have a corresponding value in the external class.

- The `key` attribute identifies the name of the field in the external class to which this field links.

- The rarely-used `map` attribute identifies a second class to which the external class links; it enables this field to define a direct relationship to an external class with one degree of separation, to avoid having to retrieve all of the linked members of an intermediate class just to retrieve the instances from the actual desired target class.

- The `class` attribute identifies the external class to which this field links.

The `<permacrud>` element defines the permissions that must have been granted to a user to operate on instances of this class.

The `<retrieve>` element is one of four possible children of the `<actions>` element that define the permissions required for each action: create, retrieve, update, and delete.

- The `permission` attribute identifies the name of the permission that must have been granted to the user to perform the action.

- The `contextfield` attribute, if it exists, defines the field in this class that identifies the library within the system for which the user must have privileges to work. If a user has been granted a given permission, but has not been granted privileges to work at a given library, they can not perform the action at that library.

The rarely-used `<context>` element identifies a linked field (`link` attribute) in this class which links to an external class that holds the field (`field` attribute) that identifies the library within the system for which the user must have privileges to work.

When you retrieve an instance of a class, you can ask for the result to *flesh* some or all of the linked fields of that class, so that the linked instances are returned embedded directly in your requested instance. In that same request you can ask for the fleshed instances to in turn have their linked fields fleshed. By bundling all of this into a single request and result sequence, you can avoid the network overhead of requiring the client to request the base object, then request each linked object in turn.

You can also iterate over a collection of instances and set the automatically generated `isdeleted`, `isupdated`, or `isnew` properties to indicate that the given instance has been deleted, updated, or created respectively. Evergreen can then act in batch mode over the collection to perform the requested actions on any of the instances that have been flagged for action.

Returning streaming results

In the previous implementation of the `opensrf.simple-text.split` method, we returned a reference to the complete array of results. For small values being delivered over the network, this is perfectly acceptable, but for

large sets of values this can pose a number of problems for the requesting client. Consider a service that returns a set of bibliographic records in response to a query like "all records edited in the past month"; if the underlying database is relatively active, that could result in thousands of records being returned as a single network request. The client would be forced to block until all of the results are returned, likely resulting in a significant delay, and depending on the implementation, correspondingly large amounts of memory might be consumed as all of the results are read from the network in a single block.

OpenSRF offers a solution to this problem. If the method returns results that can be divided into separate meaningful units, you can register the OpenSRF method as a streaming method and enable the client to loop over the results one unit at a time until the method returns no further results. In addition to registering the method with the provided name, OpenSRF also registers an additional method with `.atomic` appended to the method name. The `.atomic` variant gathers all of the results into a single block to return to the client, giving the caller the ability to choose either streaming or atomic results from a single method definition.

In the following example, the text splitting method has been reimplemented to support streaming; very few changes are required:

Text splitting method - streaming mode.

```
sub text_split {
    my $self = shift;
    my $conn = shift;
    my $text = shift;
    my $delimiter = shift || ' ';

    my @split_text = split $delimiter, $text;
    foreach my $string (@split_text) {           #
        $conn->respond($string);
    }
    return undef;
}

__PACKAGE__->register_method(
    method    => 'text_split',
    api_name  => 'opensrf.simple-text.split',
    stream    => 1                               #
);
```

Rather than returning a reference to the array, a streaming method loops over the contents of the array and invokes the `respond()` method of the connection object on each element of the array.

Registering the method as a streaming method instructs OpenSRF to also register an atomic variant (`opensrf.simple-text.split.atomic`).

Error! Warning! Info! Debug!

As hard as it may be to believe, it is true: applications sometimes do not behave in the expected manner, particularly when they are still under development. The server language bindings for OpenSRF include integrated support for logging messages at the levels of ERROR, WARNING, INFO, DEBUG, and the extremely verbose INTERNAL to either a local file or to a syslogger service. The destination of the log files, and the level of verbosity to be logged, is set in the `opensrf_core.xml` configuration file. To add logging to our Perl example, we just have to add the `OpenSRF::Utils::Logger` package to our list of used Perl modules, then invoke the logger at the desired logging level.

You can include many calls to the OpenSRF logger; only those that are higher than your configured logging level will actually hit the log. The following example exercises all of the available logging levels in OpenSRF:

```
use OpenSRF::Utils::Logger;
my $logger = OpenSRF::Utils::Logger;
# some code in some function
{
    $logger->error("Hmm, something bad DEFINITELY happened!");
    $logger->warn("Hmm, something bad might have happened.");
    $logger->info("Something happened.");
    $logger->debug("Something happened; here are some more details.");
    $logger->internal("Something happened; here are all the gory details.")
}
```

If you call the mythical OpenSRF method containing the preceding OpenSRF logger statements on a system running at the default logging level of INFO, you will only see the INFO, WARN, and ERR messages, as follows:

Results of logging calls at the default level of INFO.

```
[2010-03-17 22:27:30] opensrf.simple-text [ERR :5681:SimpleText.pm:277:] Hmm, something bad DEFINITELY happened!
[2010-03-17 22:27:30] opensrf.simple-text [WARN:5681:SimpleText.pm:278:] Hmm, something bad might have happened.
[2010-03-17 22:27:30] opensrf.simple-text [INFO:5681:SimpleText.pm:279:] Something happened.
```

If you then increase the the logging level to INTERNAL (5), the logs will contain much more information, as follows:

Results of logging calls at the default level of INTERNAL.

```
[2010-03-17 22:48:11] opensrf.simple-text [ERR :5934:SimpleText.pm:277:] Hmm, something bad DEFINITELY happened!
[2010-03-17 22:48:11] opensrf.simple-text [WARN:5934:SimpleText.pm:278:] Hmm, something bad might have happened.
[2010-03-17 22:48:11] opensrf.simple-text [INFO:5934:SimpleText.pm:279:] Something happened.
[2010-03-17 22:48:11] opensrf.simple-text [DEBG:5934:SimpleText.pm:280:] Something happened; here are some more deta
[2010-03-17 22:48:11] opensrf.simple-text [INTL:5934:SimpleText.pm:281:] Something happened; here are all the gory d
[2010-03-17 22:48:11] opensrf.simple-text [ERR :5934:SimpleText.pm:283:] Resolver did not find a cache hit
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:Cache.pm:125:] Stored opensrf.simple-text.test_cache.masaa => "
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:Application.pm:579:] Coderef for [OpenSRF::Application::Demo::S
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:Application.pm:586:] A top level Request object is responding d
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:Application.pm:190:] Method duration for [opensrf.simple-text.t
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:AppSession.pm:780:] Calling queue_wait(0)
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:AppSession.pm:769:] Resending...0
[2010-03-17 22:48:21] opensrf.simple-text [INTL:5934:AppSession.pm:450:] In send
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:AppSession.pm:506:] AppSession sending RESULT to opensrf@privat
[2010-03-17 22:48:21] opensrf.simple-text [DEBG:5934:AppSession.pm:506:] AppSession sending STATUS to opensrf@privat
...
```

To see everything that is happening in OpenSRF, try leaving your logging level set to INTERNAL for a few minutes - just ensure that you have a lot of free disk space available if you have a moderately busy system!

Caching results: one secret of scalability

If you have ever used an application that depends on a remote Web service outside of your control-say, if you need to retrieve results from a microblogging service-you know the pain of latency and dependability (or the lack thereof). To improve response time in OpenSRF applications, you can take advantage of the support offered by the `OpenSRF::Utils::Cache` module for communicating with a local instance or cluster of memcache daemons to store and retrieve persistent values.

```
use OpenSRF::Utils::Cache;                                          #
sub test_cache {
    my $self = shift;
    my $conn = shift;
    my $test_key = shift;
    my $cache = OpenSRF::Utils::Cache->new('global');               #
    my $cache_key = "opensrf.simple-text.test_cache.$test_key";     #
    my $result = $cache->get_cache($cache_key) || undef;            #
    if ($result) {
        $logger->info("Resolver found a cache hit");
        return $result;
    }
    sleep 10;                                                       #
    my $cache_timeout = 300;                                        #
    $cache->put_cache($cache_key, "here", $cache_timeout);          #
    return "There was no cache hit.";
}
```

This example:

Imports the OpenSRF::Utils::Cache module
Creates a cache object
Creates a unique cache key based on the OpenSRF method name and request input value
Checks to see if the cache key already exists; if so, it immediately returns that value
If the cache key does not exist, the code sleeps for 10 seconds to simulate a call to a slow remote Web service, or an intensive process
Sets a value for the lifetime of the cache key in seconds
When the code has retrieved its value, then it can create the cache entry, with the cache key, value to be stored ("here"), and the timeout value in seconds to ensure that we do not return stale data on subsequent calls

Initializing the service and its children: child labour

When an OpenSRF service is started, it looks for a procedure called `initialize()` to set up any global variables shared by all of the children of the service. The `initialize()` procedure is typically used to retrieve configuration settings from the `opensrf.xml` file.

An OpenSRF service spawns one or more children to actually do the work requested by callers of the service. For every child process an OpenSRF service spawns, the child process clones the parent environment and then each child process runs the `child_init()` process (if any) defined in the OpenSRF service to initialize any child-specific settings.

When the OpenSRF service kills a child process, it invokes the `child_exit()` procedure (if any) to clean up any resources associated with the child process. Similarly, when the OpenSRF service is stopped, it calls the `DESTROY()` procedure to clean up any remaining resources.

Retrieving configuration settings

The settings for OpenSRF services are maintained in the `opensrf.xml` XML configuration file. The structure of the XML document consists of a root element `<opensrf>` containing two child elements:

- `<default>` contains an `<apps>` element describing all OpenSRF services running on this system — see the section called “Registering a service with the OpenSRF configuration files” --, as well as any other arbitrary XML descriptions required for global configuration purposes. For example, Evergreen uses this section for email notification and inter-library patron privacy settings.

- `<hosts>` contains one element per host that participates in this OpenSRF system. Each host element must include an `<activeapps>` element that lists all of the services to start on this host when the system starts up. Each host element can optionally override any of the default settings.

OpenSRF includes a service named `opensrf.settings` to provide distributed cached access to the configuration settings with a simple API:

- `opensrf.settings.default_config.get`: accepts zero arguments and returns the complete set of default settings as a JSON document
- `opensrf.settings.host_config.get`: accepts one argument (hostname) and returns the complete set of settings, as customized for that hostname, as a JSON document
- `opensrf.settings.xpath.get`: accepts one argument (an XPath expression) and returns the portion of the configuration file that matches the expression as a JSON document

For example, to determine whether an Evergreen system uses the opt-in support for sharing patron information between libraries, you could either invoke the `opensrf.settings.default_config.get` method and parse the JSON document to determine the value, or invoke the `opensrf.settings.xpath.get` method with the XPath `/opensrf/default/share/user/opt_in` argument to retrieve the value directly.

In practice, OpenSRF includes convenience libraries in all of its client language bindings to simplify access to configuration values. C offers osrfConfig.c, Perl offers `OpenSRF::Utils::SettingsClient`, Java offers `org.opensrf.util.SettingsClient`, and Python offers `osrf.set`. These libraries locally cache the configuration file to avoid network roundtrips for every request and enable the developer to request specific values without having to manually construct XPath expressions.

Getting under the covers with OpenSRF

Now that you have seen that it truly is easy to create an OpenSRF service, we can take a look at what is going on under the covers to make all of this work for you.

Get on the messaging bus - safely

One of the core innovations of OpenSRF was to use the Extensible Messaging and Presence Protocol (XMPP, more colloquially known as Jabber) as the messaging bus that ties OpenSRF services together across servers. XMPP is an "XML protocol for near-real-time messaging, presence, and request-response services" (http://www.ietf.org/rfc/rfc3920.txt) that OpenSRF relies on to handle most of the complexity of networked communications. OpenSRF achieves a measure of security for its services through the use of public and private XMPP domains; all OpenSRF services automatically register themselves with the private XMPP domain, but only those services that register themselves with the public XMPP domain can be invoked from public OpenSRF clients.

In a minimal OpenSRF deployment, two XMPP users named "router" connect to the XMPP server, with one connected to the private XMPP domain and one connected to the public XMPP domain. Similarly, two XMPP users named "opensrf" connect to the XMPP server via the private and public XMPP domains. When an OpenSRF service is started, it uses the "opensrf" XMPP user to advertise its availability with the corresponding router on that XMPP domain; the XMPP server automatically assigns a Jabber ID (JID) based on the client hostname to each service's listener process and each connected drone process waiting to carry out requests. When an OpenSRF router receives a request to invoke a method on a given service, it connects the requester to the next available listener in the list of registered listeners for that service.

The opensrf and router user names, passwords, and domain names, along with the list of services that should be public, are contained in the `opensrf_core.xml` configuration file.

Message body format

OpenSRF was an early adopter of JavaScript Object Notation (JSON). While XMPP is an XML protocol, the Evergreen developers recognized that the compactness of the JSON format offered a significant reduction in bandwidth for the volume of messages that would be generated in an application of that size. In addition, the ability of languages such as JavaScript, Perl, and Python to generate native objects with minimal parsing offered an attractive advantage over invoking an XML parser for every message. Instead, the body of the XMPP message is a simple JSON structure. For a simple request, like the following example that simply reverses a string, it looks like a significant overhead: but we get the advantages of locale support and tracing the request from the requester through the listener and responder (drone).

A request for opensrf.simple-text.reverse("foobar"):

```
<message from='router@private.localhost/opensrf.simple-text'
  to='opensrf@private.localhost/opensrf.simple-text_listener_at_localhost_6275'
  router_from='opensrf@private.localhost/_karmic_126678.3719_6288'
  router_to='' router_class='' router_command='' osrf_xid=''
>
  <thread>1266781414.366573.12667814146288</thread>
  <body>
[
  {"__c":"osrfMessage","__p":
    {"threadTrace":"1","locale":"en-US","type":"REQUEST","payload":
      {"__c":"osrfMethod","__p":
        {"method":"opensrf.simple-text.reverse","params":["foobar"]}
      }
    }
  }
]
  </body>
</message>
```

A response from opensrf.simple-text.reverse("foobar").

```
<message from='opensrf@private.localhost/opensrf.simple-text_drone_at_localhost_6285'
  to='opensrf@private.localhost/_karmic_126678.3719_6288'
  router_command='' router_class='' osrf_xid=''
>
  <thread>1266781414.366573.12667814146288</thread>
  <body>
[
  {"__c":"osrfMessage","__p":
    {"threadTrace":"1","payload":
      {"__c":"osrfResult","__p":
        {"status":"OK","content":"raboof","statusCode":200}
      } ,"type":"RESULT","locale":"en-US"}
  },
  {"__c":"osrfMessage","__p":
    {"threadTrace":"1","payload":
      {"__c":"osrfConnectStatus","__p":
        {"status":"Request Complete","statusCode":205}
      },"type":"STATUS","locale":"en-US"}
  }
]
  </body>
</message>
```

The content of the `<body>` element of the OpenSRF request and result should look familiar; they match the structure of the OpenSRF over HTTP examples that we previously dissected.

Registering OpenSRF methods in depth

Let's explore the call to `__PACKAGE__->register_method()`; most of the elements of the hash are optional, and for the sake of brevity we omitted them in the previous example. As we have seen in the results of the introspection call, a verbose registration method call is recommended to better enable the internal documentation. So, for the sake of completeness here, is the set of elements that you should pass to `__PACKAGE__->register_method()`:

- `method`: the name of the procedure in this module that is being registered as an OpenSRF method
- `api_name`: the invocable name of the OpenSRF method; by convention, the OpenSRF service name is used as the prefix
- `api_level`: (optional) can be used for versioning the methods to allow the use of a deprecated API, but in practical use is always 1
- `argc`: (optional) the minimal number of arguments that the method expects
- `stream`: (optional) if this argument is set to any value, then the method supports returning multiple values from a single call to subsequent requests, and OpenSRF automatically creates a corresponding method with ".atomic" appended to its name that returns the complete set of results in a single request; streaming methods are useful if you are returning hundreds of records and want to act on the results as they return
- `signature`: (optional) a hash describing the method's purpose, arguments, and return value
  - `desc`: a description of the method's purpose
  - `params`: an array of hashes, each of which describes one of the method arguments
    - `name`: the name of the argument
    - `desc`: a description of the argument's purpose
    - `type`: the data type of the argument: for example, string, integer, boolean, number, array, or hash
  - `return`: a hash describing the return value of the method
    - `desc`: a description of the return value
    - `type`: the data type of the return value: for example, string, integer, boolean, number, array, or hash

Evergreen-specific OpenSRF services

Evergreen is currently the primary showcase for the use of OpenSRF as an application architecture. Evergreen 2.6.0 includes the following set of OpenSRF services:

- `open-ils.acq` Supports tasks for managing the acquisitions process
- `open-ils.actor`: Supports common tasks for working with user accounts and libraries.
- `open-ils.auth`: Supports authentication of Evergreen users.

- `open-ils.auth_proxy`: Support using external services such as LDAP directories to authenticate Evergreen users
- `open-ils.cat`: Supports common cataloging tasks, such as creating, modifying, and merging bibliographic and authority records.
- `open-ils.circ`: Supports circulation tasks such as checking out items and calculating due dates.
- `open-ils.collections`: Supports tasks to assist collections services for contacting users with outstanding fines above a certain threshold.
- `open-ils.cstore`: Supports unrestricted access to Evergreen fieldmapper objects. This is a private service.
- `open-ils.fielder`
- `open-ils.justintime`: Support tasks for determining if an action/trigger event is still valid
- `open-ils.pcrud`: Supports access to Evergreen fieldmapper objects, restricted by staff user permissions. This is a private service. objects.
- `open-ils.permacrud`: Supports access to Evergreen fieldmapper objects, restricted by staff user permissions. This is a private service.
- `open-ils.reporter`: Supports the creation and scheduling of reports.
- `open-ils.reporter-store`: Supports access to Evergreen fieldmapper objects for the reporting service. This is a private service.
- `open-ils.resolver` Support tasks for integrating with an OpenURL resolver.
- `open-ils.search`: Supports searching across bibliographic records, authority records, serial records, Z39.50 sources, and ZIP codes.
- `open-ils.serial`: Support tasks for serials management
- `open-ils.storage`: A deprecated method of providing access to Evergreen fieldmapper objects. Implemented in Perl, this service has largely been replaced by the much faster C-based `open-ils.cstore` service.
- `open-ils.supercat`: Supports transforms of MARC records into other formats, such as MODS, as well as providing Atom and RSS feeds and SRU access.
- `open-ils.trigger`: Supports event-based triggers for actions such as overdue and holds available notification emails.
- `open-ils.url_verify`: Support tasks for validating URLs
- `open-ils.vandelay`: Supports the import and export of batches of bibliographic and authority records.
- `opensrf.settings`: Supports communicating opensrf.xml settings to other services.

Of some interest is that the `open-ils.reporter-store` and `open-ils.cstore` services have identical implementations. Surfacing them as separate services enables a deployer of Evergreen to ensure that the reporting

service does not interfere with the performance-critical `open-ils.cstore` service. One can also direct the reporting service to a read-only database replica to, again, avoid interference with `open-ils.cstore` which must write to the master database.

There are only a few significant services that are not built on OpenSRF, such as the SIP and Z39.50 servers. These services implement different protocols and build on existing daemon architectures (Simple2ZOOM for Z39.50), but still rely on the other OpenSRF services to provide access to the Evergreen data. The non-OpenSRF services are reasonably self-contained and can be deployed on different servers to deliver the same sort of deployment flexibility as OpenSRF services, but have the disadvantage of not being integrated into the same configuration and control infrastructure as the OpenSRF services.

Evergreen after one year: reflections on OpenSRF

Project Conifer has been live on Evergreen for just over a year now, and as one of the primary technologists I have had to work closely with the OpenSRF infrastructure during that time. As such, I am in a position to identify some of the strengths and weaknesses of OpenSRF based on our experiences.

Strengths of OpenSRF

As a service infrastructure, OpenSRF has been remarkably reliable. We initially deployed Evergreen on an unreleased version of both OpenSRF and Evergreen due to our requirements for some functionality that had not been delivered in a stable release at that point in time, and despite this risky move we suffered very little unplanned downtime in the opening months. On July 27, 2009 we moved to a newer (but still unreleased) version of the OpenSRF and Evergreen code, and began formally tracking our downtime. Since then, we have achieved more than 99.9% availability - including scheduled downtime for maintenance. This compares quite favourably to the maximum of 75% availability that we were capable of achieving on our previous library system due to the nightly downtime that was required for our backup process. The OpenSRF "maximum request" configuration parameter for each service that kills off drone processes after they have served a given number of requests provides a nice failsafe for processes that might otherwise suffer from a memory leak or hung process. It also helps that when we need to apply an update to a Perl service that is running on multiple servers, we can apply the updated code, then restart the service on one server at a time to avoid any downtime.

As promised by the OpenSRF infrastructure, we have also been able to tune our cluster of servers to provide better performance. For example, we were able to change the number of maximum concurrent processes for our database services when we noticed that we seeing a performance bottleneck with database access. Making a configuration change go live simply requires you to restart the `opensrf.setting` service to pick up the configuration change, then restart the affected service on each of your servers. We were also able to turn off some of the less-used OpenSRF services, such as `open-ils.collections`, on one of our servers to devote more resources on that server to the more frequently used services and other performance-critical processes such as Apache.

The support for logging and caching that is built into OpenSRF has been particularly helpful with the development of a custom service for SFX holdings integration into our catalogue. Once I understood how OpenSRF works, most of the effort required to build that SFX integration service was spent on figuring out how to properly invoke the SFX API to display human-readable holdings. Adding a new OpenSRF service and registering several new methods for the service was relatively easy. The support for directing log messages to syslog in OpenSRF has also been a boon for both development and debugging when problems arise in a cluster of five servers; we direct all of our log messages to a single server where we can inspect the complete set of messages for the entire cluster in context, rather than trying to piece them together across servers.

Weaknesses

The primary weakness of OpenSRF is the lack of either formal or informal documentation for OpenSRF. There are many frequently asked questions on the Evergreen mailing lists and IRC channel that indicate that some of the people running Evergreen or trying to run Evergreen have not been able to find documentation to help them understand, even at a high level, how the OpenSRF Router and services work with XMPP and the Apache Web server to provide a working Evergreen system. Also, over the past few years several developers have indicated an interest in developing Ruby and PHP bindings for OpenSRF, but the efforts so far have resulted in no working code. Without a formal specification, clearly annotated examples, and unit tests for the major OpenSRF communication use cases that could be ported to the new language as a base set of expectations for a working binding, the hurdles for a developer new to OpenSRF are significant. As a result, Evergreen integration efforts with popular frameworks like Drupal, Blacklight, and VuFind result in the best practical option for a developer with limited time — database-level integration — which has the unfortunate side effect of being much more likely to break after an upgrade.

In conjunction with the lack of documentation that makes it hard to get started with the framework, a disincentive for new developers to contribute to OpenSRF itself is the lack of integrated unit tests. For a developer to contribute a significant, non-obvious patch to OpenSRF, they need to manually run through various (undocumented, again) use cases to try and ensure that the patch introduced no unanticipated side effects. The same problems hold for Evergreen itself, although the Constrictor stress-testing framework offers a way of performing some automated system testing and performance testing.

These weaknesses could be relatively easily overcome with the effort through contributions from people with the right skill sets. This article arguably offers a small set of clear examples at both the networking and application layer of OpenSRF. A technical writer who understands OpenSRF could contribute a formal specification to the project. With a formal specification at their disposal, a quality assurance expert could create an automated test harness and a basic set of unit tests that could be incrementally extended to provide more coverage over time. If one or more continual integration environments are set up to track the various OpenSRF branches of interest, then the OpenSRF community would have immediate feedback on build quality. Once a unit testing framework is in place, more developers might be willing to develop and contribute patches as they could sanity check their own code without an intense effort before exposing it to their peers.

Summary

In this article, I attempted to provide both a high-level and detailed overview of how OpenSRF works, how to build and deploy new OpenSRF services, how to make requests to OpenSRF method from OpenSRF clients or over HTTP, and why you should consider it a possible infrastructure for building your next high-performance system that requires the capability to scale out. In addition, I surveyed the Evergreen services built on OpenSRF and reflected on the strengths and weaknesses of the platform based on the experiences of Project Conifer after a year in production, with some thoughts about areas where the right application of skills could make a significant difference to the Evergreen and OpenSRF projects.

Appendix: Python client

Following is a Python client that makes the same OpenSRF calls as the Perl client:

```
#!/usr/bin/env python
"""OpenSRF client example in Python"""
import osrf.system
import osrf.ses

def osrf_substring(session, text, sub):
    """substring: Accepts a string and a number as input, returns a string"""
    request = session.request('opensrf.simple-text.substring', text, sub)

    # Retrieve the response from the method
    # The timeout parameter is optional
    response = request.recv(timeout=2)

    request.cleanup()
    # The results are accessible via content()
    return response.content()

def osrf_split(session, text, delim):
    """split: Accepts two strings as input, returns an array of strings"""
    request = session.request('opensrf.simple-text.split', text, delim)
    response = request.recv()
    request.cleanup()
    return response.content()

def osrf_statistics(session, strings):
    """statistics: Accepts an array of strings as input, returns a hash"""
    request = session.request('opensrf.simple-text.statistics', strings)
    response = request.recv()
    request.cleanup()
    return response.content()


if __name__ == "__main__":
    file = '/openils/conf/opensrf_core.xml'

    # Pull connection settings from <config><opensrf> section of opensrf_core.xml
    osrf.system.System.connect(config_file=file, config_context='config.opensrf')

    # Set up a connection to the opensrf.settings service
    session = osrf.ses.ClientSession('opensrf.simple-text')

    result = osrf_substring(session, "foobar", 3)
    print(result)
    print

    result = osrf_split(session, "This is a test", " ")
    print("Received %d elements: [" % len(result)),
    print(', '.join(result)), ']'

    many_strings = (
        "First I think I'll have breakfast",
        "Then I think that lunch would be nice",
        "And then seventy desserts to finish off the day"
    )
    result = osrf_statistics(session, many_strings)
    print("Length: %d" % result["length"])
    print("Word count: %d" % result["word_count"])

    # Cleanup connection resources
    session.cleanup()
```

Python's `dnspython` module refuses to read `/etc/resolv.conf`, so to access hostnames that are not served up via DNS, such as the extremely common case of `localhost`, you may need to install a package like `dnsmasq` to act as a local DNS server for those hostnames.

Chapter 9. Support Scripts

Various scripts are included with Evergreen in the `/openils/bin/` directory (and in the source code in `Open-ILS/src/support-scripts` and `Open-ILS/src/extras`). Some of them are used during the installation process, such as `eg_db_config`, while others are usually run as cron jobs for routine maintenance, such as `fine_generator.pl` and `hold_targeter.pl`. Others are useful for less frequent needs, such as the scripts for importing/exporting MARC records. You may explore these scripts and adapt them for your local needs. You are also welcome to share your improvements or ask any questions on the Evergreen IRC channel or email lists.

Here is a summary of the most commonly used scripts. The script name links to more thorough documentation, if available.

- action_trigger_runner.pl — Useful for creating events for specified hooks and running pending events
- authority_authority_linker.pl — Links reference headings in authority records to main entry headings in other authority records. Should be run at least once a day (only for changed records).
- authority_control_fields.pl — Links bibliographic records to the best matching authority record. Should be run at least once a day (only for changed records). You can accomplish this by running *authority_control_fields.pl --days-back=1*
- autogen.sh — Generates web files used by the OPAC, especially files related to organization unit hierarchy, fieldmapper IDL, locales selection, facet definitions, compressed JS files and related cache key
- clark-kent.pl — Used to start and stop the reporter (which runs scheduled reports)
- eg_db_config — Creates database and schema, updates config files, sets Evergreen administrator username and password
- fine_generator.pl
- hold_targeter.pl
- marc2are.pl — Converts authority records from MARC format to Evergreen objects suitable for importing via pg_loader.pl (or parallel_pg_loader.pl)
- marc2bre.pl — Converts bibliographic records from MARC format to Evergreen objects suitable for importing via pg_loader.pl (or parallel_pg_loader.pl)
- marc2sre.pl — Converts serial records from MARC format to Evergreen objects suitable for importing via pg_loader.pl (or parallel_pg_loader.pl)
- marc_export — Exports authority, bibliographic, and serial holdings records into any of these formats: USMARC, UNIMARC, XML, BRE, ARE
- osrf_control — Used to start, stop and send signals to OpenSRF services
- parallel_pg_loader.pl — Uses the output of marc2bre.pl (or similar tools) to generate the SQL for importing records into Evergreen in a parallel fashion

authority_control_fields: Connecting Bibliographic and Authority records

This script matches headings in bibliographic records to the appropriate authority records. When it finds a match, it will add a subfield 0 to the matching bibliographic field.

Here is how the matching works:

Bibliographic field	Authority field it matches	Subfields that it examines
100	100	a,b,c,d,f,g,j,k,l,n,p,q,t,u
110	110	a,b,c,d,f,g,k,l,n,p,t,u
111	111	a,c,d,e,f,g,j,k,l,n,p,q,t,u
130	130	a,d,f,g,h,k,l,m,n,o,p,r,s,t
600	100	a,b,c,d,f,g,h,j,k,l,m,n,o,p,q,r,s,t,v,x,y,z
610	110	a,b,c,d,f,g,h,k,l,m,n,o,p,r,s,t,v,w,x,y,z
611	111	a,c,d,e,f,g,h,j,k,l,n,p,q,s,t,v,x,y,z
630	130	a,d,f,g,h,k,l,m,n,o,p,r,s,t,v,x,y,z
648	148	a,v,x,y,z
650	150	a,b,v,x,y,z
651	151	a,v,x,y,z
655	155	a,v,x,y,z
700	100	a,b,c,d,f,g,j,k,l,n,p,q,t,u
710	110	a,b,c,d,f,g,k,l,n,p,t,u
711	111	a,c,d,e,f,g,j,k,l,n,p,q,t,u
730	130	a,d,f,g,h,j,k,m,n,o,p,r,s,t
751	151	a,v,x,y,z
800	100	a,b,c,d,e,f,g,j,k,l,n,p,q,t,u,4
830	130	a,d,f,g,h,k,l,m,n,o,p,r,s,t

marc_export: Exporting Bibliographic Records into MARC files

The following procedure explains how to export Evergreen bibliographic records into MARC files using the **marc_export** support script. All steps should be performed by the `opensrf` user from your Evergreen server.

Processing time for exporting records depends on several factors such as the number of records you are exporting. It is recommended that you divide the export ID files (records.txt) into a manageable number of records if you are exporting a large number of records.

1. Create a text file list of the Bibliographic record IDs you would like to export from Evergreen. One way to do this is using SQL:

```
SELECT DISTINCT bre.id FROM biblio.record_entry AS bre
    JOIN asset.call_number AS acn ON acn.record = bre.id
    WHERE bre.deleted='false' and owning_lib=101 \g /home/opensrf/records.txt;
```

   This query creates a file called `records.txt` containing a column of distinct IDs of items owned by the organizational unit with the id 101.

2. Navigate to the support-scripts folder

```
cd /home/opensrf/Evergreen-ILS*/Open-ILS/src/support-scripts/
```

3. Run **marc_export**, using the ID file you created in step 1 to define which files to export. The following example exports the records into MARCXML format.

```
cat /home/opensrf/records.txt | ./marc_export --store -i -c /openils/conf/opensrf_core.xml \
    -x /openils/conf/fm_IDL.xml -f XML --timeout 5 > exported_files.xml
```

`marc_export` does not output progress as it executes.

Options

The **marc_export** support script includes several options. You can find a complete list by running `./marc_export -h`. A few key options are also listed below:

--descendants and --library

The `marc_export` script has two related options, `--descendants` and `--library`. Both options take one argument of an organizational unit

The `--library` option will export records with holdings at the specified organizational unit only. By default, this only includes physical holdings, not electronic ones (also known as located URIs).

The `descendants` option works much like the `--library` option except that it is aware of the org. tree and will export records with holdings at the specified organizational unit and all of its descendants. This is handy if you want to export the records for all of the branches of a system. You can do that by specifying this option and the system's shortname, instead of specifying multiple `--library` options for each branch.

Both the `--library` and `--descendants` options can be repeated. All of the specified org. units and their descendants will be included in the output. You can also combine `--library` and `--descendants` options when necessary.

--items

The `--items` option will add an 852 field for every relevant item to the MARC record. This 852 field includes the following information:

Subfield	Contents
$b (occurrence 1)	Call number owning library shortname
$b (occurrence 2)	Item circulating library shortname
$c	Shelving location
$g	Circulation modifier
$j	Call number
$k	Call number prefix
$m	Call number suffix
$p	Barcode
$s	Status
$t	Copy number
$x	Miscellaneous item information
$y	Price

--since

You can use the `--since` option to export records modified after a certain date and time.

--store

By default, marc_export will use the reporter storage service, which should work in most cases. But if you have a separate reporter database and you know you want to talk directly to your main production database, then you can set the `--store` option to `cstore` or `storage`.

--uris

The `--uris` option (short form: `-u`) allows you to export records with located URIs (i.e. electronic resources). When used by itself, it will export only records that have located URIs. When used in conjunction with `--items`, it will add records with located URIs but no items/copies to the output. If combined with a `--library` or `--descendants` option, this option will limit its output to those records with URIs at the designated libraries. The best way to use this option is in combination with the `--items` and one of the `--library` or `--descendants` options to export **all** of a library's holdings both physical and electronic.

Parallel Ingest with pingest.pl

A program named pingest.pl allows fast bibliographic record ingest. It performs ingest in parallel so that multiple batches can be done simultaneously. It operates by splitting the records to be ingested up into batches and running all of the ingest methods on each batch. You may pass in options to control how many batches are run at the same time, how many records there are per batch, and which ingest operations to skip.

The browse ingest is presently done in a single process over all of the input records as it cannot run in parallel with itself. It does, however, run in parallel with the other ingests.

Command Line Options

pingest.pl accepts the following command line options:

--host	The server where PostgreSQL runs (either host name or IP address). The default is read from the PGHOST environment variable or "localhost."
--port	The port that PostgreSQL listens to on host. The default is read from the PGPORT environment variable or 5432.
--db	The database to connect to on the host. The default is read from the PGDATABASE environment variable or "evergreen."
--user	The username for database connections. The default is read from the PGUSER environment variable or "evergreen."
--password	The password for database connections. The default is read from the PGPASSWORD environment variable or "evergreen."
--batch-size	Number of records to process per batch. The default is 10,000.
--max-child	Max number of worker processes (i.e. the number of batches to process simultaneously). The default is 8.
--skip-browse , --skip-attrs , --skip-search , --skip-facets , --skip-display	Skip the selected reingest component.

--attr	This option allows the user to specify which record attributes to reingest. It can be used one or more times to specify one or more attributes to ingest. It can be omitted to reingest all record attributes. This option is ignored if the `--skip-attrs` option is used.
	
	The `--attr` option is most useful after doing something specific that requires only a partial ingest of records. For instance, if you add a new language to the `config.coded_value_map` table, you will want to reingest the `item_lang` attribute on all of your records. The following command line will do that, and only that, ingest:
	
	`$ /openils/bin/pingest.pl --skip-browse --skip-search --skip-facets \`
	`  --skip-display --attr=item_lang`

Importing Authority Records from Command Line

The major advantages of the command line approach are its speed and its convenience for system administrators who can perform bulk loads of authority records in a controlled environment. For alternate instructions, see the cataloging manual.

1. Run **marc2are.pl** against the authority records, specifying the user name, password, MARC type (USMARC or XML). Use `STDOUT` redirection to either pipe the output directly into the next command or into an output file for inspection. For example, to process a file with authority records in MARCXML format named `auth_small.xml` using the default user name and password, and directing the output into a file named `auth.are`:

```
cd Open-ILS/src/extras/import/
perl marc2are.pl --user admin --pass open-ils --marctype XML auth_small.xml > auth.are
```

The MARC type will default to USMARC if the `--marctype` option is not specified.

2. Run **parallel_pg_loader.pl** to generate the SQL necessary for importing the authority records into your system. This script will create files in your current directory with filenames like `pg_loader-output.are.sql` and `pg_loader-output.sql` (which runs the previous SQL file). To continue with the previous example by processing our new `auth.are` file:

```
cd Open-ILS/src/extras/import/
perl parallel_pg_loader.pl --auto are --order are auth.are
```

To save time for very large batches of records, you could simply pipe the output of **marc2are.pl** directly into **parallel_pg_loader.pl**.

3. Load the authority records from the SQL file that you generated in the last step into your Evergreen database using the psql tool. Assuming the default user name, host name, and database name for an Evergreen instance, that command looks like:

```
psql -U evergreen -h localhost -d evergreen -f pg_loader-output.sql
```

Juvenile-to-adult batch script

The batch `juv_to_adult.srfsh` script is responsible for toggling a patron from juvenile to adult. It should be set up as a cron job.

This script changes patrons to adult when they reach the age value set in the library setting named "Juvenile Age Threshold" (`global.juvenile_age_threshold`). When no library setting value is present at a given patron's home library, the value passed in to the script will be used as a default.

MARC Stream Importer

The MARC Stream Importer can import authority records or bibliographic records. A single running instance of the script can import either type of record, based on the record leader.

This support script has its own configuration file, *marc_stream_importer.conf*, which includes settings related to logs, ports, uses, and access control.

The importer is even more flexible than the staff client import, including the following options:

- *--bib-auto-overlay-exact* and *--auth-auto-overlay-exact*: overlay/merge on exact 901c matches

- *--bib-auto-overlay-1match* and *--auth-auto-overlay-1match*: overlay/merge when exactly one match is found
- *--bib-auto-overlay-best-match* and *--auth-auto-overlay-best-match*: overlay/merge on best match
- *--bib-import-no-match* and *--auth-import-no-match*: import when no match is found

One advantage to using this tool instead of the staff client Import interface is that the MARC Stream Importer can load a group of files at once.

Processing Action Triggers

To run action triggers, an Evergreen administrator will need to run the trigger processing script. This should be set up as a cron job to run periodically. To run the script, use this command:

```
/openils/bin/action_trigger_runner.pl --process-hooks --run-pending
```

You have several options when running the script:

- --run-pending: Run pending events to send emails or take other actions as specified by the reactor in the event definition.
- --process-hooks: Create hook events
- --osrf-config=[config_file]: OpenSRF core config file. Defaults to: /openils/conf/opensrf_core.xml
- --custom-filters=[filter_file]: File containing a JSON Object which describes any hooks that should use a user-defined filter to find their target objects. Defaults to: /openils/conf/action_trigger_filters.json
- --max-sleep=[seconds]: When in process-hooks mode, wait up to [seconds] for the lock file to go away. Defaults to 3600 (1 hour).
- --hooks=hook1[,hook2,hook3,…]: Define which hooks to create events for. If none are defined, it defaults to the list of hooks defined in the --custom-filters option. Requires --process-hooks.
- --granularity=[label]: Limit creating events and running pending events to those only with [label] granularity setting.
- --debug-stdout: Print server responses to STDOUT (as JSON) for debugging.
- --lock-file=[file_name]: Sets the lock file for the process.
- --verbose: Show details of script processing.
- --help: Show help information.

Examples:

- Run all pending events that have no granularity set. This is what you tell CRON to run at regular intervals.

  ```
  perl action_trigger_runner.pl --run-pending
  ```

- Batch create all "checkout.due" events

  ```
  perl action_trigger_runner.pl --hooks=checkout.due --process-hooks
  ```

- Batch create all events for a specific granularity and to send notices for all pending events with that same granularity.

```
perl action_trigger_runner.pl --run-pending --granularity=Hourly --process-hooks
```

Chapter 10. Daemons and services

Starting and Stopping the Reporter Daemon

Before you can view reports, the Evergreen administrator must start the reporter daemon from the command line of the Evergreen server.

The reporter daemon periodically checks for requests for new reports or scheduled reports and gets them running.

Starting the Reporter Daemon

To start the reporter daemon, run the following command as the opensrf user:

```
clark-kent.pl --daemon
```

You can also specify other options:

- **sleep=interval**: number of seconds to sleep between checks for new reports to run; defaults to 10
- **lockfile=filename**: where to place the lockfile for the process; defaults to /tmp/reporter-LOCK
- **concurrency=integer**: number of reporter daemon processes to run; defaults to 1
- **bootstrap=filename**: OpenSRF bootstrap configuration file; defaults to /openils/conf/opensrf_core.xml

The open-ils.reporter process must be running and enabled on the gateway before the reporter daemon can be started.

Remember that if the server is restarted, the reporter daemon will need to be restarted before you can view reports unless you have configured your server to start the daemon automatically at start up time.

Stopping the Reporter Daemon

To stop the reporter daemon, you have to kill the process and remove the lockfile. Assuming you're running just a single process and that the lockfile is in the default location, perform the following commands as the opensrf user:

```
kill `ps wax | grep "Clark Kent" | grep -v grep | cut -b1-6`
```

```
rm /tmp/reporter-LOCK
```

ebook_api service

The `open-ils.ebook_api` service looks up title and patron information from specified ebook vendor APIs.

The Evergreen catalog accesses data from this service through OpenSRF JS bindings.

The `OpenILS::Utils::HTTPClient` module is required for this service.

hold-targeter service

The `open-ils.hold-targeter` service is used to target holds.

QStore service

The QStore service is used by the user buckets feature in the Web client.

Chapter 11. Developing with pgTAP tests

Setting up pgTAP on your development server

Currently, Evergreen pgTAP tests expect a version of pgTAP (0.93) that is not yet available in the packages for most Linux distributions. Therefore, you will have to install pgTAP from source as follows:

1. Download, make, and install pgTAP on your database server. pgTAP can be downloaded from http://pgxn.org/dist/pgtap/ and the instructions for building and installing the extension are available from http://pgtap.org/documentation.html

2. Create the pgTAP extension in your Evergreen database. Using `psql`, connect to your Evergreen database and then issue the command:

   ```
   CREATE EXTENSION pgtap;
   ```

Running pgTAP tests

The pgTAP tests can be found in subdirectories of `Open-ILS/src/sql/Pg/` as follows:

- `t`: contains pgTAP unit tests that can be run on a freshly installed Evergreen database
- `live_t`: contains pgTAP unit tests meant to be run on an Evergreen database that also has had the "concerto" sample data loaded on it

To run the pgTAP unit and regression tests, use the `pg_prove` command. For example, from the Evergreen source directory, you can issue the command: `pg_prove -U evergreen Open-ILS/src/sql/Pg/t Open-ILS/src/sql/Pg/t/regress`

Part IV. System Configuration

Table of Contents

12. Describing your people 80
Setting the staff user's working location 80
Comparing approaches for managing permissions 81
Managing permissions in the staff client 81
Where to find existing permissions and what they mean 82
Where to find existing Permission Groups 82
Adding or removing permissions from a Permission Group 82
Managing role-based permission groups in the staff client 83
Secondary Group Permissions 83
Managing role-based permission groups in the database 86
Authentication Proxy 87
Patron Address City/State/County Pre-Populate by ZIP Code 89
Scoping and Permissions 89
Setup Steps 89
ZIP Code Data 91
Development 92
Apache Rewrite Tricks 92
Short URLs 92
Domain Based Content with RewriteMaps 92
Apache Access Handler Perl Module 94
Use Cases 95
Proxying Websites 96
13. Updating translations using Launchpad 97
Prerequisites 97
Updating the translations 97

Chapter 12. Describing your people

Many different members of your staff will use your Evergreen system to perform the wide variety of tasks required of the library.

When the Evergreen installation was completed, a number of permission groups should have been automatically created. These permission groups are:

- Users
- Patrons
- Staff
- Catalogers
- Circulators
- Acquisitions
- Acquisitions Administrator
- Cataloging Administrator
- Circulation Administrator
- Local Administrator
- Serials
- System Administrator
- Global Administrator
- Data Review
- Volunteers

Each of these permission groups has a different set of permissions connected to them that allow them to do different things with the Evergreen system. Some of the permissions are the same between groups; some are different. These permissions are typically tied to one or more working location (sometimes referred to as a working organizational unit or work OU) which affects where a particular user can exercise the permissions they have been granted.

Setting the staff user’s working location

To grant a working location to a staff user in the staff client:

1. Search for the patron. Select **Search > Search for Patrons** from the top menu.
2. When you retrieve the correct patron record, select **Other > User Permission Editor** from the upper right corner. The permissions associated with this account appear in the right side of the client, with the **Working Location** list at the top of the screen.

3. The **Working Location** list displays the Organizational Units in your consortium. Select the check box for each Organization Unit where this user needs working permissions. Clear any other check boxes for Organization Units where the user no longer requires working permissions.

4. Scroll all the way to the bottom of the page and click **Save**. This user account is now ready to be used at your library.

As you scroll down the page you will come to the **Permissions** list. These are the permissions that are given through the **Permission Group** that you assigned to this user. Depending on your own permissions, you may also have the ability to grant individual permissions directly to this user.

Comparing approaches for managing permissions

The Evergreen community uses two different approaches to deal with managing permissions for users:

- **Staff Client**

  Evergreen libraries that are most comfortable using the staff client tend to manage permissions by creating different profiles for each type of user. When you create a new user, the profile you assign to the user determines their basic set of permissions. This approach requires many permission groups that contain overlapping sets of permissions: for example, you might need to create a *Student Circulator* group and a *Student Cataloger* group. Then if a new employee needs to perform both of these roles, you need to create a third *Student Cataloger / Circulator* group representing the set of all of the permissions of the first two groups.

  The advantage to this approach is that you can maintain the permissions entirely within the staff client; a drawback to this approach is that it can be challenging to remember to add a new permission to all of the groups. Another drawback of this approach is that the user profile is also used to determine circulation and hold rules, so the complexity of your circulation and hold rules might increase significantly.

- **Database Access**

  Evergreen libraries that are comfortable manipulating the database directly tend to manage permissions by creating permission groups that reflect discrete roles within a library. At the database level, you can make a user belong to many different permission groups, and that can simplify your permission management efforts. For example, if you create a *Student Circulator* group and a *Student Cataloger* group, and a new employee needs to perform both of these roles, you can simply assign them to both of the groups; you do not need to create an entirely new permission group in this case. An advantage of this approach is that the user profile can represent only the user's borrowing category and requires only the basic *Patrons* permissions, which can simplify your circulation and hold rules.

Permissions and profiles are not carved in stone. As the system administrator, you can change them as needed. You may set and alter the permissions for each permission group in line with what your library, or possibly your consortium, defines as the appropriate needs for each function in the library.

Managing permissions in the staff client

In this section, we'll show you in the staff client:

- where to find the available permissions

- where to find the existing permission groups
- how to see the permissions associated with each group
- how to add or remove permissions from a group

We also provide an appendix with a listing of suggested minimum permissions for some essential groups. You can compare the existing permissions with these suggested permissions and, if any are missing, you will know how to add them.

Where to find existing permissions and what they mean

In the staff client, in the upper right corner of the screen, click on **Administration > Server Administration > Permissions**.

The list of available permissions will appear on screen and you can scroll down through them to see permissions that are already available in your default installation of Evergreen.

There are over 500 permissions in the permission list. They appear in two columns: **Code** and **Description**. Code is the name of the permission as it appear in the Evergreen database. Description is a brief note on what the permission allows. All of the most common permissions have easily understandable descriptions.

Where to find existing Permission Groups

In the staff client, in the upper right corner of the screen, navigate to **Administration > Server Administration > Permission Groups**.

Two panes will open on your screen. The left pane provides a tree view of existing Permission Groups. The right pane contains two tabs: Group Configuration and Group Permissions.

In the left pane, you will find a listing of the existing Permission Groups which were installed by default. Click on the + sign next to any folder to expand the tree and see the groups underneath it. You should see the Permission Groups that were listed at the beginning of this chapter. If you do not and you need them, you will have to create them.

Adding or removing permissions from a Permission Group

First, we will remove a permission from the Staff group.

1. From the list of Permission Groups, click on **Staff**.
2. In the right pane, click on the **Group Permissions** tab. You will now see a list of permissions that this group has.
3. From the list, choose **CREATE_CONTAINER**. This will now be highlighted.
4. Click the **Delete Selected** button. CREATE_CONTAINER will be deleted from the list. The system will not ask for a confirmation. If you delete something by accident, you will have to add it back.
5. Click the **Save Changes** button.

You can select a group of individual items by holding down the *Ctrl* key and clicking on them. You can select a list of items by clicking on the first item, holding down the *Shift* key, and clicking on the last item in the list that you want to select.

Now, we will add the permission we just removed back to the Staff group.

1. From the list of Permission Groups, click on **Staff**.
2. In the right pane, click on the **Group Permissions** tab.
3. Click on the **New Mapping** button. The permission mapping dialog box will appear.
4. From the Permission drop down list, choose **CREATE_CONTAINER**.
5. From the Depth drop down list, choose **Consortium**.
6. Click the checkbox for **Grantable**.
7. Click the **Add Mapping** button. The new permission will now appear in the Group Permissions window.
8. Click the **Save Changes** button.

If you have saved your changes and you don't see them, you may have to click the Reload button in the upper left side of the staff client screen.

Managing role-based permission groups in the staff client

Main permission groups are granted in the staff client through Edit in the patron record using the Main (Profile) Permission Group field. Additional permission groups can be granted using secondary permission groups.

Secondary Group Permissions

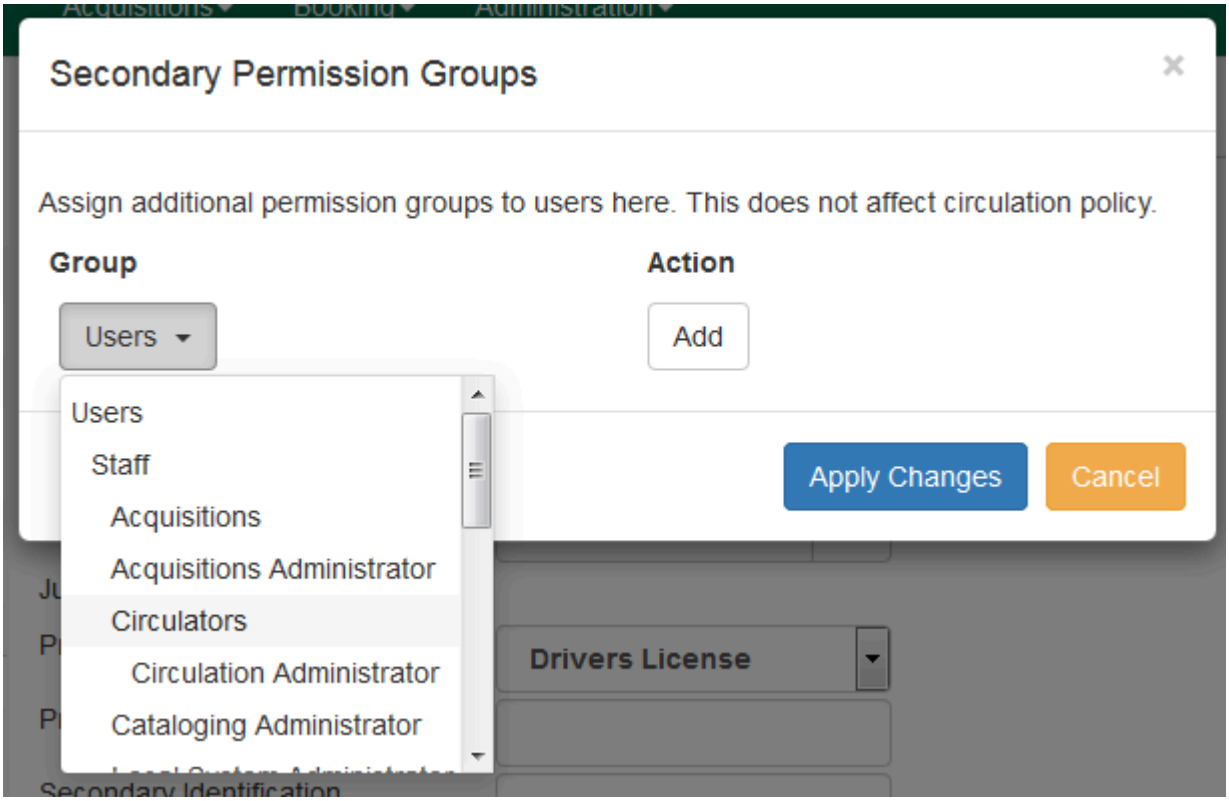
The *Secondary Groups* button functionality enables supplemental permission groups to be added to staff accounts. The **CREATE_USER_GROUP_LINK** and **REMOVE_USER_GROUP_LINK** permissions are required to display and use this feature.

In general when creating a secondary permission group do not grant the permission to login to Evergreen.

Granting Secondary Permissions Groups

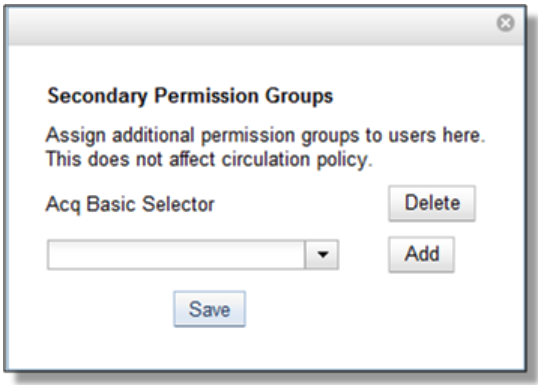
1. Open the account of the user you wish to grant secondary permission group to.
2. Click *Edit*.
3. Click *Secondary Groups*, located to the right of the *Main (Profile) Permission Group*.

4. From the dropdown menu select one of the secondary permission groups.



5. Click *Add*.

6. Click *Apply Changes*.



7. Click *Save* in the top right hand corner of the *Edit Screen* to save the user's account.

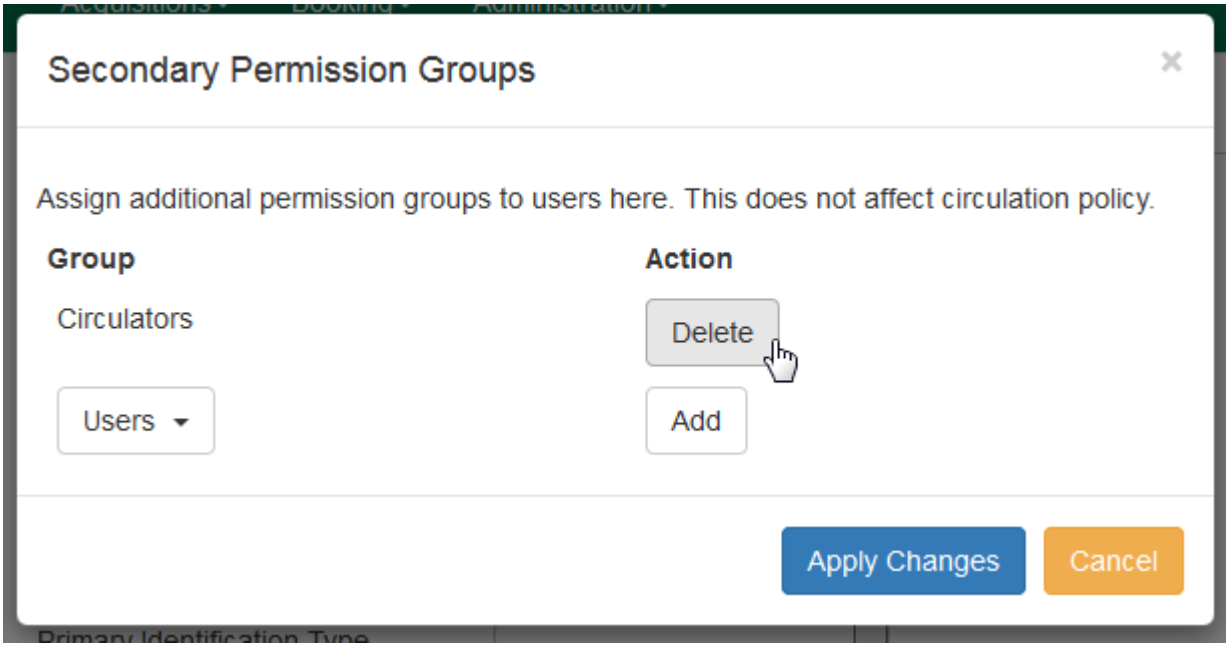
Removing Secondary Group Permissions

1. Open the account of the user you wish to remove the secondary permission group from.

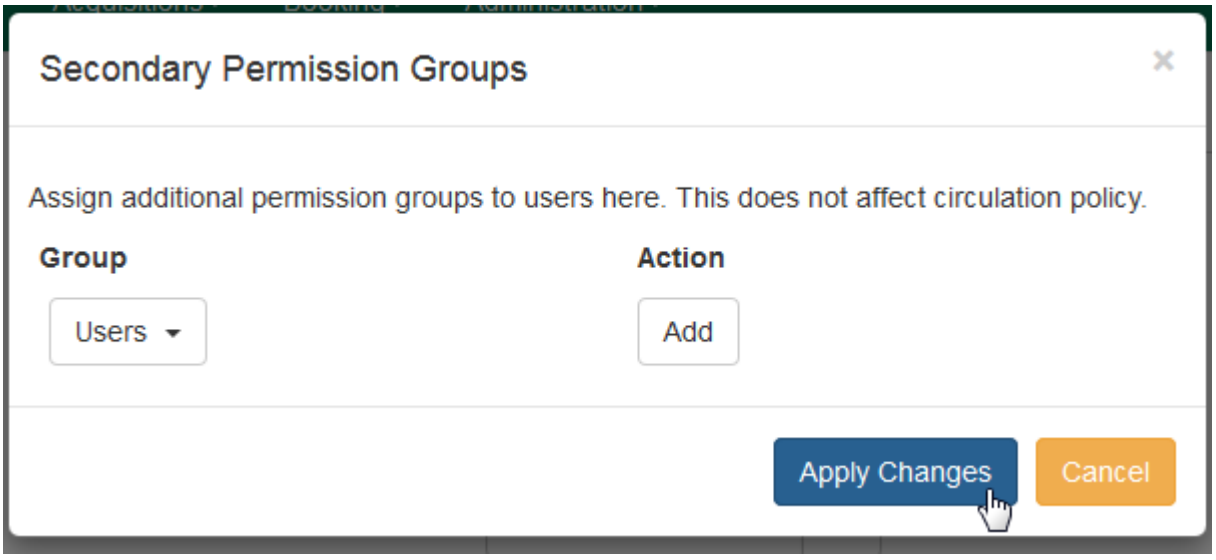
2. Click *Edit*.

3. Click *Secondary Groups*, located to the right of the *Main (Profile) Permission Group*.

4. Click *Delete* beside the permission group you would like to remove.



5. Click *Apply Changes*.



6. Click *Save* in the top right hand corner of the *Edit Screen* to save the user's account.

Managing role-based permission groups in the database

While the ability to assign a user to multiple permission groups has existed in Evergreen for years, a staff client interface is not currently available to facilitate the work of the Evergreen administrator. However, if you or members of your team are comfortable working directly with the Evergreen database, you can use this approach to separate the borrowing profile of your users from the permissions that you grant to staff, while minimizing the amount of overlapping permissions that you need to manage for a set of permission groups that would otherwise multiply exponentially to represent all possible combinations of staff roles.

In the following example, we create three new groups:

- a *Student* group used to determine borrowing privileges
- a *Student Cataloger* group representing a limited set of cataloging permissions appropriate for students
- a *Student Circulator* group representing a limited set of circulation permissions appropriate for students

Then we add three new users to our system: one who needs to perform some cataloging duties as a student; one who needs perform some circulation duties as a student; and one who needs to perform both cataloging and circulation duties. This section demonstrates how to add these permissions to the users at the database level.

To create the Student group, add a new row to the *permission.grp_tree* table as a child of the *Patrons* group:

```
INSERT INTO permission.grp_tree (name, parent, usergroup, description, application_perm)
SELECT 'Students', pgt.id, TRUE, 'Student borrowers', 'group_application.user.patron.student'
FROM permission.grp_tree pgt
 WHERE name = 'Patrons';
```

To create the Student Cataloger group, add a new row to the *permission.grp_tree* table as a child of the *Staff* group:

```
INSERT INTO permission.grp_tree (name, parent, usergroup, description, application_perm)
SELECT 'Student Catalogers', pgt.id, TRUE, 'Student catalogers', 'group_application.user.staff.student_cataloger'
FROM permission.grp_tree pgt
WHERE name = 'Staff';
```

To create the Student Circulator group, add a new row to the *permission.grp_tree* table as a child of the *Staff* group:

```
INSERT INTO permission.grp_tree (name, parent, usergroup, description, application_perm)
SELECT 'Student Circulators', pgt.id, TRUE, 'Student circulators', 'group_application.user.staff.student_circulator'
FROM permission.grp_tree pgt
WHERE name = 'Staff';
```

We want to give the Student Catalogers group the ability to work with MARC records at the consortial level, so we assign the UPDATE_MARC, CREATE_MARC, and IMPORT_MARC permissions at depth 0:

```
WITH pgt AS (
  SELECT id
  FROM permission.grp_tree
  WHERE name = 'Student Catalogers'
)
INSERT INTO permission.grp_perm_map (grp, perm, depth)
SELECT pgt.id, ppl.id, 0
FROM permission.perm_list ppl, pgt
WHERE ppl.code IN ('UPDATE_MARC', 'CREATE_MARC', 'IMPORT_MARC');
```

Similarly, we want to give the Student Circulators group the ability to check out items and record in-house uses at the system level, so we assign the COPY_CHECKOUT and CREATE_IN_HOUSE_USE permissions at depth 1 (overriding the same *Staff* permissions that were granted only at depth 2):

```
WITH pgt AS (
  SELECT id
  FROM permission.grp_tree
  WHERE name = 'Student Circulators'
) INSERT INTO permission.grp_perm_map (grp, perm, depth)
SELECT pgt.id, ppl.id, 1
FROM permission.perm_list ppl, pgt
WHERE ppl.code IN ('COPY_CHECKOUT', 'CREATE_IN_HOUSE_USE');
```

Finally, we want to add our students to the groups. The request may arrive in your inbox from the library along the lines of "Please add Mint Julep as a Student Cataloger, Bloody Caesar as a Student Circulator, and Grass Hopper as a Student Cataloguer / Circulator; I've already created their accounts and given them a work organizational unit." You can translate that into the following SQL to add the users to the pertinent permission groups, adjusting for the inevitable typos in the names of the users.

First, add our Student Cataloger:

```
WITH pgt AS (
  SELECT id FROM permission.grp_tree
  WHERE name = 'Student Catalogers'
)
INSERT INTO permission.usr_grp_map (usr, grp)
SELECT au.id, pgt.id
FROM actor.usr au, pgt
WHERE first_given_name = 'Mint' AND family_name = 'Julep';
```

Next, add the Student Circulator:

```
WITH pgt AS (
  SELECT id FROM permission.grp_tree
  WHERE name = 'Student Circulators'
)
INSERT INTO permission.usr_grp_map (usr, grp)
SELECT au.id, pgt.id
FROM actor.usr au, pgt
WHERE first_given_name = 'Bloody' AND family_name = 'Caesar';
```

Finally, add the all-powerful Student Cataloger / Student Circulator:

```
 WITH pgt AS (
  SELECT id FROM permission.grp_tree
  WHERE name IN ('Student Catalogers', 'Student Circulators')
)
INSERT INTO permission.usr_grp_map (usr, grp)
SELECT au.id, pgt.id
FROM actor.usr au, pgt
WHERE first_given_name = 'Grass' AND family_name = 'Hopper';
```

While adopting this role-based approach might seem labour-intensive when applied to a handful of students in this example, over time it can help keep the permission profiles of your system relatively simple in comparison to the alternative approach of rapidly reproducing permission groups, overlapping permissions, and permissions granted on a one-by-one basis to individual users.

Authentication Proxy

To support integration of Evergreen with organizational authentication systems, and to reduce the proliferation of user names and passwords, Evergreen offers a service called open-ils.auth_proxy. If you enable the service,

open-ils.auth_proxy supports different authentication mechanisms that implement the authenticate method. You can define a chain of these authentication mechanisms to be tried in order within the **<authenticators>** element of the *opensrf.xml* configuration file, with the option of falling back to the native mode that uses Evergreen's internal method of password authentication.

This service only provides authentication. There is no support for automatic provisioning of accounts. To authenticate using any authentication system, the user account must first be defined in the Evergreen database. The user will be authenticated based on the Evergreen username and must match the user's ID on the authentication system.

In order to activate Authentication Proxy, the Evergreen system administrator will need to complete the following steps:

1. Edit **opensrf.xml**.

   a. Set the **open-ils.auth_proxy** app settings **enabled** tag to **true**

   b. Add the **authenticator** to the list of authenticators or edit the existing example authenticator:

```
<authenticator>
        <name>ldap</name>
        <module>OpenILS::Application::AuthProxy::LDAP_Auth</module>
        <hostname>name.domain.com</hostname>
        <basedn>ou=people,dc=domain,dc=com</basedn>
        <authid>cn=username,ou=specials,dc=domain,dc=com</authid>
        <id_attr>uid</id_attr>
        <password>my_ldap_password_for_authid_user</password>
        <login_types>
                <type>staff</type>
                <type>opac</type>
        </login_types>
        <org_units>
                <unit>103</unit>
                <unit>104</unit>
        </org_units>
</authenticator>
```

   - **name** : Used to identify each authenticator.
   - **module** : References to the perl module used by Evergreen to process the request.
   - **hostname** : Hostname of the authentication server.
   - **basedn** : Location of the data on your authentication server used to authenticate users.
   - **authid** : Administrator ID information used to connect to the Authentication server.
   - **id_attr** : Field name in the authenticator matching the username in the Evergreen database.
   - **password** : Administrator password used to connect to the authentication server. Password for the **authid**.
   - **login_types** : Specifies which types of logins will use this authenticator. This might be useful if staff use a different LDAP directory than general users.
   - **org_units** : Specifies which org units will use the authenticator. This is useful in a consortium environment where libraries will use separate authentication systems.

2. Restart Evergreen and Apache to activate configuration changes.

If using proxy authentication with library employees that will click the *Change Operator* feature in the client software, then add "Temporary" as a **login_types**.

Patron Address City/State/County Pre-Populate by ZIP Code

This feature saves staff time and increases accuracy when entering patron address information by automatically filling in the City, State and County information based on the ZIP code entered by the staff member.

Released: Evergreen 0.1, available in all versions.

Please be aware of the following when using this feature.

- ZIP codes do not always match 1 to 1 with City, State and County. ZIP codes were designed for postal delivery and represent postal delivery zones that may cover more than one city, state or county.
  - It is currently only possible to have one match per ZIP code, but you can add an alert to those entries to prompt staff to double check the entered data.
- Only the first 5 digits of the ZIP are used. ZIP+4 is not currently supported.
- The zips.txt data is loaded once at service startup and stored in memory, so changes to the zips.txt data file require that Evergreen be restarted. Specifically, you need to restart the "open-ils.search" OpenSRF service.

Scoping and Permissions

There are no staff client permissions associated with this feature since there is no staff client interface.

This feature affects all users of the system; there is no way to have separate settings per Org Unit.

Setup Steps

Step 1 - Setup Data File

The default location and name of the data file is /openils/var/data/zips.txt on your Evergreen server. You can choose a different location if needed.

The file format of your zips.txt will look like this (delimited by the .):

ID|**StateAbb**|**City**|**ZIP**|**IsDefault**|StateID|**County**|AreaCode|**AlertMesg**

The only fields that are used are **StateAbb**, **City**, **ZIP**, **IsDefault**, **County** and **AlertMesg**.

Most fields can be left blank if the information is not available and that data will not be entered.

Data Field Descriptions

1. ID - ID field to uniquely identify this row. Not required, can be left blank.

2. **StateAbb** - State abbreviation like "MN" or "ND".

3. **City** - Name of city.

4. **ZIP** - ZIP code, only first 5 digits used.

5. **IsDefault** - Must be set to 1 for the row to be used. Easy way to disable/enable a row.

6. StateID - Unknown and unused.

7. **County** - County name.

8. AreaCode - Phone number area code, unused.

9. **AlertMesg** - Message to display to staff to alert them of any special circumstances.

The Address Alerts feature — described in the Staff Client Sysadmin manual — can also be used to alert staff about certain addresses.

Here is an example of what the data file should look like.

Example zips.txt.

```
|MN|Moorhead|56561|1||Clay||
|MN|Moorhead|56562|1||Clay||
|MN|Moorhead|56563|1||Clay||
|MN|Sabin|56580|1||Clay||
|MN|Ulen|56585|1||Clay||
|MN|Lake Itasca|56460|1||Clearwater County||
|MN|Bagley|56621|1||Clearwater||
|MN|Clearbrook|56634|1||Clearwater||
|MN|Gonvick|56644|1||Clearwater||
```

Step 2 - Enable Feature

The next step is to tell the system to use the zips.txt file that you created. This is done by editing /openils/conf/opensrf.xml. Look about halfway into the file and you may very well see a commented section in the file that looks similar to this:

```
        <!-- zip code database file -->
        <!--<zips_file>/openils/var/data/zips.txt</zips_file>-->
    </app_settings>
</open-ils.search>
```

Uncomment the area by . .. Change the file path if you placed your file in a different location. The file should look like this after you are done.

```
        <!-- zip code database file -->
        <zips_file>/openils/var/data/zips.txt</zips_file>
    </app_settings>
</open-ils.search>
```

Save and Restart. Save your changes to the opensrf.xml file, restart Evergreen and restart Apache.

The specific opensrf services you need to restart are "opensrf.setting" and "open-ils.search".

Step 3 - Test

Open up the staff client and try to register a new patron. When you get to the address section, enter a ZIP code that you know is in your zips.txt file. The data from the file that matches your ZIP will auto fill the city, state and county fields.

ZIP Code Data

There are several methods you can use to populate your zips.txt with data.

Manual Entry

If you only have a few communities that you serve, entering data manually may be the simplest approach.

Geonames.org Data

Geonames.org provides free ZIP code to city, state and county information licensed under the Creative Commons Attribution 3.0 License, which means you need to put a link to them on your website. Their data includes primary city, state and county information only. It doesn't include info about which other cities are included in a ZIP code. Visit http://www.geonames.org for more info.

The following code example shows you how to download and reformat the data into the zips.txt format. You have the option to filter the data to only include certain states also.

```
## How to get a generic Evergreen zips.txt for free
wget http://download.geonames.org/export/zip/US.zip
unzip US.zip
cut -f2,3,5,6 US.txt \
| perl -ne 'chomp; @f=split(/\t/); print "|" . join("|", (@f[2,1,0], "1", "", $f[3], "")), "|\n";' \
> zips.txt

##Optionally filter the data to only include certain states
egrep "^\|(ND|MN|WI|SD)\|" zips.txt  > zips-mn.txt
```

Commercial Data

There are many vendors that sell databases that include ZIP code to city, state and county information. A web search will easily find them. Many of the commercial vendors will include more information on which ZIP codes cover multiple cities, counties and states, which you could use to populate the alert field.

Existing Patron Database

Another possibility is to use your current patron database to build your zips.txt. Pull out the current ZIP, city, state, county unique rows and use them to form your zips.txt.

Small Sites. For sites that serve a small geographic area (less than 30 ZIP codes), an sql query like the following will create a zips.txt for you. It outputs the number of matches as the first field and sorts by ZIP code and number of matches. You would need to go through the resulting file and deal with duplicates manually.

```
psql egdb26 -A -t -F $'|' \
 -c "SELECT count(substring(post_code from 1 for 5)) as zipcount, state, \
 city, substring(post_code from 1 for 5) as pc, \
 '1', '', county, '', '' FROM actor.usr_address \
 group by pc, city, state, county \
 order by pc, zipcount DESC" > zips.txt
```

Larger Sites. For larger sites Ben Ostrowsky at ESI created a pair of scripts that handles deduplicating the results and adding in county information. Instructions for use are included in the files.

- http://git.esilibrary.com/?p=migration-tools.git;a=blob;f=elect_ZIPs
- http://git.esilibrary.com/?p=migration-tools.git;a=blob;f=enrich_ZIPs

Development

If you need to make changes to how this feature works, such as to add support for other postal code formats, here is a list of the files that you need to look at.

1. **Zips.pm -** contains code for loading the zips.txt file into memory and replying to search queries. Open-ILS / src / perlmods / lib / OpenILS / Application / Search / Zips.pm
2. **register.js** - This is where patron registration logic is located. The code that queries the ZIP search service and fills the address is located here. Open-ILS / web / js / ui / default / actor / user / register.js

Apache Rewrite Tricks

It is possible to use Apache's Rewrite Module features to perform a number of useful tricks that can make people's lives much easier.

Short URLs

Making short URLs for common destinations can simplify making printed media as well as shortening or simplifying what people need to type. These are also easy to add and require minimal maintenance, and generally can be implemented with a single line addition to your eg_vhost.conf file.

```
# My Account - http://host.ext/myaccount -> My Account Page
RewriteRule ^/myaccount https://%{HTTP_HOST}/eg/opac/myopac/main [R]

# ISBN Search - http://host.ext/search/isbn/<ISBN NUMBER> -> Search Page
RewriteRule ^/search/isbn/(.*) /eg/opac/results?_special=1&qtype=identifier|isbn&query=$1 [R]
```

Domain Based Content with RewriteMaps

One creative use of Rewrite features is domain-based configuration in a single eg_vhost.conf file. Regardless of how many VirtualHost blocks use the configuration you don't need to duplicate things for minor changes, and can in fact use wildcard VirtualHost blocks to serve multiple subdomains.

For the wildcard blocks you will want to use a ServerAlias directive, and for SSL VirtualHost blocks ensure you have a wildcard SSL certificate.

```
ServerAlias *.example.com
```

For actually changing things based on the domain, or subdomain, you can use RewriteMaps. Each RewriteMap is generally a lookup table of some kind. In the following examples we will generally use text files, though database lookups and external programs are also possible.

Note that in the examples below we generally store things in Environment Variables. From within Template Toolkit templates you can access environment variables with the ENV object.

Template Toolkit ENV example, link library name/url if set.

```
[% IF ENV.eglibname && ENV.egliburl %]<a href="[% ENV.egliburl %]">[% ENV.eglibname %]</a>[% END %]
```

The first lookup to do is a domain to identifier, allowing us to re-use identifiers for multiple domains. In addition we can also supply a default identifier, for when the domain isn't present in the lookup table.

Apache Config.

```
# This internal map allows us to lowercase our hostname, removing case issues in our lookup table
# If you prefer uppercase you can use "uppercase int:toupper" instead.
RewriteMap lowercase int:tolower
# This provides a hostname lookup
RewriteMap eglibid txt:/openils/conf/libid.txt
# This stores the identifier in a variable (eglibid) for later use
# In this case CONS is the default value for when the lookup table has no entry
RewriteRule . - [E=eglibid:${eglibid:${lowercase:%{HTTP_HOST}}|CONS}]
```

Contents of libid.txt File.

```
# Comments can be included
# Multiple TLDs for Branch 1
branch1.example.com BRANCH1
branch1.example.net BRANCH1
# Branches 2 and 3 don't have alternate TLDs
branch2.example.com BRANCH2
branch3.example.com BRANCH3
```

Once we have identifiers we can look up other information, when appropriate. For example, say we want to look up library names and URLs:

Apache Config.

```
# Library Name Lookup - Note we provide no default in this case.
RewriteMap eglibname txt:/openils/conf/libname.txt
RewriteRule . - [E=eglibname:${eglibname:%{ENV:eglibid}}]
# Library URL Lookup - Also with no default.
RewriteMap egliburl txt:/openils/conf/liburl.txt
RewriteRule . - [E=egliburl:${egliburl:%{ENV:eglibid}}]
```

Contents of libname.txt File.

```
# Note that we cannot have spaces in the "value", so instead &#32; is used.   is also an option.
BRANCH1 Branch&#32;One
BRANCH2 Branch&#32;Two
BRANCH3 Branch&#32;Three
CONS Example&#32;Consortium&#32;Name
```

Contents of liburl.txt File.

```
BRANCH1 http://branch1.example.org
BRANCH2 http://branch2.example.org
BRANCH3 http://branch3.example.org
CONS http://example.org
```

Or, perhaps set the "physical location" variable for default search/display library:

Apache Config.

```
# Lookup "physical location" IDs
RewriteMap eglibphysloc txt:/openils/conf/libphysloc.txt
# Note: physical_loc is a variable used in the TTOPAC and should not be re-named
RewriteRule . - [E=physical_loc:${eglibphysloc:%{ENV:eglibid}}]
```

Contents of libphysloc.txt File.

```
BRANCH1 4
BRANCH2 5
BRANCH3 6
CONS 1
```

Going further, you could also replace files to be downloaded, such as images or stylesheets, on the fly:

Apache Config.

```
# Check if a file exists based on eglibid and the requested file name
# Say, BRANCH1/opac/images/main_logo.png
RewriteCond %{DOCUMENT_ROOT}/%{ENV:eglibid}%{REQUEST_URI} -f
# Serve up the eglibid version of the file instead
RewriteRule (.*) /%{ENG:eglibid}$1
```

Note that template files themselves cannot be replaced in that manner.

Apache Access Handler Perl Module

The OpenILS::WWW::AccessHandler Perl module is intended for limiting patron access to configured locations in Apache. These locations could be folder trees, static files, non-Evergreen dynamic content, or other Apache features/modules. It is intended as a more patron-oriented and transparent version of the OpenILS::WWW::Proxy and OpenILS::WWW:Proxy::Authen modules.

Instead of using Basic Authentication the AccessHandler module instead redirects to the OPAC for login. Once logged in additional checks can be performed, based on configured variables:

- Permission Checks (at Home OU or specified location)
- Home OU Checks (Org Unit or Descendant)
- "Good standing" Checks (Not Inactive or Barred)

Use of the module is a simple addition to a Location block in Apache:

```
<Location /path/to/be/protected>
    PerlAccessHandler OpenILS::WWW::AccessHandler
    # For each option you wish to set:
    PerlSetVar OPTION "VALUE"
</Location>
```

The available options are:

OILSAccessHandlerLoginURL
- Default: /eg/opac/login
- The page to redirect to when Login is needed

OILSAccessHandlerLoginURLRedirectVar
- Default: redirect_to
- The variable the login page wants the "destination" URL stored in

OILSAccessHandlerFailURL
- Default: <unset>
- URL to go to if Permission, Good Standing, or Home OU checks fail. If not set a 403 error is generated instead. To customize the 403 you could use an ErrorDocument statement.

OILSAccessHandlerCheckOU	• Default: <User Home OU> • Org Unit to check Permissions at and/or to load Referrer from. Can be a shortname or an ID.
OILSAccessHandlerPermission	• Default: <unset> • Permission, or comma- or space-delimited set of permissions, the user must have to access the protected area.
OILSAccessHandlerGoodStanding	• Default: 0 • If set to a true value the user must be both Active and not Barred.
OILSAccessHandlerHomeOU	• Default: <unset> • An Org Unit, or comma- or space-delimited set of Org Units, that the user's Home OU must be equal to or a descendant of to access this resource. Can be set to shortnames or IDs.
OILSAccessHandlerReferrerSetting	• Default: <unset> • Library Setting to pull a forced referrer string out of, if set.

As the AccessHandler module does not actually serve the content it is protecting, but instead merely hands control back to Apache when it is done authenticating, you can protect almost anything else you can serve with Apache.

Use Cases

The general use of this module is "protect access to something else" - what that something else is will vary. Some possibilities:

- Apache features
  - Automatic Directory Indexes
  - Proxies (see below)
    - Electronic Databases
    - Software on other servers/ports
- Non-Evergreen software
  - Timekeeping software for staff
  - Specialized patron request packages
- Static files and folders
  - Semi-public Patron resources
  - Staff-only downloads

Proxying Websites

One potentially interesting use of the AccessHandler module is to protect an Apache Proxy configuration. For example, after installing and enabling mod_proxy, mod_proxy_http, and mod_proxy_html you could proxy websites like so:

```
<Location /proxy/>
    # Base "Rewrite URLs" configuration
    ProxyHTMLLinks  a       href
    ProxyHTMLLinks  area        href
    ProxyHTMLLinks  link        href
    ProxyHTMLLinks  img     src longdesc usemap
    ProxyHTMLLinks  object      classid codebase data usemap
    ProxyHTMLLinks  q       cite
    ProxyHTMLLinks  blockquote  cite
    ProxyHTMLLinks  ins     cite
    ProxyHTMLLinks  del     cite
    ProxyHTMLLinks  form        action
    ProxyHTMLLinks  input       src usemap
    ProxyHTMLLinks  head        profile
    ProxyHTMLLinks  base        href
    ProxyHTMLLinks  script      src for

    # To support scripting events (with ProxyHTMLExtended On)
    ProxyHTMLEvents onclick ondblclick onmousedown onmouseup \
            onmouseover onmousemove onmouseout onkeypress \
            onkeydown onkeyup onfocus onblur onload \
            onunload onsubmit onreset onselect onchange

    # Limit all Proxy connections to authenticated sessions by default
    PerlAccessHandler OpenILS::WWW::AccessHandler

    # Strip out Evergreen cookies before sending to remote server
    RequestHeader edit Cookie "^(.*?)ses=.*?(?:$|;)(.*)$" $1$2
    RequestHeader edit Cookie "^(.*?)eg_loggedin=.*?(?:$|;)(.*)$" $1$2
</Location>

<Location /proxy/example/>
    # Proxy example.net
    ProxyPass http://www.example.net/
    ProxyPassReverse http://www.example.net/
    ProxyPassReverseCookieDomain example.net example.com
    ProxyPassReverseCookiePath / /proxy/example/

    ProxyHTMLEnable On
    ProxyHTMLURLMap http://www.example.net/ /proxy/example/
    ProxyHTMLURLMap / /proxy/mail/
    ProxyHTMLCharsetOut *

    # Limit to BR1 and BR3 users
    PerlSetVar OILSAccessHandlerHomeOU "BR1,BR3"
</Location>
```

As mentioned above, this can be used for multiple reasons. In addition to websites such as online databases for patron use you may wish to proxy software for staff or patron use to make it appear on your catalog domain, or perhaps to keep from needing to open extra ports in a firewall.

Chapter 13. Updating translations using Launchpad

This document describes how to update the translations in an Evergreen branch by pulling them from Launchpad, as well as update the files to be translated in Launchpad by updating the POT files in the Evergreen master branch.

Prerequisites

You must install all of the Python prerequisites required for building translations, per http://evergreen-ils.org/dokuwiki/doku.php?id=evergreen-admin:customizations:i18n

- polib
- translate-toolkit
- levenshtein
- setuptools
- simplejson
- lxml

Updating the translations

1. Check out the latest translations from Launchpad by branching the Bazaar repository:

   ```
   bzr branch lp:~denials/evergreen/translation-export
   ```

   This creates a directory called "translation-export".

2. Ensure you have an updated Evergreen release branch.

3. Run the `build/i18n/scripts/update_pofiles` script to copy the translations into the right place and avoid any updates that are purely metadata (dates generated, etc).

4. Commit the lot! And backport to whatever release branches need the updates.

5. Build updated POT files:

   ```
   cd build/i18n
   make newpot
   ```

   This will extract all of the strings from the latest version of the files in Evergreen.

6. (This part needs automation): Then, via the magic of `git diff` and `git add`, go through all of the changed files and determine which ones actually have string changes. Recommended approach is to re-run `git diff` after each `git add`.

7. Commit the updated POT files and backport to the pertinent release branches.

Part V. Cataloging Administration

Table of Contents

14. Cataloging Staff Interface 100
Administering the Physical Characteristics Wizard 100
15. Cataloging timesavers and shortcuts 101
MARC Templates 101
Adding MARC Templates 101
16. Notes about the Bibliographic Schema in the Database 103
Bibliographic fingerprint 103

Chapter 14. Cataloging Staff Interface

Administering the Physical Characteristics Wizard

The MARC 007 Field Physical Characteristics Wizard enables catalogers to interact with a database wizard that leads the user step-by-step through the MARC 007 field positions. The wizard displays the significance of the current position and provides dropdown lists of possible values for the various components of the MARC 007 field in a more user-friendly way.

The information driving the MARC 007 Field Physical Characteristics Wizard is already a part of the Evergreen database. This data can be customized by individual sites and / or updated when the Library of Congress dictates new values or positions in the 007 field. There are three relevant tables where the information that drives the wizard is stored:

1. **config.marc21_physical_characteristic_type_map** contains the list of materials, or values, for the positions of the 007 field.

2. **config.marc21_physical_characteristic_subfield_map** contains rows that list the meaning of the various positions in the 007 field for each Category of Material.

3. **config.marc21_physical_characteristic_value_map** lists all of the values possible for all of the positions in the config.marc21_physical_characteristic_subfield_map table.

Chapter 15. Cataloging timesavers and shortcuts

MARC Templates

MARC Templates make the cataloging process more efficient for catalogers. At this time, MARC Templates have to be created on the server, rather than in the Web client.

Adding MARC Templates

1. Create a marc template in the directory */openils/var/templates/marc/*. It should be in xml format. Here is an example file `k_book.xml`:

```
<record>
  <leader>00620cam a2200205Ka 4500</leader>
  <controlfield tag="008">070101s                              eng d</controlfield>
  <datafield tag="010" ind1="" ind2="">
    <subfield code="a"></subfield>
  </datafield>
  <datafield tag="020" ind1="" ind2="">
    <subfield code="a"></subfield>
  </datafield>
  <datafield tag="082" ind1="0" ind2="4">
    <subfield code="a"></subfield>
  </datafield>
  <datafield tag="092" ind1="" ind2="">
    <subfield code="a"></subfield>
  </datafield>
  <datafield tag="100" ind1="" ind2="">
    <subfield code="a"></subfield>
  </datafield>
  <datafield tag="245" ind1="" ind2="">
    <subfield code="a"></subfield>
    <subfield code="b"></subfield>
    <subfield code="c"></subfield>
  </datafield>
  <datafield tag="260" ind1="" ind2="">
    <subfield code="a"></subfield>
    <subfield code="b"></subfield>
    <subfield code="c"></subfield>
  </datafield>
  <datafield tag="300" ind1="" ind2="">
    <subfield code="a"></subfield>
    <subfield code="b"></subfield>
    <subfield code="c"></subfield>
  </datafield>
  <datafield tag="500" ind1="" ind2="">
    <subfield code="a"></subfield>
  </datafield>
  <datafield tag="650" ind1="" ind2="">
    <subfield code="a"></subfield>
    <subfield code="v"></subfield>
  </datafield>
  <datafield tag="650" ind1="" ind2="">
    <subfield code="a"></subfield>
  </datafield>
</record>
```

2. Add the template to the to the marctemplates list in the *open-ils.cat* section of the Evergreen configuration file `opensrf.xml`.

3. Restart perl services for changes to take effect with the command `/openils/bin/osrf_control -l --restart --service=open-ils.cat`

Chapter 16. Notes about the Bibliographic Schema in the Database

Bibliographic fingerprint

Evergreen creates a fingerprint for each bib record, which can be found in the `fingerprint` column of the `biblio.record_entry` table. This fingerprint is used to group together different bib records in a Group Formats & Editions search in the public catalog.

The bibliographic fingerprint incorporates several subfields to distinguish between different items, including:

- $n and $p from MARC title fields to better distinguish among records of the same series that may share the same title but have a different part.

The bibliographic fingerprint distinguishes among the fields contributing to the fingerprint. This helps the system distinguish between a record for the movie *Blue Steel* and another record for the book *Blue* written by Danielle *Steel*.

Part VI. Managing Staff from the Command Line

Table of Contents

17. Changing passwords 106

Chapter 17. Changing passwords

If you need to change a patron or staff account password without using the staff client, here is how you can reset it with SQL.

Connect to your Evergreen database using *psql* or similar tool, and retrieve and verify your admin username:

```
psql -U <user-name> -h <hostname> -d <database>
```

```
SELECT id, usrname, passwd from actor.usr where usrname = 'admin';
```

If you do not remember the username that you set, search for it in the *actor.usr* table, and then reset the password.

```
UPDATE actor.usr SET passwd = <password> WHERE id=<id of row to be updated>;
```

The new password will automatically be hashed.

Part VII. Patron Data

Table of Contents

18. Aging Circulations 109
Global Flags 109
What Data is Aged? 109
How Circulations are Aged 110
Impacts on Billing Data 110
19. Purging holds 111
20. Purge User Activity 112

Chapter 18. Aging Circulations

Use case

Aging circulations helps to protect patron privacy and save disk space.

Evergreen allows for the bulk anonymization of circulation histories. Evergreen calls this aged circulation. Circulation statistics are preserved (total circs, last checkout/renewal date, checkout/renewal/checkin workstation, etc) but patron information (name : barcode) is replaced with <Aged Circulation> text and the link to the patron record is removed.

In the client, <Aged Circulation> will show in the patron field in Circulation History Tab and Show Last Few Circulations.

In the database, every time you attempt to `DELETE` a row from `action.circ`, it copies over the appropriate data to `action.aged_circulation`, then deletes the `action.circ` row.

Global Flags

There are four global flags used for aging circulations.

1. Historical Circulation Retention Age - determines the timeframe for aging circulations based on transaction age (7 days, 14 days, 30 days, etc).
2. Historical Circulations Per Item - determines how many circulations to keep (ex. 1, 2, 3). If set to 1, Evergreen will always keep the last (most recent) circulation.
3. Historical Circulations use most recent xact_finish date instead of last circ's (true or false)
4. Historical Circulations are kept for global retention age at a minimum, regardless of user preferences (true or false)

What Data is Aged?

Only completed transactions are aged. These circulations have been checked in (returned) and **do not** contain any unpaid fines or bills.

Data that is not aged includes:

- open transactions (i.e. checked out)
- closed transactions with unpaid fines
- closed transactions with unpaid bills
- the last X circulation(s) (determined by historical circulations per item flag)

Aging circulations will not affect a patron being able to keep their checkout history. Minimal metadata is stored in the patron checkout history table. Once the corresponding circulation is aged, the full circulation metadata is no longer linked to the patron's reading history.

Just aging circulations is not sufficient to protect patron circulation history. Fully protecting these data would also involve a thoughtful approach to logs and backups of these data.

You can create a cron job to automatically age circulations.

How Circulations are Aged

The action.aged_circulation table is for statistical reporting while breaking the link to the patron who had the item checked out.

Circulations get moved under three circumstances in stock Evergreen:

1. A patron is deleted. This moves all of the patron's circulations from action.circulation to action.aged_circulation
2. A row or row(s) in action.circulation are deleted. The action.age_circ_on_delete trigger moves deleted action.circulations to action.aged_circulation.
3. The action.purge_circulations function is run. This function is meant to be run periodically to enforce patron privacy. It's behavior is controlled by two internal flags: history.circ.retention_age and history.circ.retention_count.

The purge_circulations function is often run from a cron via the purge_circulations.srfsh script.

The purge_circulations function will take a **long** time to run for the first time on a system that has had much activity. The srfsh script will likely time out before the database function finishes and nothing will get moved.

Impacts on Billing Data

Rows are deleted from money.materialized_billable_xact_summary when circulations are aged. This table is the basis for billing reports and views.

currently grocery bills are ignored and not aged.

Chapter 19. Purging holds

Similar to purging circulations one may wish to purge old (filled or canceled) hold information. This feature adds a database function and settings for doing so.

Purged holds are moved to the *action.aged_hold_request* table with patron identifying information scrubbed, much like circulations are moved to *action.aged_circulation*.

The settings allow for a default retention age as well as distinct retention ages for holds filled, holds canceled, and holds canceled by specific cancel causes. The most specific one wins unless a patron is retaining their hold history. In the latter case the patron’s holds are retained either way.

Note that the function still needs to be called, which could be set up as a cron job or done more manually, say after statistics collection. You can use the *purge_holds.srfsh* script to purge holds from cron.

Chapter 20. Purge User Activity

User activity types are now set to transient by default for new Evergreen installs. This means only the most recent activity entry per user per activity type is retained in the database.

Use case

Setting more user activity types collects less patron data, which helps protect patron privacy. Additionally, the *actor.usr_activity* table gets really big really fast if all event types are non-transient.

This change does not affect existing activity types, which were set to non-transient by default. To make an activity type transient, modify the *Transient* field of the desired type in the staff client under Admin → Server Administration → User Activity Types.

Setting an activity type to transient means data for a given user will be cleaned up automatically if and when the user performs the activity in question. However, administrators can also force an activity cleanup via SQL. This is useful for ensuring that all old activity data is deleted and for controlling when the cleanup occurs, which may be useful on very large actor.usr_activity tables.

To force clean all activity types:

```
SELECT actor.purge_usr_activity_by_type(etype.id)
    FROM config.usr_activity_type etype;
```

This could take hours to run on a very large actor.usr_activity table.

Part VIII. Backing up your Evergreen System

Table of Contents

21. Database backups 115
Creating logical database backups 115
Restoring from logical database backups 115
Creating physical database backups with support for point-in-time recovery 116
Creating a replicated database 117

Chapter 21. Database backups

Although it might seem pessimistic, spending some of your limited time preparing for disaster is one of the best investments you can make for the long-term health of your Evergreen system. If one of your servers crashes and burns, you want to be confident that you can get a working system back in place — whether it is your database server that suffers, or an Evergreen application server.

At a minimum, you need to be able to recover your system's data from your PostgreSQL database server: patron information, circulation transactions, bibliographic records, and the like. If all else fails, you can at least restore that data to a stock Evergreen system to enable your staff and patrons to find and circulate materials while you work on restoring your local customizations such as branding, colors, or additional functionality. This section describes how to back up your data so that you or a colleague can help you recover from various disaster scenarios.

Creating logical database backups

The simplest method to back up your PostgreSQL data is to use the `pg_dump` utility to create a logical backup of your database. Logical backups have the advantage of taking up minimal space, as the indexes derived from the data are not part of the backup. For example, an Evergreen database with 2.25 million records and 3 years of transactions that takes over 120 GB on disk creates just a 7.0 GB compressed backup file. The drawback to this method is that you can only recover the data at the exact point in time at which the backup began; any updates, additions, or deletions of your data since the backup began will not be captured. In addition, when you restore a logical backup, the database server has to recreate all of the indexes—so it can take several hours to restore a logical backup of that 2.25 million record Evergreen database.

As the effort and server space required for logical database backups are minimal, your first step towards preparing for disaster should be to automate regular logical database backups. You should also ensure that the backups are stored in a different physical location, so that if a flood or other disaster strikes your primary server room, you will not lose your logical backup at the same time.

To create a logical dump of your PostgreSQL database:

1. Issue the command to back up your database: `pg_dump -Fc <database-name> > <backup-filename>`. If you are not running the command as the postgres user on the database server itself, you may need to include options such as `-U <user-name>` and `-h <hostname>` to connect to the database server. You can use a newer version of the PostgreSQL to run `pg_dump` against an older version of PostgreSQL if your client and server operating systems differ. The `-Fc` option specifies the "custom" format: a compressed format that gives you a great deal of flexibility at restore time (for example, restoring only one table from the database instead of the entire schema).

2. If you created the logical backup on the database server itself, copy it to a server located in a different physical location.

You should establish a routine of nightly logical backups of your database, with older logical backups being automatically deleted after a given interval.

Restoring from logical database backups

To increase your confidence in the safety of your data, you should regularly test your ability to restore from a logical backup. Restoring a logical backup that you created using the custom format requires the use of the `pg_restore` tool as follows:

1. On the server on which you plan to restore the logical backup, ensure that you have installed PostgreSQL and the corresponding server package prerequisites. The `Makefile.install` prerequisite installer than came with your version of Evergreen contains an installation target that should satisfy these requirements. Refer to the installation documentation for more details.

2. As the `postgres` user, create a new database using the `createdb` command into which you will restore the data. Base the new database on the *template0* template database to enable the combination of UTF8 encoding and C locale options, and specify the character type and collation type as "C" using the `--lc-ctype` and `--lc-collate` parameters. For example, to create a new database called "testrestore": `createdb --template=template0 --lc-ctype=C --lc-collate=C testrestore`

3. As the `postgres` user, restore the logical backup into your newly created database using the `pg_restore` command. You can use the `-j` parameter to use more CPU cores at a time to make your recovery operation faster. If your target database is hosted on a different server, you can use the `-U <user-name>` and `-h <hostname>` options to connect to that server. For example, to restore the logical backup from a file named evergreen_20121212.dump into the "testrestore" database on a system with 2 CPU cores: `pg_restore -j 2 -d testrestore evergreen_20171212.dump`

Creating physical database backups with support for point-in-time recovery

While logical database backups require very little space, they also have the disadvantage of taking a great deal of time to restore for anything other than the smallest of Evergreen systems. Physical database backups are little more than a copy of the database file system, meaning that the space required for each physical backup will match the space used by your production database. However, physical backups offer the great advantage of almost instantaneous recovery, because the indexes already exist and simply need to be validated when you begin database recovery. Your backup server should match the configuration of your master server as closely as possible including the version of the operating system and PostgreSQL.

Like logical backups, physical backups also represent a snapshot of the data at the point in time at which you began the backup. However, if you combine physical backups with write-ahead-log (WAL) segment archiving, you can restore a version of your database that represents any point in time between the time the backup began and the time at which the last WAL segment was archived, a feature referred to as point-in-time recovery (PITR). PITR enables you to undo the damage that an accidentally or deliberately harmful UPDATE or DELETE statement could inflict on your production data, so while the recovery process can be complex, it provides fine-grained insurance for the integrity of your data when you run upgrade scripts against your database, deploy new custom functionality, or make global changes to your data.

To set up WAL archiving for your production Evergreen database, you need to modify your PostgreSQL configuration (typically located on Debian and Ubuntu servers in `/etc/postgresql/<version>/postgresql.conf`):

1. Change the value of `archive_mode` to on

2. Set the value of archive_command to a command that accepts the parameters `%f` (representing the file name of the WAL segment) and %p (representing the complete path name for the WAL segment, including the file name). You should copy the WAL segments to a remote file system that can be read by the same server on which you plan to create your physical backups. For example, if `/data/wal` represents a remote file system to which your database server can write, a possible value of archive_command could be: `test ! -f /data/wal/%f && cp %p /data/wal/%f`, which effectively tests to see if the destination file already exists, and if

it does not, copies the WAL segment to that location. This command can be and often is much more complex (for example, using `scp` or `rsync` to transfer the file to the remote destination rather than relying on a network share), but you can start with something simple.

Once you have modified your PostgreSQL configuration, you need to restart the PostgreSQL server before the configuration changes will take hold: . Stop your OpenSRF services. . Restart your PostgreSQL server. . Start your OpenSRF services and restart your Apache HTTPD server.

To create a physical backup of your production Evergreen database:

1. From your backup server, issue the `pg_basebackup -x -D <data-destination-directory> -U <user-name> -h <hostname> <database-name>` command to create a physical backup of database <database-name> on your backup server.

You should establish a process for creating regular physical backups at periodic intervals, bearing in mind that the longer the interval between physical backups, the more WAL segments the backup database will have to replay at recovery time to get back to the most recent changes to the database. For example, to be able to relatively quickly restore the state of your database to any point in time over the past four weeks, you might take physical backups at weekly intervals, keeping the last four physical backups and all of the corresponding WAL segments.

Creating a replicated database

If you have a separate server that you can use to run a replica of your database, consider replicating your database to that server. In the event that your primary database server suffers a hardware failure, having a database replica gives you the ability to fail over to your database replica with very little downtime and little or no data loss. You can also improve the performance of your overall system by directing some read-only operations, such as reporting, to the database replica. In this section, we describe how to replicate your database using PostgreSQL's streaming replication support.

You need to prepare your master PostgreSQL database server to support streaming replicas with several configuration changes. The PostgreSQL configuration file is typically located on Debian and Ubuntu servers at `/etc/postgresql/<version>/postgresql.conf`. The PostgreSQL host-based authentication (`pg_hba.conf`) configuration file is typically located on Debian and Ubuntu servers at `/etc/postgresql/<version>/pg_hba.conf`. Perform the following steps on your master database server:

1. Turn on streaming replication support. In postgresql.conf on your master database server, change `max_wal_senders` from the default value of 0 to the number of streaming replicas that you need to support. Note that these connections count as physical connections for the sake of the `max_connections` parameter, so you might need to increase that value at the same time.

2. Enable your streaming replica to endure brief network outages without having to rely on the archived WAL segments to catch up to the master. In `postgresql.conf` on your production database server, change `wal_keep_segments` to a value such as 32 or 64.

3. Increase the maximum number of log file segments between automatic WAL checkpoints. In `postgresql.conf` on your production database server, change checkpoint_segments from its default of 3 to a value such as 16 or 32. This improves the performance of your database at the cost of additional disk space.

4. Create a database user for the specific purpose of replication. As the postgres user on the production database server, issue the following commands, where replicant represents the name of the new user:

```
createuser replicant
psql -d <database> ALTER ROLE replicant WITH REPLICATION;
```

5. Enable your replica database to connect to your master database server as a streaming replica. In `pg_hba.conf` on your master database server, add a line to enable the database user replicant to connect to the master database server from IP address 192.168.0.164:

```
host    replication   replicant       192.168.0.164/32          md5
```

6. To enable the changes to take effect, restart your PostgreSQL database server.

To avoid downtime, you can prepare your master database server for streaming replication at any maintenance interval; then weeks or months later, when your replica server environment is available, you can begin streaming replication. Once you are ready to set up the streaming replica, perform the following steps on your replica server:

1. Ensure that the version of PostgreSQL on your replica server matches the version running on your production server. A difference in the minor version (for example, 9.1.3 versus 9.1.5) will not prevent streaming replication from working, but an exact match is recommended.

2. Create a physical backup of the master database server.

3. Add a `recovery.conf` file to your replica database configuration directory. This file contains the information required to begin recovery once you start the replica database:

```
# turn on standby mode, disabling writes to the database
standby_mode = 'on'
# assumes WAL segments are available at network share /data/wal
restore_command = 'cp /data/wal/%f %p'
# connect to the master database to being streaming replication
primary_conninfo = 'host=kochab.cs.uoguelph.ca user=replicant password=<password>
```

4. Start the PostgreSQL database server on your replica server. It should connect to the master. If the physical backup did not take too long and you had a high enough value for `wal_keep_segments` set on your master server, the replica should begin streaming replication. Otherwise, it will replay WAL segments until it catches up enough to begin streaming replication.

5. Ensure that the streaming replication is working. Check the PostgreSQL logs on your replica server and master server for any errors. Connect to the replica database as a regular database user and check for recent changes that have been made to your master server.

Congratulations, you now have a streaming replica database that reflects the latest changes to your Evergreen data! Combined with a routine of regular logical and physical database backups and WAL segment archiving stored on a remote server, you have a significant insurance policy for your system's data in the event that disaster does strike.

Part IX. UX Administration

Table of Contents

22. TPac Configuration and Customization 122
- Template toolkit documentation 122
- TPAC URL 122
- Perl modules used directly by TPAC 122
- Default templates 122
- Apache configuration files 122
- TPAC CSS and media files 123
- Mapping templates to URLs 123
- How to override templates 123
  - Defining multiple layers of overrides 124
- Changing some text in the TPAC 125
- Troubleshooting 126

23. Designing your catalog 127
- Configuring and customizing the public interface 127
  - Locating the default template files 127
  - Mapping templates to URLs 127
  - How to override template files 128
  - Configuring the custom templates directory in Apache's eg.conf 128
  - Adjusting colors for your public interface 128
  - Adjusting fonts in your public interface 129
  - Media file locations in the public interface 129
  - Changing some text in the public interface 129
  - Adding translations to PO file 130
  - Adding and removing MARC fields from the record details display page 130
- Setting the default physical location for your library environment 131
- Setting a default language and adding optional languages 131
  - Updating translations in Evergreen using current translations from Launchpad 132
- Change Date Format in Patron Account View 132
- Including External Content in Your Public Interface 133
  - OpenLibrary 133
  - ChiliFresh 133
  - Content Café 134
  - Obalkyknih.cz 134
  - Google Analytics 135
  - NoveList 135
  - RefWorks 135
  - SFX OpenURL Resolver 135
  - Syndetic Solutions 135
  - Clear External/Added Content Cache 136
  - Configure a Custom Image for Missing Images 137
- Including Locally Hosted Content in Your Public Interface 137
  - File Location and Format 137
  - Example 137
- Styling the searchbar on the homepage 138

24. Designing the patron search experience 139
- Editing the formats select box options in the search interface 139
- Adding and removing search fields in advanced search 139
- Changing the display of facets and facet groups 140
- Facilitating search scope changes 141
- Sitemap generator 141
  - Running the sitemap generator 141
  - Sitemap details 142
  - Scheduling 142

Troubleshooting TPAC errors 142
25. Ebook API integration 143
Ebook API service configuration 143
OverDrive API integration 143
OneClickdigital API integration 144
Additional configuration 145
26. Managing audio alerts 146
Globally silencing sounds 146
Self-check interface 146

Chapter 22. TPac Configuration and Customization

Template toolkit documentation

For more general information about template toolkit see: official documentation.

The purpose of this chapter is to focus on the Evergreen-specific uses of Template Toolkit (*TT*) in the OPAC.

TPAC URL

The URL for the TPAC on a default Evergreen system is http://localhost/eg/opac/home (adjust `localhost` to match your hostname or IP address, naturally!)

Perl modules used directly by TPAC

- `Open-ILS/src/perlmods/lib/OpenILS/WWW/EGCatLoader.pm`
- `Open-ILS/src/perlmods/lib/OpenILS/WWW/EGCatLoader/Account.pm`
- `Open-ILS/src/perlmods/lib/OpenILS/WWW/EGCatLoader/Container.pm`
- `Open-ILS/src/perlmods/lib/OpenILS/WWW/EGCatLoader/Record.pm`
- `Open-ILS/src/perlmods/lib/OpenILS/WWW/EGCatLoader/Search.pm`
- `Open-ILS/src/perlmods/lib/OpenILS/WWW/EGCatLoader/Util.pm`

Default templates

The source template files are found in `Open-ILS/src/templates/opac`.

These template files are installed in `/openils/var/templates/opac`.

NOTE. You should generally avoid touching the installed default template files, unless you are contributing changes that you want Evergreen to adopt as a new default. Even then, while you are developing your changes, consider using template overrides rather than touching the installed templates until you are ready to commit the changes to a branch. See below for information on template overrides.

Apache configuration files

The base Evergreen configuration file on Debian-based systems can be found in `/etc/apache2/sites-enabled/eg.conf`. This file defines the basic virtual host configuration for Evergreen (hostnames and ports), then single-sources the bulk of the configuration for each virtual host by including `/etc/apache2/eg_vhost.conf`.

TPAC CSS and media files

The CSS files used by the default TPAC templates are stored in the repo in `Open-ILS/web/css/skin/default/opac/` and installed in `/openils/var/web/css/skin/default/opac/`.

The media files—mostly PNG images—used by the default TPAC templates are stored in the repo in `Open-ILS/web/images/` and installed in `/openils/var/web/images/`.

Mapping templates to URLs

The mapping for templates to URLs is straightforward. Following are a few examples, where `<templates>` is a placeholder for one or more directories that will be searched for a match:

- `http://localhost/eg/opac/home` ⇒ `/openils/var/<templates>/opac/home.tt2`
- `http://localhost/eg/opac/advanced` ⇒ `/openils/var/<templates>/opac/advanced.tt2`
- `http://localhost/eg/opac/results` ⇒ `/openils/var/<templates>/opac/results.tt2`

The template files themselves can process, be wrapped by, or include other template files. For example, the `home.tt2` template currently involves a number of other template files to generate a single HTML file:

Example Template Toolkit file: opac/home.tt2.

```
[%  PROCESS "opac/parts/header.tt2";
    WRAPPER "opac/parts/base.tt2";
    INCLUDE "opac/parts/topnav.tt2";
    ctx.page_title = l("Home") %]
    <div id="search-wrapper">
        [% INCLUDE "opac/parts/searchbar.tt2" %]
    </div>
    <div id="content-wrapper">
        <div id="main-content-home">
            <div class="common-full-pad"></div>
            [% INCLUDE "opac/parts/homesearch.tt2" %]
            <div class="common-full-pad"></div>
        </div>
    </div>
[% END %]
```

We will dissect this example in some more detail later, but the important thing to note is that the file references are relative to the top of the template directory.

How to override templates

Overrides for templates go in a directory that parallels the structure of the default templates directory. The overrides then get pulled in via the Apache configuration.

In the following example, we demonstrate how to create a file that overrides the default "Advanced search page" (`advanced.tt2`) by adding a new templates directory and editing the new file in that directory.

Adding an override for the Advanced search page (example).

```
bash$ mkdir -p /openils/var/templates_custom/opac
bash$ cp /openils/var/templates/opac/advanced.tt2 \
         /openils/var/templates_custom/opac/.
bash$ vim /openils/var/templates_custom/opac/advanced.tt2
```

We now need to teach Apache about the new templates directory. Open `eg.conf` and add the following `<Location /eg>` element to each of the `<VirtualHost>` elements in which you want to include the overrides. The default Evergreen configuration includes a `VirtualHost` directive for port 80 (HTTP) and another one for port 443 (HTTPS); you probably want to edit both, unless you want the HTTP user experience to be different from the HTTPS user experience.

Configuring the custom templates directory in Apache's eg.conf.

```
<VirtualHost *:80>
    # <snip>

    # - absorb the shared virtual host settings
    Include eg_vhost.conf
    <Location /eg>
        PerlAddVar OILSWebTemplatePath "/openils/var/templates_algoma"
    </Location>

    # <snip>
</VirtualHost>
```

Finally, reload the Apache configuration to pick up the changes:

Reloading the Apache configuration.

```
bash# /etc/init.d/apache2 reload
```

You should now be able to see your change at http://localhost/eg/opac/advanced

Defining multiple layers of overrides

You can define multiple layers of overrides, so if you want every library in your consortium to have the same basic customizations, and then apply library-specific customizations, you can define two template directories for each library.

In the following example, we define the `template_CONS` directory as the set of customizations to apply to all libraries, and `template_BR#` as the set of customizations to apply to library BR1 and BR2.

As the consortial customizations apply to all libraries, we can add the extra template directory directly to `eg_vhost.conf`:

Apache configuration for all libraries (eg_vhost.conf).

```
# Templates will be loaded from the following paths in reverse order.
PerlAddVar OILSWebTemplatePath "/openils/var/templates"
PerlAddVar OILSWebTemplatePath "/openils/var/templates_CONS"
```

Then we define a virtual host for each library to add the second layer of customized templates on a per-library basis. Note that for the sake of brevity we only show the configuration for port 80.

Apache configuration for each virtual host (eg.conf).

```
<VirtualHost *:80>
    ServerName br1.concat.ca
    DocumentRoot /openils/var/web/
    DirectoryIndex index.html index.xhtml
    Include eg_vhost.conf
    <Location /eg>
        PerlAddVar OILSWebTemplatePath "/openils/var/templates_BR1"
    </Location>
</VirtualHost>

<VirtualHost *:80>
    ServerName br2.concat.ca
    DocumentRoot /openils/var/web/
    DirectoryIndex index.html index.xhtml
    Include eg_vhost.conf
    <Location /eg>
        PerlAddVar OILSWebTemplatePath "/openils/var/templates_BR2"
    </Location>
</VirtualHost>
```

Changing some text in the TPAC

Out of the box, the TPAC includes a number of placeholder text and links. For example, there is a set of links cleverly named *Link 1*, *Link 2*, and so on in the header and footer of every page in the TPAC. Let's customize that for our `templates_BR1` skin.

To begin with, we need to find the page(s) that contain the text in question. The simplest way to do that is with the handy utility `ack`, which is much like `grep` but with built-in recursion and other tricks. On Debian-based systems, the command is `ack-grep` as `ack` conflicts with an existing utility. In the following example, we search for files that contain the text "Link 1":

Searching for text matching "Link 1".

```
bash$ ack-grep "Link 1" /openils/var/templates/opac
/openils/var/templates/opac/parts/topnav_links.tt2
4:            <a href="http://example.com">[% l('Link 1') %]</a>
```

Next, we copy the file into our overrides directory and edit it with `vim`:

Copying the links file into the overrides directory.

```
bash$ cp /openils/var/templates/opac/parts/topnav_links.tt2 \
         /openils/var/templates_BR1/opac/parts/topnav_links.tt2
bash$ vim /openils/var/templates_BR1/opac/parts/topnav_links.tt2
```

Finally, we edit the link text in `opac/parts/header.tt2`.

Content of the opac/parts/header.tt2 file.

```
<div id="gold-links-holder">
    <div id="gold-links">
        <div id="header-links">
            <a href="http://example.com">[% l('Link 1') %]</a>
            <a href="http://example.com">[% l('Link 2') %]</a>
            <a href="http://example.com">[% l('Link 3') %]</a>
            <a href="http://example.com">[% l('Link 4') %]</a>
            <a href="http://example.com">[% l('Link 5') %]</a>
        </div>
    </div>
</div>
```

For the most part, the page looks like regular HTML, but note the `[%_(" ")%]` that surrounds the text of each link. The `[% ... %]` signifies a TT block, which can contain one or more TT processing instructions. `l(" ...`

`");` is a function that marks text for localization (translation); a separate process can subsequently extract localized text as GNU gettext-formatted PO files.

NOTE. As Evergreen supports multiple languages, any customizations to Evergreen's default text must use the localization function. Also, note that the localization function supports placeholders such as `[_1]`, `[_2]` in the text; these are replaced by the contents of variables passed as extra arguments to the `l()` function.

Once we have edited the link and link text to our satisfaction, we can load the page in our Web browser and see the live changes immediately (assuming we are looking at the BR1 overrides, of course).

Troubleshooting

If there is a problem such as a TT syntax error, it generally shows up as a an ugly server failure page. If you check the Apache error logs, you will probably find some solid clues about the reason for the failure. For example, in the following example the error message identifies the file in which the problem occurred as well as the relevant line numbers:

Example error message in Apache error logs.

```
bash# grep "template error" /var/log/apache2/error_log
[Tue Dec 06 02:12:09 2011] [warn] [client 127.0.0.1] egweb: template error:
  file error - parse error - opac/parts/record/summary.tt2 line 112-121:
  unexpected token (!=)\n  [% last_cn = 0;\n        FOR copy_info IN
  ctx.copies;\n            callnum = copy_info.call_number_label;\n
```

Chapter 23. Designing your catalog

When people want to find things in your Evergreen system, they will check the catalog. In Evergreen, the catalog is made available through a web interface, called the *OPAC* (Online Public Access Catalog). In the latest versions of the Evergreen system, the OPAC is built on a set of programming modules called the Template Toolkit. You will see the OPAC sometimes referred to as the *TPAC*.

In this chapter, we'll show you how to customize the OPAC, change it from its default configuration, and make it your own.

Configuring and customizing the public interface

The public interface is referred to as the TPAC or Template Toolkit (TT) within the Evergreen community. The template toolkit system allows you to customize the look and feel of your OPAC by editing the template pages (.tt2) files as well as the associated style sheets.

Locating the default template files

The default URL for the TPAC on a default Evergreen system is *http://localhost/eg/opac/home* (adjust *localhost* to match your hostname or IP address).

The default template file is installed in */openils/var/templates/opac*.

You should generally avoid touching the installed default template files, unless you are contributing changes for Evergreen to adopt as a new default. Even then, while you are developing your changes, consider using template overrides rather than touching the installed templates until you are ready to commit the changes to a branch. See below for information on template overrides.

Mapping templates to URLs

The mapping for templates to URLs is straightforward. Following are a few examples, where <*templates*> is a placeholder for one or more directories that will be searched for a match:

- *http://localhost/eg/opac/home # /openils/var/<templates>/opac/home.tt2*
- *http://localhost/eg/opac/advanced # /openils/var/<templates>/opac/advanced.tt2*
- *http://localhost/eg/opac/results # /openils/var/<templates>/opac/results.tt2*

The template files themselves can process, be wrapped by, or include other template files. For example, the *home.tt2* template currently involves a number of other template files to generate a single HTML file.

Example Template Toolkit file: *opac/home.tt2*.

```
[%  PROCESS "opac/parts/header.tt2";
    WRAPPER "opac/parts/base.tt2";
    INCLUDE "opac/parts/topnav.tt2";
    ctx.page_title = l("Home") %]
    <div id="search-wrapper">
      [% INCLUDE "opac/parts/searchbar.tt2" %]
    </div>
    <div id="content-wrapper">
        <div id="main-content-home">
             <div class="common-full-pad"></div>
             [% INCLUDE "opac/parts/homesearch.tt2" %]
             <div class="common-full-pad"></div>
        </div>
     </div>
[% END %]
```

Note that file references are relative to the top of the template directory.

How to override template files

Overrides for template files or TPAC pages go in a directory that parallels the structure of the default templates directory. The overrides then get pulled in via the Apache configuration.

The following example demonstrates how to create a file that overrides the default "Advanced search page" (*advanced.tt2*) by adding a new *templates_custom* directory and editing the new file in that directory.

```
bash$ mkdir -p /openils/var/templates_custom/opac
bash$ cp /openils/var/templates/opac/advanced.tt2 \
         /openils/var/templates_custom/opac/.
bash$ vim /openils/var/templates_custom/opac/advanced.tt2
```

Configuring the custom templates directory in Apache's eg.conf

You now need to teach Apache about the new custom template directory. Edit */etc/apache2/sites-available/eg.conf* and add the following <*Location /eg*> element to each of the <*VirtualHost*> elements in which you want to include the overrides. The default Evergreen configuration includes a VirtualHost directive for port 80 (HTTP) and another one for port 443 (HTTPS); you probably want to edit both, unless you want the HTTP user experience to be different from the HTTPS user experience.

```
<VirtualHost *:80>
    # <snip>

    # - absorb the shared virtual host settings
    Include eg_vhost.conf
    <Location /eg>
        PerlAddVar OILSWebTemplatePath "/openils/var/templates_custom"
    </Location>

    # <snip>
</VirtualHost>
```

Finally, reload the Apache configuration to pick up the changes. You should now be able to see your change at *http://localhost/eg/opac/advanced* where *localhost* is the hostname of your Evergreen server.

Adjusting colors for your public interface

You may adjust the colors of your public interface by editing the *colors.tt2* file. The location of this file is in */openils/var/templates/opac/parts/css/colors.tt2*. When you customize the colors of your public interface, remember to create a custom file in your custom template folder and edit the custom file and not the file located in you default template.

Adjusting fonts in your public interface

Font sizes can be changed in the *colors.tt2* file located in */openils/var/templates/opac/parts/css/*. Again, create and edit a custom template version and not the file in the default template.

Other aspects of fonts such as the default font family can be adjusted in */openils/var/templates/opac/css/style.css.tt2*.

Media file locations in the public interface

The media files (mostly PNG images) used by the default TPAC templates are stored in the repository in *Open-ILS/web/images/* and installed in */openils/var/web/images/*.

Changing some text in the public interface

Out of the box, TPAC includes a number of placeholder text and links. For example, there is a set of links cleverly named Link 1, Link 2, and so on in the header and footer of every page in TPAC. Here is how to customize that for a *custom templates* skin.

To begin with, find the page(s) that contain the text in question. The simplest way to do that is with the grep -s command. In the following example, search for files that contain the text "Link 1":

```
bash$ grep -r "Link 1" /openils/var/templates/opac
/openils/var/templates/opac/parts/topnav_links.tt2
4:            <a href="http://example.com">[% l('Link 1') %]</a>
```

Next, copy the file into our overrides directory and edit it with vim.

Copying the links file into the overrides directory.

```
bash$ cp /openils/var/templates/opac/parts/topnav_links.tt2 \
/openils/var/templates_custom/opac/parts/topnav_links.tt2
bash$ vim /openils/var/templates_custom/opac/parts/topnav_links.tt2
```

Finally, edit the link text in *opac/parts/header.tt2*. Content of the *opac/parts/header.tt2* file.

```
<div id="gold-links-holder">
    <div id="gold-links">
        <div id="header-links">
            <a href="http://example.com">[% l('Link 1') %]</a>
            <a href="http://example.com">[% l('Link 2') %]</a>
            <a href="http://example.com">[% l('Link 3') %]</a>
            <a href="http://example.com">[% l('Link 4') %]</a>
            <a href="http://example.com">[% l('Link 5') %]</a>
        </div>
    </div>
</div>
```

For the most part, the page looks like regular HTML, but note the `[%_(" ")%]` that surrounds the text of each link. The `[% ... %]` signifies a TT block, which can contain one or more TT processing instructions. `l(" ... ");` is a function that marks text for localization (translation); a separate process can subsequently extract localized text as GNU gettext-formatted PO (Portable Object) files.

As Evergreen supports multiple languages, any customization to Evergreen's default text must use the localization function. Also, note that the localization function supports placeholders such as `[_1]`, `[_2]` in the text; these are replaced by the contents of variables passed as extra arguments to the `l()` function.

Once the link and link text has been edited to your satisfaction, load the page in a Web browser and see the live changes immediately.

Adding translations to PO file

After you have added custom text in translatable form to a TT2 template, you need to add the custom strings and its translations to the PO file containing the translations. Evergreen PO files are stored in */openils/var/template/data/locale/*

The PO file consists of pairs of the text extracted from the code: Message ID denoted as *msgid* and message string denoted as *msgstr*. When adding the custom string to PO file:

- The line with English expressions must start with *msgid*. The English term must be enclosed in double apostrophes.
- The line with translation start with /msgstr/. The translation to local language must be and enclosed in enclosed in double apostrophes.
- It is recommended to add a note in which template and on which line the particular string is located. The lines with notes must be marked as comments i.e., start with number sign (#)

Example:

```
# ---------------------------------------------------------------------
# The lines below contains the custom strings manually added to the catalog
# ---------------------------------------------------------------------

#: ../../Open-ILS/src/custom_templates/opac/parts/topnav_links.tt2:1
msgid "Union Catalog of the Czech Republic"
msgstr "Souborný katalog #eské republiky"


#: ../../Open-ILS/src/custom_templates/opac/parts/topnav_links.tt2:1
msgid "Uniform Information Gateway "
msgstr "Jednotná informa#ní brána"
```

It is good practice to save backup copy of the original PO file before changing it.

After making changes, restart Apache to make the changes take effect. As root run the command:

```
service apache2 restart
```

Adding and removing MARC fields from the record details display page

It is possible to add and remove the MARC fields and subfields displayed in the record details page. In order to add MARC fields to be displayed on the details page of a record, you will need to map the MARC code to variables in the */openils/var/templates/opac/parts/misc_util.tt2 file*.

For example, to map the template variable *args.pubdates* to the date of publication MARC field 260, subfield c, add these lines to *misc_util.tt2*:

```
args.pubdates = [];
FOR sub IN xml.findnodes('//*[@tag="260"]/*[@code="c"]');
    args.pubdates.push(sub.textContent);
END;
args.pubdate = (args.pubdates.size) ? args.pubdates.0 : ''
```

You will then need to edit the */openils/var/templates/opac/parts/record/summary.tt2* file in order to get the template variable for the MARC field to display.

For example, to display the date of publication code you created in the *misc_util.tt2* file, add these lines:

```
[% IF attrs.pubdate; %]
    <span itemprop="datePublished">[% attrs.pubdate | html; %]</span>
[% END; %]
```

You can add any MARC field to your record details page. Moreover, this approach can also be used to display MARC fields in other pages, such as your results page.

Using bibliographic source variables

For bibliographic records, there is a "bib source" that can be associated with every record. This source and its ID are available as record attributes called *bib_source.source* and *bib_source.id*. These variables do not present themselves in the catalog display by default.

Example use case

In this example, a library imports e-resource records from a third party and uses the bib source to indicate where the records came from. Patrons can place holds on these titles, but they must be placed via the vendor website, not in Evergreen. By exposing the bib source, the library can alter the Place Hold link for these records to point at the vendor website.

Setting the default physical location for your library environment

physical_loc is an Apache environment variable that sets the default physical location, used for setting search scopes and determining the order in which copies should be sorted. This variable is set in */etc/apache2/sites-available/eg.conf*. The following example demonstrates the default physical location being set to library ID 104:

```
SetEnv physical_loc 104
```

Setting a default language and adding optional languages

OILSWebLocale adds support for a specific language. Add this variable to the Virtual Host section in */etc/apache2/eg_vhost.conf*.

OILSWebDefaultLocale specifies which locale to display when a user lands on a page in TPAC and has not chosen a different locale from the TPAC locale picker. The following example shows the *fr_ca* locale being added to the locale picker and being set as the default locale:

```
PerlAddVar OILSWebLocale "fr_ca"
PerlAddVar OILSWebLocale "/openils/var/data/locale/opac/fr-CA.po"
PerlAddVar OILSWebDefaultLocale "fr-CA"
```

Below is a table of the currently supported languages packaged with Evergreen:

Language	Code	PO file
Arabic - Jordan	ar_jo	/openils/var/data/locale/opac/ar-JO.po
Armenian	hy_am	/openils/var/data/locale/opac/hy-AM.po
Czech	cs_cz	/openils/var/data/locale/opac/cs-CZ.po
English - Canada	en_ca	/openils/var/data/locale/opac/en-CA.po
English - Great Britain	en_gb	/openils/var/data/locale/opac/en-GB.po
*English - United States	en_us	not applicable
French - Canada	fr_ca	/openils/var/data/locale/opac/fr-CA.po
Portuguese - Brazil	pt_br	/openils/var/data/locale/opac/pt-BR.po
Spanish	es_es	/openils/var/data/locale/opac/es-ES.po

*American English is built into Evergreen so you do not need to set up this language and there are no PO files.

Updating translations in Evergreen using current translations from Launchpad

Due to Evergreen release workflow/schedule, some language strings may already have been translated in Launchpad, but are not yet packaged with Evergreen. In such cases, it is possible to manually replace the PO file in Evergreen with an up-to-date PO file downloaded from Launchpad.

1. Visit the Evergreen translation site in Launchpad
2. Select required language (e.g. *Czech* or *Spanish*)
3. Open the *tpac* template and then select option *Download translation*. Note: to be able to download the translation file you need to be logged in to Launchpad.
4. Select *PO format* and submit the *request for download* button. You can also request for download of all existing templates and languages at once, see https://translations.launchpad.net/evergreen/master/+export. The download link will be sent You to email address provided.
5. Download the file and name it according to the language used (e.g., *cs-CZ.po* for Czech or *es-ES.po* for Spanish)
6. Copy the downloaded file to */openils/var/template/data/locale*. It is a good practice to backup the original PO file before.
7. Be sure that the desired language is set as default, using the Default language procedures.

Analogously, to update the web staff client translations, download the translation template *webstaff* and copy it to *openils/var/template/data/locale/staff*.

Changes require web server reload to take effect. As root run the command

```
service apache2 restart
```

Change Date Format in Patron Account View

Libraries with same-day circulations may want their patrons to be able to view the due **time** as well as due date when they log in to their OPAC account. To accomplish this, go to *opac/myopac/circs.tt2*. Find the line that reads:

```
[% date.format(due_date, DATE_FORMAT) %]
```

Replace it with:

```
[% date.format(due_date, '%D %I:%M %p') %]
```

Including External Content in Your Public Interface

The public interface allows you to include external services and content in your public interface. These can include book cover images, user reviews, table of contents, summaries, author notes, annotations, user suggestions, series information among other services. Some of these services are free while others require a subscription.

The following are some of the external content services which you can configure in Evergreen.

OpenLibrary

The default install of Evergreen includes OpenLibrary book covers. The settings for this are controlled by the <added_content> section of */openils/conf/opensrf.xml*. Here are the key elements of this configuration:

```
<module>OpenILS::WWW::AddedContent::OpenLibrary</module>
```

This section calls the OpenLibrary perl module. If you wish to link to a different book cover service other than OpenLibrary, you must refer to the location of the corresponding Perl module. You will also need to change other settings accordingly.

```
<timeout>1</timeout>
```

Max number of seconds to wait for an added content request to return data. Data not returned within the timeout is considered a failure.

```
<retry_timeout>600</retry_timeout>
```

This setting is the amount of time to wait before we try again.

```
<max_errors>15</max_errors>
```

Maximum number of consecutive lookup errors a given process can have before added content lookups are disabled for everyone. To adjust the site of the cover image on the record details page edit the config.tt2 file and change the value of the record.summary.jacket_size. The default value is "medium" and the available options are "small", "medium" and "large."

ChiliFresh

ChiliFresh is a subscription-based service which allows book covers, reviews and social interaction of patrons to appear in your catalog. To activate ChiliFresh, you will need to open the Apache configuration file */etc/apache2/eg_vhost.conf* and edit several lines:

1. Uncomment (remove the "#" at the beginning of the line) and add your ChiliFresh account number:

```
#SetEnv OILS_CHILIFRESH_ACCOUNT
```

1. Uncomment this line and add your ChiliFresh Profile:

```
#SetEnv OILS_CHILIFRESH_PROFILE
```

Uncomment the line indicating the location of the Evergreen JavaScript for ChiliFresh:

```
#SetEnv OILS_CHILIFRESH_URL http://chilifresh.com/on-site /js/evergreen.js
```

1. Uncomment the line indicating the secure URL for the Evergreen JavaScript :

```
#SetEnv OILS_CHILIFRESH_HTTPS_URL https://secure.chilifresh.com/on-site/js/evergreen.js
```

Content Café

Content Café is a subscription-based service that can add jacket images, reviews, summaries, tables of contents and book details to your records.

In order to activate Content Café, edit the */openils/conf/opensrf.xml* file and change the <*module*> element to point to the ContentCafe Perl Module:

```
<module>OpenILS::WWW::AddedContent::ContentCafe</module>
```

To adjust settings for Content Café, edit a couple of fields with the <*ContentCafe*> Section of */openils/conf/opensrf.xml*.

Edit the *userid* and *password* elements to match the user id and password for your Content Café account.

This provider retrieves content based on ISBN or UPC, with a default preference for ISBNs. If you wish for UPCs to be preferred, or wish one of the two identifier types to not be considered at all, you can change the "identifier_order" option in opensrf.xml. When the option is present, only the identifier(s) listed will be sent.

Obalkyknih.cz

Setting up Obalkyknih.cz account

If your library wishes to use added content provided by Obalkyknih.cz, a service based in the Czech Republic, you have to create an Obalkyknih.cz account. Please note that the interface is only available in Czech. After logging in your Obalkyknih.cz account, you have to add your IP address and Evergreen server address to your account settings. (In case each library uses an address of its own, all of these addresses have to be added.)

Enabling Obalkyknih.cz in Evergreen

Set obalkyknih_cz.enabled to true in */openils/var/templates/opac/parts/config.tt2*:

```
obalkyknih_cz.enabled = 'true';
```

Enable added content from Obalkyknih.cz in */openils/conf/opensrf.xml* configuration file (and – at the same time – disable added content from Open Library, i.e., Evergreen's default added content provider):

```
<!-- <module>OpenILS::WWW::AddedContent::OpenLibrary</module> -->
<module>OpenILS::WWW::AddedContent::ObalkyKnih</module>
```

Using default settings for Obalkyknih.cz means all types of added content from Obalkyknih.cz are visible in your online catalog. If the module is enabled, book covers are always displayed. Other types of added content (summaries, ratings or tables of contents) can be:

- switched off using *false* option,
- switched on again using *true* option.

The following types of added content are used:

- summary (or annotation)
- tocPDF (table of contents available as image)
- tocText (table of contents available as text)
- review (user reviews)

An example of how to switch off summaries:

```
<summary>false</summary>
```

Google Analytics

Google Analytics is a free service to collect statistics for your Evergreen site. Statistic tracking is disabled by default through the Evergreen client software when library staff use your site within the client, but active when anyone uses the site without the client. This was a preventive measure to reduce the potential risks for leaking patron information. In order to use Google Analytics you will first need to set up the service from the Google Analytics website at http://www.google.com/analytics/. To activate Google Analytics you will need to edit *config.tt2* in your template. To enable the service set the value of google_analytics.enabled to true and change the value of *google_analytics.code* to be the code in your Google Analytics account.

NoveList

Novelist is a subscription-based service providing reviews and recommendation for books in you catalog. To activate your Novelist service in Evergreen, open the Apache configuration file */etc/apache2/eg_vhost.conf* and edit the line:

```
#SetEnv OILS_NOVELIST_URL
```

You should use the URL provided by NoveList.

RefWorks

RefWorks is a subscription-based online bibliographic management tool. If you have a RefWorks subscription, you can activate RefWorks in Evergreen by editing the *config.tt2* file located in your template directory. You will need to set the *ctx.refworks.enabled* value to *true*. You may also set the RefWorks URL by changing the *ctx.refworks.url* setting on the same file.

SFX OpenURL Resolver

An OpenURL resolver allows you to find electronic resources and pull them into your catalog based on the ISBN or ISSN of the item. In order to use the SFX OpenURL resolver, you will need to subscribe to the Ex Libris SFX service. To activate the service in Evergreen edit the *config.tt2* file in your template. Enable the resolver by changing the value of *openurl.enabled* to *true* and change the *openurl.baseurl* setting to point to the URL of your OpenURL resolver.

Syndetic Solutions

Syndetic Solutions is a subscription service providing book covers and other data for items in your catalog. In order to activate Syndetic, edit the */openils/conf/opensrf.xml* file and change the <*module*> element to point to the Syndetic Perl Module:

```
<module>OpenILS::WWW::AddedContent::Syndetic</module>
```

You will also need to edit the *<userid>* element to be the user id provided to you by Syndetic.

Then, you will need to uncomment and edit the *<base_url>* element so that it points to the Syndetic service:

```
<base_url>http://syndetics.com/index.aspx</base_url>
```

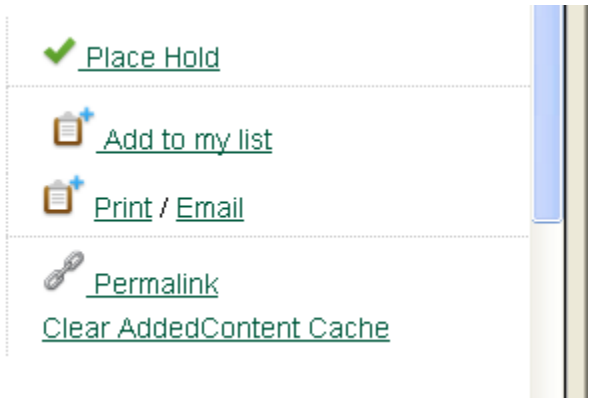
For changes to be activated for your public interface you will need to restart Evergreen and Apache.

The Syndetic Solutions provider retrieves images based on the following identifiers found in bibliographic records:

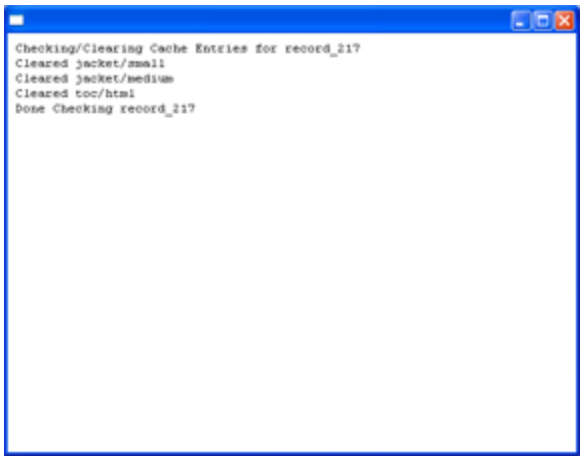
- ISBN
- UPC
- ISSN

Clear External/Added Content Cache

On the catalog's record summary page, there is a link for staff that will forcibly clear the cache of the Added Content for that record. This is helpful for when the Added Content retrieved the wrong cover jacket art, summary, etc. and caches the wrong result.



Once clicked, there is a pop up that will display what was cleared from the cache.



You will need to reload the record in the staff client to obtain the new images from your Added Content Supplier.

Configure a Custom Image for Missing Images

You can configure a "no image" image other than the standard 1-pixel blank image. The example eg_vhost.conf file provides examples in the comments. Note: Evergreen does not provide default images for these.

Including Locally Hosted Content in Your Public Interface

It is also possible to show added content that has been generated locally by placing the content in a specific spot on the web server. It is possible to have local book jackets, reviews, TOC, excerpts or annotations.

File Location and Format

By default the files will need to be placed in directories under **/openils/var/web/opac/extras/ac/** on the server(s) that run Apache.

The files need to be in specific folders depending on the format of the added content. Local Content can only be looked up based on the record ID at this time.

URL Format: http://catalog/opac/extras/ac/**{type}/{format}/r/{recordid}**

- **type** is one of **jacket**, **reviews**, **toc**, **excerpt** or **anotes**.
- **format** is type dependent:
  - for jacket, one of small, medium or large
  - others, one of html, xml or json … html is the default for non-image added content
- **recordid** is the bibliographic record id (bre.id).

Example

If you have some equipment that you are circulating such as a laptop or eBook reader and you want to add an image of the equipment that will show up in the catalog.

If you are adding jacket art for a traditional type of media (book, CD, DVD) consider adding the jacket art to the http://openlibrary.org project instead of hosting it locally. This would allow other libraries to benefit from your work.

Make note of the Record ID of the bib record. You can find this by looking at the URL of the bib in the catalog. http://catalog/eg/opac/record/**123**, 123 is the record ID. These images will only show up for one specific record.

Create 3 different sized versions of the image in png or jpg format.

- **Small** - 80px x 80px - named *123-s.jpg* or *123-s.png* - This is displayed in the browse display.
- **Medium** - 240px x 240px - named *123-m.jpg* or *123-m.png* - This is displayed on the summary page.

- **Large** - 400px x 399px - named *123-l.jpg* or *123-l.png* - This is displayed if the summary page image is clicked on.

The image dimensions are up to you, use what looks good in your catalog.

Next, upload the images to the evergreen server(s) that run apache, and move/rename the files to the following locations/name. You will need to create directories that are missing.

- Small - Move the file **123-s.jpg** to **/openils/var/web/opac/extras/ac/jacket/small/r/123**
- Medium - Move the file **123-m.jpg** to **/openils/var/web/opac/extras/ac/jacket/medium/r/123**.
- Large - Move the file **123-l.jpg** to **/openils/var/web/opac/extras/ac/jacket/large/r/123**.

The system doesn't need the file extension to know what kind of file it is.

Reload the bib record summary in the web catalog and your new image will display.

Styling the searchbar on the homepage

The `.searchbar-home` class is added to the div that contains the searchbar when on the homepage. This allows sites to customize the searchbar differently on the homepage than in search results pages, and other places the search bar appears. For example, adding the following CSS would create a large, Google-style search bar on the homepage only:

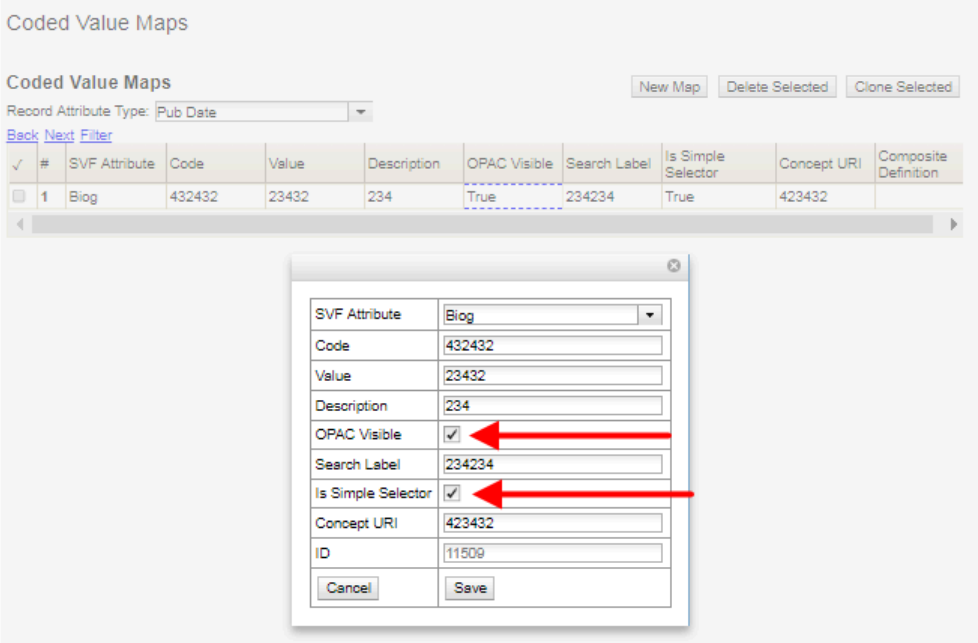
```
.searchbar-home .search-box {
  width: 80%;
  height: 3em;
}

.searchbar-home #search_qtype_label,
.searchbar-home #search_itype_label,
.searchbar-home #search_locg_label {
  display:none;
}
```

Chapter 24. Designing the patron search experience

Editing the formats select box options in the search interface

You may wish to remove, rename or organize the options in the formats select box. This can be accomplished from the staff client.

1. From the staff client, navigate to **Administration** → **Server Administration** → **Marc Coded Value Maps**
2. Select *Type* from the **Record Attribute Type** select box.
3. Double click on the format type you wish to edit.



To change the label for the type, enter a value in the **Search Label** field.

To move the option to a top list separated by a dashed line from the others, check the **Is Simple Selector** check box.

To hide the type so that it does not appear in the search interface, uncheck the **OPAC Visible** checkbox.

Changes will be immediate.

Adding and removing search fields in advanced search

It is possible to add and remove search fields on the advanced search page by editing the *opac/parts/config.tt2* file in your template directory. Look for this section of the file:

```
search.adv_config = [
    {adv_label => l("Item Type"), adv_attr => ["mattype", "item_type"]},
    {adv_label => l("Item Form"), adv_attr => "item_form"},
    {adv_label => l("Language"),  adv_attr => "item_lang"},
    {adv_label => l("Audience"),  adv_attr => ["audience_group", "audience"], adv_break => 1},
    {adv_label => l("Video Format"), adv_attr => "vr_format"},
    {adv_label => l("Bib Level"), adv_attr => "bib_level"},
    {adv_label => l("Literary Form"), adv_attr => "lit_form", adv_break => 1},
    {adv_label => l("Search Library"), adv_special => "lib_selector"},
    {adv_label => l("Publication Year"), adv_special => "pub_year"},
    {adv_label => l("Sort Results"), adv_special => "sort_selector"},
];
```

For example, if you delete the line:

```
{adv_label => l("Language"),  adv_attr => "item_lang"},
```

the language field will no longer appear on your advanced search page. Changes will appear immediately after you save your changes.

You can also add fields based on Search Facet Groups that you create in the staff client's Local Administration menu. This can be helpful if you want to simplify your patrons' experience by presenting them with only certain limiters (e.g. the most commonly used languages in your area). To do this,

1. Click **Administration** → **Local Administration** → **Search Filter Groups**.
2. Click **New**.
3. Enter descriptive values into the code and label fields. The owner needs to be set to your consortium.
4. Once the Facet Group is created, click on the blue hyperlinked code value.
5. Click the **New** button to create the necessary values for your field.
6. Go to the *opac/parts/config.tt2* file, and add a line like the following, where **Our Library's Field** is the name you'd like to be displayed next to your field, and **facet_group_code** is the code you've added using the staff client.

   ```
   {adv_label => l("Our Library's Field"), adv_filter => "facet_group_code"},
   ```

Changing the display of facets and facet groups

Facets can be reordered on the search results page by editing the *opac/parts/config.tt2* file in your template directory.

Edit the following section of *config.tt2*, changing the order of the facet categories according to your needs:

```
facet.display = [
    {facet_class => 'author',  facet_order => ['personal', 'corporate']},
    {facet_class => 'subject', facet_order => ['topic']},
    {facet_class => 'series',  facet_order => ['seriestitle']},
    {facet_class => 'subject', facet_order => ['name', 'geographic']}
];
```

You may also change the default number of facets appearing under each category by editing the *facet.default_display_count* value in *config.tt2*. The default value is 5.

Facilitating search scope changes

Users often search in a limited scope, such as only searching items in their local library. When they aren't able find materials that meet their needs in a limited scope, they may wish to repeat their search in a system-wide or consortium-wide scope. Evergreen provides an optional button and checkbox to alter the depth of the search to a defined level.

The button and checkbox are both enabled by default and can be configured in the Depth Button/Checkbox section of config.tt2.

Noteworthy settings related to these features include:

- `ctx.depth_sel_checkbox` — set this to 1 to display the checkbox, 0 to hide it.
- `ctx.depth_sel_button` — set this to 1 to display the button, 0 to hide it.
- `ctx.depth_sel_depth` — the depth that should be applied by the button and checkbox. A value of 0 would typically search the entire consortium, and 1 would typically search the library's system.

Sitemap generator

A sitemap directs search engines to the pages of interest in a web site so that the search engines can intelligently crawl your site. In the case of Evergreen, the primary pages of interest are the bibliographic record detail pages.

The sitemap generator script creates sitemaps that adhere to the http://sitemaps.org specification, including:

- limiting the number of URLs per sitemap file to no more than 50,000 URLs;
- providing the date that the bibliographic record was last edited, so that once a search engine has crawled all of your sites' record detail pages, it only has to reindex those pages that are new or have changed since the last crawl;
- generating a sitemap index file that points to each of the sitemap files.

Running the sitemap generator

The `sitemap_generator` script must be invoked with the following argument:

- `--lib-hostname`: specifies the hostname for the catalog (for example, `--lib-hostname https://catalog.example.com`); all URLs will be generated appended to this hostname

Therefore, the following arguments are useful for generating multiple sitemaps per Evergreen instance:

- `--lib-shortname`: limit the list of record URLs to those which have copies owned by the designated library or any of its children;
- `--prefix`: provides a prefix for the sitemap index file names

Other options enable you to override the OpenSRF configuration file and the database connection credentials, but the default settings are generally fine.

Note that on very large Evergreen instances, sitemaps can consume hundreds of megabytes of disk space, so ensure that your Evergreen instance has enough room before running the script.

Sitemap details

The sitemap generator script includes located URIs as well as items listed in the `asset.opac_visible_copies` materialized view, and checks the children or ancestors of the requested libraries for holdings as well.

Scheduling

To enable search engines to maintain a fresh index of your bibliographic records, you may want to include the script in your cron jobs on a nightly or weekly basis.

Sitemap files are generated in the same directory from which the script is invoked, so a cron entry will look something like:

```
12 2 * * * cd /openils/var/web && /openils/bin/sitemap_generator
```

Troubleshooting TPAC errors

If there is a problem such as a TT syntax error, it generally shows up as an ugly server failure page. If you check the Apache error logs, you will probably find some solid clues about the reason for the failure. For example, in the following example, the error message identifies the file in which the problem occurred as well as the relevant line numbers.

Example error message in Apache error logs:

```
bash# grep "template error" /var/log/apache2/error_log
[Tue Dec 06 02:12:09 2011] [warn] [client 127.0.0.1] egweb: template error:
 file error - parse error - opac/parts/record/summary.tt2 line 112-121:
  unexpected token (!=)\n  [% last_cn = 0;\n        FOR copy_info IN
  ctx.copies;\n            callnum = copy_info.call_number_label;\n
```

Chapter 25. Ebook API integration

Evergreen supports integration with third-party APIs provided by OverDrive and OneClickdigital.

When ebook API integration is enabled, the following features are supported:

- Bibliographic records from these vendors that appear in your public catalog will include vendor holdings and availability information.
- Patrons can check out and place holds on OverDrive and OneClickdigital ebook titles from within the public catalog.
- When a user is logged in, the public catalog dashboard and My Account interface will include information about that user's checkouts and holds for supported vendors.

The ability to check out and place holds on ebook titles is an experimental feature in 3.0. It is not recommended for production use without careful testing.

For API integration to work, you need to request API access from the vendor and configure your Evergreen system according to the instructions below. You also need to configure the new `open-ils.ebook_api` service.

This feature assumes that you are importing MARC records supplied by the vendor into your Evergreen system, using Vandelay or some other MARC import method. This feature does not search the vendor's online collections or automatically import vendor records into your system; it merely augments records that are already in Evergreen.

A future Evergreen release will add the ability for users to check out titles, place holds, etc., directly via the public catalog.

Ebook API service configuration

This feature uses the new `open-ils.ebook_api` OpenSRF service. This service must be configured in your `opensrf.xml` and `opensrf_core.xml` config files for ebook API integration to work. See `opensrf.xml.example` and `opensrf_core.xml.example` for guidance.

OverDrive API integration

Before enabling OverDrive API integration, you will need to request API access from OverDrive. OverDrive will provide the values to be used for the following new org unit settings:

- **OverDrive Basic Token**: The basic token used for API client authentication. To generate your basic token, combine your client key and client secret provided by OverDrive into a single string ("key:secret"), and then base64-encode that string. On Linux, you can use the following command: `echo -n "key:secret" | base64 -`
- **OverDrive Account ID**: The account ID (a.k.a. library ID) for your OverDrive API account.
- **OverDrive Website ID**: The website ID for your OverDrive API account.

- **OverDrive Authorization Name**: The authorization name (a.k.a. library name) designated by OverDrive for your library. If your OverDrive subscription includes multiple Evergreen libraries, you will need to add a separate value for this setting for each participating library.

- **OverDrive Password Required**: If your library's OverDrive subscription requires the patron's PIN (password) to be provided during patron authentication, set this setting to "true." If you do not require the patron's PIN for OverDrive authentication, set this setting to "false." (If set to "true," the password entered by a patron when logging into the public catalog will be cached in plain text in memcached.)

- **OverDrive Discovery API Base URI** and **OverDrive Circulation API Base URI**: By default, Evergreen uses OverDrive's production API, so you should not need to set a value for these settings. If you want to use OverDrive's integration environment, you will need to add the appropriate base URIs for the discovery and circulation APIs. See OverDrive's developer documentation for details.

- **OverDrive Granted Authorization Redirect URI**: Evergreen does not currently support granted authorization with OverDrive, so this setting is not currently in use.

For more information, consult the OverDrive API documentation.

To enable OverDrive API integration, adjust the following public catalog settings in `config.tt2`:

- `ebook_api.enabled`: set to "true".

- `ebook_api.overdrive.enabled`: set to "true".

- `ebook_api.overdrive.base_uris`: list of regular expressions matching OverDrive URLs found in the 856$9 field of older OverDrive MARC records. As of fall 2016, OverDrive's URL format has changed, and the record identifier is now found in the 037$a field of their MARC records, with "OverDrive" in 037$b. Evergreen will check the 037 field for OverDrive record identifiers; if your system includes older-style OverDrive records with the record identifier embedded in the 856 URL, you need to specify URL patterns with this setting.

OneClickdigital API integration

Before enabling OneClickdigital API integration, you will need to request API access from OneClickdigital. OneClickdigital will provide the values to be used for the following new org unit settings:

- **OneClickdigital Library ID**: The identifier assigned to your library by OneClickdigital.

- **OneClickdigital Basic Token**: Your client authentication token, supplied by OneClickdigital when you request access to their API.

For more information, consult the OneClickdigital API documentation.

To enable OneClickdigital API integration, adjust the following public catalog settings in `config.tt2`:

- `ebook_api.enabled`: set to "true".

- `ebook_api.oneclickdigital.enabled`: set to "true".

- `ebook_api.oneclickdigital.base_uris`: list of regular expressions matching OneClickdigital URLs found in the 859$9 field of your MARC records. Evergreen uses the patterns specified here to extract record identifiers for OneClickdigital titles.

Additional configuration

Evergreen communicates with third-party vendor APIs using the new `OpenILS::Utils::HTTPClient` module. This module is configured using settings in `opensrf.xml`. The default settings should work for most environments by default, but you may need to specify a custom location for the CA certificates installed on your server. You can also disable SSL certificate verification on HTTPClient requests altogether, but doing so is emphatically discouraged.

Chapter 26. Managing audio alerts

Globally silencing sounds

The file `nosound.wav` can be used to globally disable audio alerts for a specific event on an Evergreen system.

For example, to silence the alert that sounds after a successful patron search:

```
mkdir -p /openils/var/web/audio/notifications/success/patron/
cd /openils/var/web/audio/notifications/success/patron/
ln -s ../../nosound.wav by_search.wav
```

Self-check interface

Sounds may play at certain events in the self check interface. These events are defined in the `templates/circ/selfcheck/audio_config.tt2` template. To use the default sounds, you could run the following command from your Evergreen server as the **root** user (assuming that `/openils/` is your install prefix):

```
cp -r /openils/var/web/xul/server/skin/media/audio /openils/var/web/.
```

Part X. Creating a New Skin: the Bare Minimum

Table of Contents

27. Introduction 149
28. Apache directives 150
29. Customizing templates 151

Chapter 27. Introduction

When you adopt the TPAC as your catalog, you must create a new skin. This involves a combination of overriding template files and setting Apache directives to control the look and feel of your customized TPAC.

Chapter 28. Apache directives

There are a few Apache directives and environment variables of note for customizing TPAC behavior. These directives should generally live within a `<vhost>` section of your Apache configuration.

- `OILSWebDefaultLocale` specifies which locale to display when a user lands on a page in the TPAC and has not chosen a different locale from the TPAC locale picker. The following example shows the `fr_ca` locale being added to the locale picker and being set as the default locale:

```
PerlAddVar OILSWebLocale "fr_ca"
PerlAddVar OILSWebLocale "/openils/var/data/locale/opac/fr-CA.po"
PerlAddVar OILSWebDefaultLocale "fr-CA"
```

- `physical_loc` is an Apache environment variable that sets the default physical location, used for setting search scopes and determining the order in which copies should be sorted. The following example demonstrates the default physical location being set to library ID 104:

```
SetEnv physical_loc 104
```

Chapter 29. Customizing templates

When you install Evergreen, the TPAC templates include many placeholder images, text, and links. You should override most of these to provide your users with a custom experience that matches your library. Following is a list of templates that include placeholder images, text, or links that you should override.

All paths are relative to `/openils/var/templates/opac`

- `parts/config.tt2`: contains many configuration settings that affect the behavior of the TPAC, including:
  - hiding the **Place Hold** button for available items
  - enabling RefWorks support for citation management
  - adding OpenURL resolution for electronic resources
  - enabling Google Analytics tracking for your TPAC
  - displaying the "Forgot your password?" prompt
  - controlling the size of cover art on the record details page
  - defining which facets to display, and in which order
  - controlling basic and advanced search options
  - controlling if the "Show More Details" button is visible or activated by default in OPAC search results
  - hiding phone notification options (useful for libraries that do not do phone notifications)
  - disallowing password or e-mail changes (useful for libraries that use centralized authentication or single sign-on systems)
  - displaying a maintenance message in the public catalog and KPAC (this is controlled by the *ctx.maintenance_message* variable)
  - displaying previews of books when available from Google Books. This is controlled by the *ctx.google_books_preview* variable, which is set to 0 by default to protect the privacy of users who might not want to share their browsing behavior with Google.
  - disabling the "Group Formats and Editions" search. This is controlled by setting the metarecords.disabled variable to 1.
  - setting the default search to a *Group Formats and Editions* search. This is done by setting the search.metarecord_default variable to 1.
- `parts/footer.tt2` and `parts/topnav_links.tt2`: contains customizable links. Defaults like *Link 1* will not mean much to your users!

- `parts/homesearch.tt2`: holds the large Evergreen logo on the home page of the TPAC. Substitute your library's logo, or if you are adventurous, create a "most recently added items" carousel… and then share your customization with the Evergreen community.

- `parts/topnav_logo.tt2`: holds the small Evergreen logo that appears on the top left of every page in the TPAC. You will also want to remove or change the target of the link that wraps the logo and leads to the Evergreen site.

- `parts/login/form.tt2`: contains some assumptions about terminology and examples that you might prefer to change to be more consistent with your own site's existing practices. For example, you may not use *PIN* at your library because you want to encourage users to use a password that is more secure than a four-digit number.

- `parts/login/help.tt2`: contains links that point to http://example.com, images with text on them (which is not an acceptable practice for accessibility reasons), and promises of answers to frequently asked questions that might not exist at your site.

- `parts/login/password_hint.tt2`: contains a hint about your users' password on first login that is misleading if your library does not set the initial password for an account to the last four digits of the phone number associated with the account.

- `parts/myopac/main_refund_policy.tt2`: describes the policy for refunds for your library.

- `parts/myopac/prefs_hints.tt2`: suggests that users should have a valid email on file so they can receive courtesy and overdue notices. If your library does not send out email notices, you should edit this to avoid misleading your users.

- `myopac/update_password_msg.tt2`: defines the password format that needs to be used when setting a user password. If your Evergreen site has set *Password format* regex in the Library Settings Editor, you should update the language to describe the format that should be used.

- `password_reset.tt2`: in the msg_map section, you might want to change the NOT_STRONG text that appears when the user tries to set a password that does not match the required format. Ideally, this message will tell the user how they should format the password.

- `parts/css/fonts.tt2`: defines the font sizes for the TPAC in terms of one base font size, and all other sizes derived from that in percentages. The default is 12 pixels, but some design sites strongly suggest a base font size of 16 pixels. Perhaps you want to try *1em* as a base to respect your users' preferences. You only need to change one number in this file if you want to experiment with different options for your users.

- `parts/css/colors.tt2`: chances are your library's official colors do not match Evergreen's wall of dark green. This file defines the colors in use in the standard Evergreen template. In theory you should be able to change just a few colors and everything will work, but in practice you will need to experiment to avoid light-gray-on-white low-contrast combinations.

The following are templates that are less frequently overridden, but some libraries benefit from the added customization options.

- `parts/advanced/numeric.tt2`: defines the search options of the Advanced Search > Numeric search. If you wanted to add a bib call number search option, which is different from the item copy call number; you would add the following code to `numeric.tt2`.

```
<option value="identifier|bibcn">[% l('Bib Call Number') %]</option>
```

Part XI. Keeping Evergreen Current and Secure

Table of Contents

30. Introduction 155
31. Upgrading the Evergreen software 156
32. Securing the server(s) on which your Evergreen installation runs 157

Chapter 30. Introduction

When it comes to running an Evergreen system, there are two special areas of concern:

- How and when you decide to upgrade Evergreen software or apply fixes
- How to take care of the actual server(s) that your Evergreen system uses

The following hints to help you cope with these challenges.

Chapter 31. Upgrading the Evergreen software

The Evergreen community at large have agreed upon an upgrade cycle that produces new major releases twice a year, in Spring and Fall. Major releases can contain new features. The community supports each major release with 12 subsequent monthly minor releases that contain only bug fixes, and continues to provide security fixes if necessary for an additional three months after the end of the regular minor bug fix support, for a total of 15 months of support for each major release.

As a general rule, as the Evergreen community releases each new version of the Evergreen software, they also provide a guideline on how to upgrade from the previous release as part of the official Evergreen documentation at http://docs.evergreen-ils.org. Follow the instructions exactly and in the order that they are given—and if you run into a problem, report it to the community with as much detail about the error message or symptoms of the problem as you can.

Keep the Evergreen release schedule in mind when planning your own testing and upgrade schedules. If you participate in testing new Evergreen releases during the release candidate stages, you will prepare your own library for the upgrade process and help flush out any remaining bugs before the major release of the software. This also gives you time to prepare the members of your library for the upcoming changes by giving them the chance, when possible, to familiarize themselves with new features on your test system. You also have the chance to prepare supporting materials, like handouts and other kinds of documentation, to help your users before, during and after each upgrade cycle.

Chapter 32. Securing the server(s) on which your Evergreen installation runs

An Evergreen installation requires interaction between many different components and, depending on the size of your consortium and how many servers you have, it can range from quite complex to extremely. That said, there are a number of standard guidelines that you can follow to secure your server.

- Keep your server up-to-date. Apply security updates as soon as possible when they come out to prevent your system from being exposed to a known vulnerability.
- Pay close attention to account administration on the server. Do not give any user on the server more power than they need.
- Disable services that you do not need.
- Pay attention to your system's log files to see what kind of activity is happening and notice anything unusual.
- A central idea to server security is to make it unreasonably difficult for anyone who tries to compromise your system. Let them choose targets more vulnerable than yours.

This topic is very rich and there are many resources available, both in print and on the web. It is worth your time to learn more.

Appendix A. Attributions

DIG Contributors

- Hilary Caws-Elwitt, Susquehanna County Library
- Karen Collier, Kent County Public Library
- George Duimovich, NRCan Library
- Lynn Floyd, Anderson County Library
- Sally Fortin, Equinox Software
- Wolf Halton, Lyrasis
- Jennifer Pringle, SITKA
- June Rayner, eiNetwork
- Steve Sheppard
- Ben Shum, Bibliomation
- Roni Shwaish, eiNetwork
- Robert Soulliere, Mohawk College

- Remington Steed, Calvin College
- Jeanette Lundgren, CW MARS
- Tim Spindler, CW MARS
- Jane Sandberg, Linn-Benton Community College
- Lindsay Stratton, Pioneer Library System
- Yamil Suarez, Berklee College of Music
- Jenny Turner, PALS

Appendix B. Admonitions

- Note

- warning

- caution

- tip

Appendix C. Licensing

Index

A

Apache, 26
audio alerts, 146
  self check interface, 146
  silencing, 146
audio_config.tt2, 146
authentication
  LDAP, 87
  proxy, 87
authority control, 68

C

configuring, 100

D

database schema, 26
Debian, 25

E

exporting
  using the command line, 68

I

importing
  using the command line, 70, 71, 72

L

LDAP, 87
Linux
  Debian, 25
  Ubuntu, 25

M

MARC editor
  configuring, 100
MARC records
  exporting
    using the command line, 68
  importing
    using the command line, 70, 71, 72
marc2are.pl, 71
marc_export, 68

N

nosound.wav, 146

P

pgingest.pl, 70
pg_loader.pl, 71
Physical characteristics wizard, 100
Populate Address by ZIP Code
  ZIP code, 89
proxy, 87

R

reporter
  starting, 75
  starting daemon, 75
  stopping daemon, 75
reports
  starting server application, 75
  stopping server application, 75

S

self check interface, 146
  audio alerts, 146
silencing, 146
starting, 75
starting daemon, 75
starting server application, 75
stopping daemon, 75
stopping server application, 75

U

Ubuntu, 25
using the command line, 68, 70, 71, 72

Z

ZIP code, 89
zips.txt
  Populate Address by ZIP Code
    ZIP code, 89