Evergreen development

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	February 2010		DS

Contents

1 Part 1: OpenSRF applications

OpenSRF, pronounced "Open Surf", is the Open **S**ervice **R**equest **F**ramework. It was designed as an architecture on which one could easily build scalable applications.

1.1 Introduction to OpenSRF

The framework is built on JSON-over-XMPP. XML can be used, but JSON is much less verbose. XMPP is a standard messaging protocol that has been used as the backbone of low-latency, high-volume applications including instant messaging and Google Wave.

OpenSRF offers scalability via its clustering architecture; a service that is a bottleneck can be moved onto its own server; or multiple instances of the service can be run on many servers. Services can themselves be clients of other services.

OpenSRF services listen at an XMPP address such as "opensrf@private.localhost/open-ils.fielder_drone_at_localhost_7652". The initial request from an OpenSRF client is directed to the OpenSRF router, which determines whether the requested service is accessible to the client (based on the public versus private domains), and then connects the client to the service for any subsequent communication that is required.

To significantly improve the speed at which request services can respond to common requests, OpenSRF has integrated support for the caching via the `memcached` daemon. For example, the contents of the configuration files are cached by the `opensrf.settings` service when that service starts, so that rather than having to parse the XML file every time a service checks a configuration setting, the value can be retrieved with much less overhead directly from the cache.

Note
if you change a setting in one of those configuration files, you must restart the `opensrf.settings` service to update its data. You must then restart any of the services that make use of that setting to make the change take effect.

Supports Perl, C, and Python as services and clients, and Java as a client. JavaScript can access services via HTTP translator and gateway. JSON library converts messages to/from native structures for ease of development.

1.2 Configuring OpenSRF

Walk through the configuration files, explaining *why* we put the values into the files that we do:

- opensrf_core.xml
  - Distinguish between public and private services for security of Web-based applications.
  - Deprecated HTTP gateway versus OpenSRF-over-HTTP
- opensrf.xml

Tip
In a clustered OpenSRF instance, these files are normally hosted on a network share so that each member of the cluster can read them.

1.3 Starting OpenSRF services

Note
I won't go through this during a live session. Perhaps I can cut this out entirely. . .

Issue the following commands as the `opensrf` user. If you are running OpenSRF on a single-server machine, you can use the `-l` flag to force the hostname to be treated as `localhost`.

1. Start the OpenSRF router:

```
osrf_ctl.sh -a start_router
```

> **Important**
> The router must only run on a single machine in a given brick.

2. Start all OpenSRF Perl services defined for this host:

```
osrf_ctl.sh -a start_perl
```

> **Tip**
> You can start an individual Perl service using:

```
opensrf-perl.pl -s <service-name> -a start -p <PID-directory>
```

3. Start all OpenSRF C services defined for this host:

```
osrf_ctl.sh -a start_c
```

1.4 Stopping OpenSRF services

Issue the following commands as the `opensrf` user. If you are running OpenSRF on a single-server machine, you can use the `-l` flag to force the hostname to be treated as `localhost`.

1. Stop the OpenSRF router:

```
osrf_ctl.sh -a stop_router
```

2. Stop all OpenSRF Perl services defined for this host:

```
osrf_ctl.sh -a stop_perl
```

> **Tip**
> You can stop an individual Perl service using:

```
opensrf-perl.pl -s <service-name> -a stop -p <PID-directory>
```

3. Stop all OpenSRF C services defined for this host:

```
osrf_ctl.sh -a stop_c
```

> **Important**
> PID files for OpenSRF services are stored and looked up in `/openils/var/run` by default with `osrf_ctl.sh`, and in `/tmp/` with `opensrf-perl.pl`. For a clustered server instance of Evergreen, you must store the PIDs on a directory that is local to each server, or else one of your cluster servers may try killing processes on itself that actually have PIDs on other servers.

2 Examining sample code

Show internal documentation for methods. Do some stupid srfsh tricks (`introspect` for one) and show `docgen.xsl` in action.

2.1 SRFSH stupid tricks

```
srfsh# introspect open-ils.auth
... returns documentation for all methods registered for open-ils.auth

srfsh# introspect open-ils.auth "open-ils.auth.authenticate"
... returns documentation for all methods with names beginning with
    "open-ils.auth.authenticate" registered for open-ils.auth

srfsh# open open-ils.cstore
... begins a stateful connection with open-ils.cstore
srfsh# request open-ils.cstore open-ils.cstore.transaction.begin
... begins a transaction
srfsh# request open-ils.cstore open-ils.cstore.direct.config.language_map.delete \
   {"code": {"like":"a%"}}
... deletes all of the entries from config.language_map that have a
... code beginning with "e"
srfsh# request open-ils.cstore open-ils.cstore.transaction.rollback
... rolls back the transaction
srfsh# close open-ils.cstore
... closes the stateful connection with open-ils.cstore
```

2.2 Perl

2.2.1 Services

See `OpenSRF/src/perl/lib/OpenSRF/UnixServer.pm` to understand how the optional methods for initializing and cleaning up OpenSRF services are invoked:

- `initialize()`
- `child_init()`
- `child_exit()`

Services are implemented as Perl functions. Each service needs to be registered with:

```
__PACKAGE__->register_method(
  method => 'method name',                    # ❶
  api_name => 'API name',                     # ❷
  api_level => 1,                             # ❸
  argc => # of args,                          # ❹
  signature => {                              # ❺
    desc => "Description",
    params => [
      {
        name => 'parameter name',
        desc => 'parameter description',
        type => '(array|hash|number|string)'
      }
    ],
    return => {
```

```
      desc => 'Description of return value',
      type => '(array|hash|number|string)'
    }
  }
);
```

1. The method name is the name of the Perl method that is called when a client invokes the corresponding OpenSRF method.
2. The API name is the OpenSRF method name. By convention, each API uses the OpenSRF service name for its root, and then appends one or more levels of names to the OpenSRF service name, depending on the complexity of the service and the number of methods exposed by a given service.
3. The API level is always `1`.
4. The number of arguments that can be passed to the OpenSRF method is primarily for guidance purposes.
5. The signature is consumed by the various utilities (srfsh, docgen.xsl) that generate documentation about the OpenSRF service.

Note that arguments are converted between native data structures and JSON for us for free.

2.2.2 Client cheat sheet

This is the simplest possible OpenSRF client written in Perl:

```
#!/usr/bin/perl
use strict;
use warnings;

use OpenSRF::System;                                          # 1

# Bootstrap the system
OpenSRF::System->bootstrap_client(config_file => $ARGV[0]);   # 2

my $session = OpenSRF::AppSession->create("open-ils.resolver"); # 3

my $holdings = $session->request(                             # 4
        "open-ils.resolver.resolve_holdings",                 # 5
        "issn", "0022-362X"                                   # 6
)->gather();                                                  # 7

$session->disconnect();                                       # 8

foreach my $holding (@$holdings) {
        foreach my $entry (keys %$holding) {
                print "$entry -> " . $holding->{$entry} . "\n";
        }
}
```

1. The `OpenSRF::System` module gives our program access to the core OpenSRF client functionality.
2. The `bootstrap_client()` method reads the `opensrf_core.xml` file and sets up communication with the OpenSRF router.
3. The `OpenSRF::Appsession->create()` instance method asks the router if it can connect to the named service. If the router determines that the service is accessible (either the opensrf credentials are on the private domain, which gives it access to all public and private services; or the service is on a public domain, which is accessible to both public and private opensrf credentials), it returns an OpenSRF session with a connection to the named service.

❹ The `OpenSRF::Appsession->request()` method invokes a method of the associated service to return a request object.

❺ The method name that you want to invoke is the first argument to `request()`.

❻ The arguments to the method follow the method name.

❼ Invoking the `gather()` method on the returned request object returns a single result.

> **Note**
> If the service is expected to return multiple results, you should loop over it with `recv()` instead. But then, that wouldn't be the simplest possible client anymore would it?

❽ The `OpenSRF::Appsession->disconnect()` instance method disconnects from the service, enabling that child to go on and handle other requests.

2.3 JavaScript

Historically, JavaScript has had access to OpenSRF methods via the OpenSRF HTTP gateway Apache module. You can still see this in heavy use in the OPAC and staff client as of Evergreen 1.6, but the approach has been deprecated as it has significant performance problems with large responses. The successor for the OpenSRF gateway is the OpenSRF-over-HTTP translator Apache module, which supports streaming responses for improved performance and better support for the broad range of OpenSRF attributes.

2.3.1 Invoking methods via the HTTP Translator

The following example demonstrates the basic approach to invoking OpenSRF methods via JavaScript. It uses just three OpenSRF JavaScript libraries to simplify calls to the OpenSRF-over-HTTP translator, which became available to developers as part of the OpenSRF 1.0 / Evergreen 1.4 releases.

```
<html>
<head>
<title>Basic call to the OpenSRF-over-HTTP translator</title>
<!-- Include required scripts for basic OpenSRF-over-HTTP -->
<script type='text/javascript' src='/opac/common/js/opensrf.js'></script>     <!-- ❶ -->
<script type='text/javascript' src='/opac/common/js/opensrf_xhr.js'></script> <!-- ❷ -->
<script type='text/javascript' src='/opac/common/js/JSON_v1.js'></script>     <!-- ❸ -->

<script type='text/javascript'>

// Create a session
var ses = new OpenSRF.ClientSession('open-ils.resolver');                      // ❹

// Send a request to the target service
var req = ses.request('open-ils.resolver.resolve_holdings.raw',                // ❺
    'issn', '0022-362X');

req.oncomplete = function(r) {                                                 // ❻
    var msg;
    while (msg = r.recv()) {                                                   // ❼
        var i;
        for (i = 0; i < msg.content().length; i++) {                           // ❽
            alert(msg.content()[i].target_coverage);                           // ❾
        }
    }
}
req.send();                                                                    // ❿
</script>
```

```
</head>
<body>
<h1>Basic call to the OpenSRF-over-HTTP translator</h1>
</body>
</html>
```

1. opensrf.js defines most of the objects and methods required for a bare JavaScript call to the OpenSRF HTTP translator.
2. opensrf_xhr.js provides cross-browser XMLHttpRequest support for OpenSRF.
3. JSON_v1.js converts the requests and responses between JavaScript and the JSON format that the OpenSRF translator expects.
4. Create a client session that connects to the `open-ils.resolver` service.
5. Create a request object that identifies the target method and passes the required method arguments.
6. Define the function that will be called when the request is sent and results are returned from the OpenSRF HTTP translator.
7. Loop over the returned results using the `recv()` method.
8. The content of each result is accessible via the content() method of each returned result.
9. `open-ils.resolver.resolve_holdings` returns a hash of values, so invoking one of the hash keys (`coverage`) gives us access to that value.
10. Actually send the request to the method; the function defined by `req.oncomplete` is invoked as the results are returned.

3 Exercise

Build a new OpenSRF service.

3.1 Perl

The challenge: implement a service that caches responses from some other Web service (potentially cutting down on client-side latency for something like OpenLibrary / Google Books / xISBN services, and avoiding timeouts if the target service is not dependable). Our example will be to build an SFX lookup service. This has the additional advantage of enabling `XmlHttpRequest` from JavaScript by hosting the services on the same domain.

Let's start with the simplest possible implementation – a CGI script.

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;
use LWP::UserAgent;
use XML::LibXML;
use JSON::XS;

my $q = CGI->new;

my $issn = $q->param("issn");
my $isbn = $q->param("isbn");

my $url_base = 'http://sfx.scholarsportal.info/laurentian';

my $url_args = '?url_ver=Z39.88-2004&url_ctx_fmt=infofi/fmt:kev:mtx:ctx' .
        '&ctx_enc=UTF-8&ctx_ver=Z39.88-2004&rfr_id=info:sid/conifer' .
        '&sfx.ignore_date_threshold=1&sfx.response_type=multi_obj_detailed_xml' .
```

```
        '&__service_type=getFullTxt';

if ($issn) {
        $url_args .= "&rft.issn=$issn";
} elsif ($isbn) {
        $url_args .= "&rft.isbn=$isbn";
}

my $ua = LWP::UserAgent->new;
$ua->agent("SameOrigin/1.0");

my $req = HTTP::Request->new(GET => "$url_base$url_args");
my $res = $ua->request($req);

print $q->header('text/json');
my $xml = $res->content;
my $parser = XML::LibXML->new();
my $parsed_sfx = $parser->parse_string($xml);

my (@targets) = $parsed_sfx->findnodes('//target');

my @sfx_result;
foreach my $target (@targets) {
        my $public_name = $target->findvalue('./target_public_name');
        my $target_url = $target->findvalue('.//target_url');
        my $target_coverage = $target->findvalue('.//coverage_statement');
        my $target_embargo = $target->findvalue('.//embargo_statement');
        push @sfx_result, {
                public_name => $public_name,
                coverage => $target_coverage,
                embargo => $target_embargo,
                url => $target_url
        };
}

print encode_json(\@sfx_result);
```

Hopefully you can follow what this CGI script is doing. It works, but it has all the disadvantages of CGI: the environment needs to be built up on every request, and it doesn't remember anything from the previous times it was called, etc.

3.1.1 Turning the CGI script into an OpenSRF service

So now we want to turn this into an OpenSRF service.

1. Start by ripping out the CGI stuff, as we won't need that any more.
2. To turn this into an OpenSRF service, we create a new Perl module (`OpenILS::Application::ResolverResolver`). We no longer have to convert results between Perl and JSON values, as OpenSRF will handle that for us. We now have to register the method with OpenSRF.

```
#!/usr/bin/perl

# Copyright (C) 2009 Dan Scott <dscott@laurentian.ca>

# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
```

```
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.

=head1 NAME

OpenILS::Application::ResolverResolver - retrieves holdings from OpenURL resolvers

=head1 SYNOPSIS

Via srfsh:
  request open-ils.resolver open-ils.resolver.resolve_holdings "issn", "0022-362X"

Via Perl:
  my $session = OpenSRF::AppSession->create("open-ils.resolver");
  my $request = $session->request("open-ils.resolver.resolve_holdings", [ "issn", " ↩
      0022-362X" ] )->gather();
  $session->disconnect();

  # $request is a reference to the list of hashes

=head1 DESCRIPTION

OpenILS::Application::ResolverResolver caches responses from OpenURL resolvers
to requests for full-text holdings. Currently integration with SFX is supported.

Each org_unit can specify a different base URL as the third argument to
resolve_holdings(). Eventually org_units will have org_unit settings to hold
their resolver type and base URL.

=head1 AUTHOR

Dan Scott, dscott@laurentian.ca

=cut

package OpenILS::Application::ResolverResolver;

use strict;
use warnings;
use LWP::UserAgent;
use XML::LibXML;

# All OpenSRF applications must be based on OpenSRF::Application or
# a subclass thereof.  Makes sense, eh?
use OpenILS::Application;
use base qw/OpenILS::Application/;

# This is the client class, used for connecting to open-ils.storage
use OpenSRF::AppSession;

# ... and here we have the built in logging helper ...
use OpenSRF::Utils::Logger qw($logger);

our ($ua, $parser);

sub child_init {
    # We need a User Agent to speak to the SFX beast
```

```
    $ua = new LWP::UserAgent;
    $ua->agent('SameOrigin/1.0');

    # SFX returns XML to us; let us parse
    $parser = new XML::LibXML;
}

sub resolve_holdings {
    my $self = shift;
    my $conn = shift;
    my $id_type = shift; # keep it simple for now, either 'issn' or 'isbn'
    my $id_value = shift; # the normalized ISSN or ISBN

    # For now we'll pass the argument with a hard-coded default
    # Should pull these specifics from the database as part of initialize()
    my $url_base = shift || 'http://sfx.scholarsportal.info/laurentian';

    # Big ugly SFX OpenURL request
    my $url_args = '?url_ver=Z39.88-2004&url_ctx_fmt=infofi/fmt:kev:mtx:ctx&'
        . 'ctx_enc=UTF-8&ctx_ver=Z39.88-2004&rfr_id=info:sid/conifer&'
        . 'sfx.ignore_date_threshold=1&'
        . 'sfx.response_type=multi_obj_detailed_xml&__service_type=getFullTxt';

    if ($id_type == 'issn') {
        $url_args .= "&rft.issn=$id_value";
    } elsif ($id_type == 'isbn') {
        $url_args .= "&rft.isbn=$id_value";
    }

    # Otherwise, let's go and grab the info from the SFX server
    my $req = HTTP::Request->new('GET', "$url_base$url_args");

    # Let's see what we we're trying to request
    $logger->info("Resolving the following request: $url_base$url_args");

    my $res = $ua->request($req);

    my $xml = $res->content;
    my $parsed_sfx = $parser->parse_string($xml);

    my (@targets) = $parsed_sfx->findnodes('//target');

    my @sfx_result;
    foreach my $target (@targets) {
        my $public_name = $target->findvalue('./target_public_name');
        my $target_url = $target->findvalue('.//target_url');
        my $target_coverage = $target->findvalue('.//coverage_statement');
        my $target_embargo = $target->findvalue('.//embargo_statement');
        push @sfx_result, {
            public_name => $public_name, coverage => $target_coverage,
            embargo => $target_embargo, url => $target_url
        };
    }

    return \@sfx_result;
}

__PACKAGE__->register_method(
    method    => 'resolve_holdings',
    api_name  => 'open-ils.resolver.resolve_holdings',
    api_level => 1,
    argc      => 3,
```

```
    signature => {
        desc     => <<"          DESC",
Returns a list of the full-text holdings for a given ISBN or ISSN
        DESC
        'params' => [ {
                name => 'id_type',
                desc => 'The type of identifier ("issn" or "isbn")',
                type => 'string'
            }, {
                name => 'id_value',
                desc => 'The identifier value',
                type => 'string'
            }, {
                 name => 'url_base',
                 desc => 'The base URL for the resolver and instance',
                 type => 'string'
            },
        ],
        'return' => {
            desc => 'Returns a list of the full-text holdings for a given ISBN or ISSN' ↩
                ,
            type => 'array'
        }
    }
);


1;
```

3. Copy the file into the `/openils/lib/perl5/OpenILS/Application/` directory so that OpenSRF can find it in the `@INC` search path.
4. Add the service to `opensrf.xml` so it gets started with the other Perl services on our host of choice:

```
...
<open-ils.resolver>
  <keepalive>3</keepalive>
  <stateless>1</stateless>
  <language>perl</language>
  <implementation>OpenILS::Application::ResolverResolver</implementation>
  <max_requests>17</max_requests>
  <unix_config>
    <unix_sock>open-ils.resolver_unix.sock</unix_sock>
    <unix_pid>open-ils.resolver_unix.pid</unix_pid>
    <max_requests>1000</max_requests>
    <unix_log>open-ils.resolver_unix.log</unix_log>
    <min_children>5</min_children>
    <max_children>15</max_children>
    <min_spare_children>3</min_spare_children>
    <max_spare_children>5</max_spare_children>
  </unix_config>
  <app_settings>
    <cache_timeout>86400</cache_timeout>
    <default_url_base>http://sfx.scholarsportal.info/laurentian</default_url_base>
  </app_settings>
</open-ils.resolver>
...
<!-- In the <hosts> section -->
<localhost>
  ...
  <appname>open-ils.resolver</appname>
</localhost>
```

5. Add the service to `opensrf_core.xml` as a publicly exposed service via the HTTP gateway and translator:

```
...
<!-- In the public router section -->
<services>
  ...
  <service>open-ils.resolver</service>
</services>
...
<!-- In the public gateway section -->
<services>
<gateway>
  ...
  <services>
    <service>open-ils.resolver</service>
  </services>
</gateway>
```

6. Restart the OpenSRF Perl services to refresh the OpenSRF settings and start the service..
7. Restart Apache to enable the gateway and translator to pick up the new service.

3.1.2 Add caching

To really make this service useful, we can take advantage of OpenSRF's built-in support for caching via memcached. Keeping the values returned by the resolver for 1 week is apparently good.

We will also take advantage of the `opensrf.settings` service that holds the values defined in the `opensrf.xml` configuration file to supply some of our default arguments.

```
--- ResolverResolver.pm.basic   2009-10-22 16:52:55.000000000 -0400
+++ ResolverResolver.pm 2009-10-22 16:56:42.000000000 -0400
@@ -62,11 +62,32 @@
 # This is the client class, used for connecting to open-ils.storage
 use OpenSRF::AppSession;

+# This is an extension of Error.pm that supplies some error types to throw
+use OpenSRF::EX qw(:try);
+
+# This is a helper class for querying the OpenSRF Settings application ...
+use OpenSRF::Utils::SettingsClient;
+
 # ... and here we have the built in logging helper ...
 use OpenSRF::Utils::Logger qw($logger);

+# ... and this manages cached results for us ...
+use OpenSRF::Utils::Cache;
+
+my $prefix = "open-ils.resolver_"; # Prefix for caching values
+my $cache;
+my $cache_timeout;
+
 our ($ua, $parser);

+
+sub initialize {
+       $cache = OpenSRF::Utils::Cache->new('global');
+       my $sclient = OpenSRF::Utils::SettingsClient->new();
+       $cache_timeout = $sclient->config_value(
+                        "apps", "open-ils.resolver", "app_settings", "cache_timeout" ) ||  ↩
    300;
```

```
+}
+
 sub child_init {
        # We need a User Agent to speak to the SFX beast
        $ua = new LWP::UserAgent;
@@ -82,6 +103,9 @@
        my $id_type = shift; # keep it simple for now, either 'issn' or 'isbn'
        my $id_value = shift; # the normalized ISSN or ISBN

+       # We'll use this in our cache key
+       my $method = "open-ils.resolver.resolve_holdings";
+
        # For now we'll pass the argument with a hard-coded default
         # Should pull these specifics from the database as part of initialize()
        my $url_base = shift || 'http://sfx.scholarsportal.info/laurentian';
@@ -98,6 +122,16 @@
                $url_args .= "&rft.isbn=$id_value";
        }

+       my $ckey = $prefix . $method . $url_base . $id_type . $id_value;
+
+       # Check the cache to see if we've already looked this up
+       # If we have, shortcut our return value
+       my $result = $cache->get_cache($ckey) || undef;
+       if ($result) {
+               $logger->info("Resolver found a cache hit");
+               return $result;
+       }
+
        # Otherwise, let's go and grab the info from the SFX server
        my $req = HTTP::Request->new('GET', "$url_base$url_args");

@@ -120,6 +154,9 @@
                push @sfx_result, {public_name => $public_name, coverage =>  ↩
                    $target_coverage, embargo => $target_embargo, url => $target_url};
        }

+       # Stuff this into the cache
+       $cache->put_cache($ckey, \@sfx_result, $cache_timeout);
+
        return \@sfx_result;
 }

@@ -150,4 +187,6 @@
                }
 );

+# Add methods to clear cache for specific lookups?
+
 1;
```

3.1.3 Pulling application settings from `opensrf.xml`

In case you missed it in the previous diff, we also started pulling some application-specific settings from `opensrf.xml` during the `initialize()` phase for the service.

In the following diff, we enable the service to pull the default URL from `opensrf.xml` rather than hard-coding it into the OpenSRF service… because that's just the right thing to do.

```
=== modified file 'ResolverResolver.pm'
--- ResolverResolver.pm    2009-10-22 21:00:15 +0000
```

```
+++ ResolverResolver.pm    2009-10-24 03:00:30 +0000
@@ -77,6 +77,7 @@
 my $prefix = "open-ils.resolver_"; # Prefix for caching values
 my $cache;
 my $cache_timeout;
+my $default_url_base;              # Default resolver location

 our ($ua, $parser);

@@ -86,6 +87,8 @@
     my $sclient = OpenSRF::Utils::SettingsClient->new();
     $cache_timeout = $sclient->config_value(
         "apps", "open-ils.resolver", "app_settings", "cache_timeout" ) || 300;
+    $default_url_base = $sclient->config_value(
+        "apps", "open-ils.resolver", "app_settings", "default_url_base");
 }

 sub child_init {
@@ -102,14 +105,11 @@
     my $conn = shift;
     my $id_type = shift; # keep it simple for now, either 'issn' or 'isbn'
     my $id_value = shift; # the normalized ISSN or ISBN
+    my $url_base = shift || $default_url_base;

     # We'll use this in our cache key
     my $method = "open-ils.resolver.resolve_holdings";

-    # For now we'll pass the argument with a hard-coded default
-    # Should pull these specifics from the database as part of initialize()
-    my $url_base = shift || 'http://sfx.scholarsportal.info/laurentian';
-
     # Big ugly SFX OpenURL request
     my $url_args = '?url_ver=Z39.88-2004&url_ctx_fmt=infofi/fmt:kev:mtx:ctx&'
         . 'ctx_enc=UTF-8&ctx_ver=Z39.88-2004&rfr_id=info:sid/conifer&'
```

The `opensrf.settings` service caches the settings defined in `opensrf.xml`, so if you change a setting in the configuration files and want that change to take effect immediately, you have to:

1. Restart the `opensrf.settings` service to refresh the cached settings.
2. Restart the affected service to make the new settings take effect.

Next step: add org_unit settings for resolver type and URL on a per-org_unit basis. OrgUnit settings can be retrieved via `OpenILS::Application::AppUtils->ou_ancestor_setting_value($org_id, $setting_name)`.

This is where we step beyond OpenSRF and start getting into the Evergreen database schema (`config.org_unit_setting` table).

3.2 Further reading

OpenSRF terminology: http://open-ils.org/dokuwiki/doku.php?id=osrf-devel:terms

4 Part 2: Evergreen applications

4.1 Authentication

Although many services offer methods that can be invoked without authentication, some methods require authentication in Evergreen. Evergreen's authentication framework returns an *authentication token* when a user has successfully logged in to

represent that user session. You can then pass the authentication token to various methods to ensure, for example, that the requesting user has permission to access the circulation information attached to a particular account, or has been granted the necessary permissions at a particular library to perform the action that they are requesting.

Authentication in Evergreen is performed with the assistance of the `open-ils.auth` service, which has been written in C for performance reasons because it is invoked so frequently. A successful authentication request requires two steps:

1. Retrieve an authentication seed value by invoking the `open-ils.auth.authenticate.init` method, passing the user name as the only argument. As long as the user name contains no spaces, the method returns a seed value calculated by the MD5 checksum of a string composed of the concatenation of the time() system call, process ID, and user name.
2. Retrieve an authentication token by invoking the `open-ils.auth.authenticate.complete` method, passing a JSON hash composed of a minimum of the following arguments (where *seed* represents the value returned by the `open-ils.auth.authenticate.init` method):

```
{
    "username": username, // or "barcode": barcode,
    "password": md5sum(seed + md5sum(password)),
}
```

`open-ils.auth.authenticate.complete` also accepts the following additional arguments:

- `type`: one of "staff" (default), "opac", or "temp"
- `org`: the numeric ID of the org_unit where the login is active
- `workstation`: the registered workstation name

4.1.1 Authentication in Perl

The following example is taken directly from `OpenILS::WWW::Proxy`:

```
sub oils_login {
    my( $username, $password, $type ) = @_;

    $type |= "staff";
    my $nametype = 'username';
    $nametype = 'barcode' if ($username =~ /^\d+$/o);

    my $seed = OpenSRF::AppSession
        ->create("open-ils.auth")
        ->request( 'open-ils.auth.authenticate.init', $username )
        ->gather(1);

    return undef unless $seed;

    my $response = OpenSRF::AppSession
        ->create("open-ils.auth")
        ->request( 'open-ils.auth.authenticate.complete', {
        $nametype => $username,
            password => md5_hex($seed . md5_hex($password)),
            type => $type
        })
        ->gather(1);

    return undef unless $response;

    return $response->{payload}->{authtoken};
}
```

4.1.2 Authentication in JavaScript

The following example provides a minimal implementation of the authentication method in JavaScript. For a more complete implementation, you would differentiate between user names and barcodes, potentially accept the org_unit and workstation name for more granular permissions, and provide exception handling.

```
<html>
<title>Basic authentication via JavaScript</title>
<!-- Include required scripts for basic OpenSRF-over-HTTP -->
<script type='text/javascript' src='/opac/common/js/opensrf.js'></script>     <!-- ❶ -->
<script type='text/javascript' src='/opac/common/js/opensrf_xhr.js'></script> <!-- ❷ -->
<script type='text/javascript' src='/opac/common/js/JSON_v1.js'></script>     <!-- ❸ -->
<script type='text/javascript' src='/opac/common/js/md5.js'></script>         <!-- ❹ -->

<script type='text/javascript'>

function login_user(uname, passwd) {
    var seed;
    var authtoken;

    // Create a session
    var ses = new OpenSRF.ClientSession('open-ils.auth');                     // ❺

    // Get an authentication seed
    var req = ses.request('open-ils.auth.authenticate.init', uname);          // ❻
    req.timeout = 5;                                                          // ❼
    req.oncomplete = function(r) {
        seed = r.recv().content();
    }
    req.send();                                                               // ❽

    // Set up arguments for completing the login
    var auth_args = {                                                         // ❾
        "password": hex_md5(seed + hex_md5(passwd)),
        "type": "opac"
    };

    // For simplicity, we assume the login type is user name, not barcode
    auth_args.username = uname;                                               // ❿

    req = ses.request('open-ils.auth.authenticate.complete', auth_args);      // ⓫
    req.timeout = 5;
    req.oncomplete = function(r) {
        authtoken = r.recv().content().payload.authtoken;                     // ⓬
    }
    req.send(true);
    alert(authtoken);
}

login_user('admin', 'open-ils');
</script>

</head>
<body>
<h1>Basic call to the OpenSRF-over-HTTP translator</h1>
</body>
</html>
```

❶ opensrf.js defines most of the objects and methods required for a bare JavaScript call to the OpenSRF HTTP translator.

❷ opensrf_xhr.js provides cross-browser XMLHttpRequest support for OpenSRF.

3. JSON_v1.js converts the requests and responses between JavaScript and the JSON format that the OpenSRF translator expects.
4. md5.js provides the implementation of the md5sum algorithm in the `hex_md5` function
5. Create a client session that connects to the `open-ils.auth` service.
6. Create a request object that invokes the `open-ils.auth.authenticate.init` method, providing the user name as the salt.
7. Set the `timeout` property on the request object to make it a synchronous call.
8. Send the request. The method returns a seed value which is assigned to the `seed` variable.
9. Create the hash of parameters that will be sent in the request to the `open-ils.auth.authenticate.complete` method, including the password and authentication type.
10. Assume that the credentials being sent are based on the user name rather than the barcode. The Perl implementation tests the value of the user name variable to determine whether it contains a digit; if it does contain a digit, then it is considered a barcode rather than a user name. Ensure that your implementations are consistent!
11. Create a request object that invokes the `open-ils.auth.authenticate.complete` method, passing the entire hash of parameters. Once again, set the `timeout` parameter to make the request synchronous.
12. Assign the `authtoken` attribute of the returned payload to the `authtoken` variable.

5 Evergreen data models and access

5.1 Database schema

The database schema is tied pretty tightly to PostgreSQL. Although PostgreSQL adheres closely to ANSI SQL standards, the use of schemas, SQL functions implemented in both plpgsql and plperl, and PostgreSQL's native full-text search would make it... challenging... to port to other database platforms.

A few common PostgreSQL interfaces for poking around the schema and manipulating data are:

- psql (the command line client)
- pgadminIII (a GUI client).

Or you can read through the source files in Open-ILS/src/sql/Pg.

Let's take a quick tour through the schemas, pointing out some highlights and some key interdependencies:

- actor.org_unit → asset.copy_location
- actor.usr → actor.card
- biblio.record_entry → asset.call_number → asset.copy
- config.metabib_field → metabib.*_field_entry

5.2 Database access methods

You could use direct access to the database via Perl DBI, JDBC, etc, but Evergreen offers several database CRUD services for creating / retrieving / updating / deleting data. These avoid tying you too tightly to the current database schema and they funnel database access through the same mechanism, rather than tying up connections with other interfaces.

5.3 Evergreen Interface Definition Language (IDL)

Defines properties and required permissions for Evergreen classes. To reduce network overhead, a given object is identified via a class-hint and serialized as a JSON array of properties (no named properties).

As of 1.6, fields will be serialized in the order in which they appear in the IDL definition file, and the is_new / is_changed / is_deleted properties are automatically added. This has greatly reduced the size of the `fm_IDL.xml` file and makes DRY people happier :)

- … oils_persist:readonly tells us, if true, that the data lives in the database, but is pulled from the SELECT statement defined in the <oils_persist:source_definition> child element

5.3.1 IDL basic example (config.language_map)

```
<class id="clm" controller="open-ils.cstore open-ils.pcrud"                 # ❶
        oils_obj:fieldmapper="config::language_map"                         # ❷
        oils_persist:tablename="config.language_map"                        # ❸
        reporter:label="Language Map" oils_persist:field_safe="true">       # ❹
    <fields oils_persist:primary="code" oils_persist:sequence="">           # ❺
        <field reporter:label="Language Code" name="code"                   # ❻
            reporter:selector="value" reporter:datatype="text"/>
        <field reporter:label="Language" name="value"
            reporter:datatype="text" oils_persist:i18n="true"/>             # ❼
    </fields>
    <links/>
    <permacrud xmlns="http://open-ils.org/spec/opensrf/IDL/permacrud/v1">   # ❽
        <actions>
            <create global_required="true" permission="CREATE_MARC_CODE">   # ❾
            <retrieve global_required="true"
                permission="CREATE_MARC_CODE UPDATE_MARC_CODE DELETE_MARC_CODE">
            <update global_required="true" permission="UPDATE_MARC_CODE">
            <delete global_required="true" permission="DELETE_MARC_CODE">
        </actions>
    </permacrud>
</class>
```

❶ The `class` element defines the attributes and permissions for classes, and relationships between classes.

- The `id` attribute on the `class` element defines the class hint that is used everywhere in Evergreen.
- The `controller` attribute defines the OpenSRF services that provide access to the data for the class objects.

❷ The `oils_obj::fieldmapper` attribute defines the name of the class that is generated by `OpenILS::Utils::Fieldmap`

❸ The `oils_persist:tablename` attribute defines the name of the table that contains the data for the class objects.

❹ The reporter interface uses `reporter:label` attribute values in the source list to provide meaningful class and attribute names. The `open-ils.fielder` service generates a set of methods that provide direct access to the classes for which `oils_persist:field_safe` is `true`. For example,

```
srfsh# request open-ils.fielder open-ils.fielder.clm.atomic \
    {"query":{"code":{"=":"eng"}}}

Received Data: [
  {
    "value":"English",
    "code":"eng"
  }
]
```

5. The `fields` element defines the list of fields for the class.
   - The `oils_persist:primary` attribute defines the column that acts as the primary key for the table.
   - The `oils_persist:sequence` attribute holds the name of the database sequence.
6. Each `field` element defines one property of the class.
   - The `name` attribute defines the getter/setter method name for the field.
   - The `reporter:label` attribute defines the attribute name as used in the reporter interface.
   - The `reporter:selector` attribute defines the field used in the reporter filter interface to provide a selectable list. This gives the user a more meaningful access point than the raw numeric ID or abstract code.
   - The `reporter:datatype` attribute defines the type of data held by this property for the purposes of the reporter.
7. The `oils_persist:i18n` attribute, when `true`, means that translated values for the field's contents may be accessible in different locales.
8. The `permacrud` element defines the permissions (if any) required to **c**reate, **r**etrieve, **u**pdate, and **d**elete data for this class. `open-ils.permacrud` must be defined as a controller for the class for the permissions to be applied.
9. Each action requires one or more `permission` values that the user must possess to perform the action.
   - If the `global_required` attribute is `true`, then the user must have been granted that permission globally (depth = 0) to perform the action.
   - The `context_field` attribute denotes the `<field>` that identifies the org_unit at which the user must have the pertinent permission.
   - An action element may contain a `<context_field>` element that defines the linked class (identified by the `link` attribute) and the field in the linked class that identifies the org_unit where the permission must be held.
     - If the `<context_field>` element contains a `jump` attribute, then it defines a link to a link to a class with a field identifying the org_unit where the permission must be held.

5.3.2 Reporter data types and their possible values

- `bool`: Boolean `true` or `false`
- `id`: ID of the row in the database
- `int`: integer value
- `interval`: PostgreSQL time interval
- `link`: link to another class, as defined in the `<links>` element of the class definition
- `money`: currency amount
- `org_unit`: list of org_units
- `text`: text value
- `timestamp`: PostgreSQL timestamp

5.3.3 IDL example with linked fields (actor.workstation)

Just as tables often include columns with foreign keys that point to values stored in the column of a different table, IDL classes can contain fields that link to fields in other classes. The `<links>` element defines which fields link to fields in other classes, and the nature of the relationship:

```
<class id="aws" controller="open-ils.cstore"
        oils_obj:fieldmapper="actor::workstation"
        oils_persist:tablename="actor.workstation"
        reporter:label="Workstation">
    <fields oils_persist:primary="id"
            oils_persist:sequence="actor.workstation_id_seq">
        <field reporter:label="Workstation ID" name="id"
                reporter:datatype="id"/>
        <field reporter:label="Workstation Name" name="name"
                reporter:datatype="text"/>
        <field reporter:label="Owning Library" name="owning_lib"
                reporter:datatype="org_unit"/>
        <field reporter:label="Circulations" name="circulations"
                oils_persist:virtual="true" reporter:datatype="link"/>      # ❶
    </fields>
    <links>                                                                 # ❷
        <link field="owning_lib" reltype="has_a" key="id"                   # ❸
                map="" class="aou"/>
        <link field="circulations" reltype="has_many" key="workstation"
                map="" class="circ"/>
        <link field="circulation_checkins" reltype="has_many"
                key="checkin_workstation" map="" class="circ"/>
    </links>
</class>
```

❶ This field includes an `oils_persist:virtual` attribute with the value of `true`, meaning that the linked class `circ` is a virtual class.

❷ The `<links>` element contains 0 or more `<link>` elements.

❸ Each `<link>` element defines the field (`field`) that links to a different class (`class`), the relationship (`rel_type`) between this field and the target field (`key`). If the field in this class links to a virtual class, the (`map`) attribute defines the field in the target class that returns a list of matching objects for each object in this class.

5.4 `open-ils.cstore` data access interfaces

For each class documented in the IDL, the `open-ils.cstore` service automatically generates a set of data access methods, based on the `oils_persist:tablename` class attribute.

For example, for the class hint `clm`, cstore generates the following methods with the `config.language_map` qualifer:

- `open-ils.cstore.direct.config.language_map.id_list {"code" { "like":  "e%" } }`
  Retrieves a list composed only of the IDs that match the query.
- `open-ils.cstore.direct.config.language_map.retrieve "eng"`
  Retrieves the object that matches a specific ID.
- `open-ils.cstore.direct.config.language_map.search {"code" :  "eng"}`
  Retrieves a list of objects that match the query.
- `open-ils.cstore.direct.config.language_map.create <_object_>`
  Creates a new object from the passed in object.

- `open-ils.cstore.direct.config.language_map.update <_object_>`
  Updates the object that has been passed in.
- `open-ils.cstore.direct.config.language_map.delete "eng"`
  Deletes the object that matches the query.

5.5 open-ils.pcrud data access interfaces

For each class documented in the IDL, the `open-ils.pcrud` service automatically generates a set of data access methods, based on the `oils_persist:tablename` class attribute.

For example, for the class hint `clm`, `open-ils.pcrud` generates the following methods that parallel the `open-ils.cstore` interface:

- `open-ils.pcrud.id_list.clm <_authtoken_>, { "code": { "like": "e%" } }`
- `open-ils.pcrud.retrieve.clm <_authtoken_>, "eng"`
- `open-ils.pcrud.search.clm <_authtoken_>, { "code": "eng" }`
- `open-ils.pcrud.create.clm <_authtoken_>, <_object_>`
- `open-ils.pcrud.update.clm <_authtoken_>, <_object_>`
- `open-ils.pcrud.delete.clm <_authtoken_>, "eng"`

5.6 Transaction and savepoint control

Both `open-ils.cstore` and `open-ils.pcrud` enable you to control database transactions to ensure that a set of operations either all succeed, or all fail, atomically:

- `open-ils.cstore.transaction.begin`
- `open-ils.cstore.transaction.commit`
- `open-ils.cstore.transaction.rollback`
- `open-ils.pcrud.transaction.begin`
- `open-ils.pcrud.transaction.commit`
- `open-ils.pcrud.transaction.rollback`

At a more granular level, `open-ils.cstore` and `open-ils.pcrud` enable you to set database savepoints to ensure that a set of operations either all succeed, or all fail, atomically, within a given transaction:

- `open-ils.cstore.savepoint.begin`
- `open-ils.cstore.savepoint.commit`
- `open-ils.cstore.savepoint.rollback`
- `open-ils.pcrud.savepoint.begin`
- `open-ils.pcrud.savepoint.commit`
- `open-ils.pcrud.savepoint.rollback`

Transactions and savepoints must be performed within a stateful connection to the `open-ils.cstore` and `open-ils.pcrud` services. In `srfsh`, you can open a stateful connection using the `open` command, and then close the stateful connection using the `close` command - for example:

```
srfsh# open open-ils.cstore
... perform various transaction-related work
srfsh# close open-ils.cstore
```

5.6.1 JSON Queries

Beyond simply retrieving objects by their ID using the `\*.retrieve` methods, you can issue queries against the `\*.delete` and `\*.search` methods using JSON to filter results with simple or complex search conditions.

For example, to generate a list of barcodes that are held in a copy location that allows holds and is visible in the OPAC:

```
srfsh# request open-ils.cstore open-ils.cstore.json_query       #\ ❶
    {"select": {"acp":["barcode"], "acpl":["name"]},            #\ ❷
     "from":   {"acp":"acpl"},                                  #\ ❸
     "where":  [                                                #\ ❹
         {"+acpl": "holdable"},                                 #\ ❺
         {"+acpl": "opac_visible"}                              #\ ❻
     ]}

Received Data: {
  "barcode":"BARCODE1",
  "name":"Stacks"
}

Received Data: {
  "barcode":"BARCODE2",
  "name":"Stacks"
}
```

❶ Invoke the `json_query` service.

❷ Select the `barcode` field from the `acp` class and the `name` field from the `acpl` class.

❸ Join the `acp` class to the `acpl` class based on the linked field defined in the IDL.

❹ Add a `where` clause to filter the results. We have more than one condition beginning with the same key, so we wrap the conditions inside an array.

❺ The first condition tests whether the boolean value of the `holdable` field on the `acpl` class is true.

❻ The second condition tests whether the boolean value of the `opac_visible` field on the `acpl` class is true.

For thorough coverage of the breadth of support offered by JSON query syntax, see JSON Queries: A Tutorial.

5.6.2 Fleshing linked objects

A simplistic approach to retrieving a set of objects that are linked to an object that you are retrieving - for example, a set of call numbers linked to the barcodes that a given user has borrowed - would be to: 1. Retrieve the list of circulation objects (`circ` class) for a given user (`usr` class). 2. For each circulation object, look up the target copy (`target_copy` field, linked to the `acp` class). 3. For each copy, look up the call number for that copy (`call_number` field, linked to the `acn` class).

However, this would result in potentially hundreds of round-trip queries from the client to the server. Even with low-latency connections, the network overhead would be considerable. So, built into the `open-ils.cstore` and `open-ils.pcrud` access methods is the ability to *flesh* linked fields - that is, rather than return an identifier to a given linked field, the method can return the entire object as part of the initial response.

Most of the interfaces that return class instances from the IDL offer the ability to flesh returned fields. For example, the `open-ils.cstore.direct.\*.retrieve` methods allow you to specify a JSON structure defining the fields you wish to flesh in the returned object.

```
srfsh# request open-ils.cstore open-ils.cstore.direct.asset.copy.retrieve 1, \
    {
        "flesh": 1,                                             #\ ❶
        "flesh_fields": {                                       #\ ❷
            "acp": ["location"]
        }
    }
```

1. The `flesh` argument is the depth at which objects should be fleshed. For example, to flesh out a field that links to another object that includes a field that links to another object, you would specify a depth of 2.

2. The `flesh_fields` argument contains a list of objects with the fields to flesh for each object.

Let's flesh things a little deeper. In addition to the copy location, let's also flesh the call number attached to the copy, and then flesh the bibliographic record attached to the call number.

```
request open-ils.cstore open-ils.cstore.direct.asset.copy.retrieve 1, \
    {
        "flesh": 2,
        "flesh_fields": {
            "acp": ["location", "call_number"],
            "acn": ["record"]
         }
    }
```

5.7 Adding an IDL entry for ResolverResolver

Most OpenSRF methods in Evergreen define their object interface in the IDL. Without an entry in the IDL, the prospective caller of a given method is forced to either call the method and inspect the returned contents, or read the source to work out the structure of the JSON payload. At this stage of the tutorial, we have not defined an entry in the IDL to represent the object returned by the `open-ils.resolver.resolve_holdings` method. It is time to complete that task.

The `open-ils.resolver` service is unlike many of the other classes defined in the IDL because its data is not stored in the Evergreen database. Instead, the data is requested from an external Web service and only temporarily cached in `memcached`. Fortunately, the IDL enables us to represent this kind of class by setting the `oils_persist:virtual` class attribute to `true`.

So, let's add an entry to the IDL for the `open-ils.resolver.resolve_holdings` service:

```
<class id="rhr" oils_obj:fieldmapper="resolver::holdings_record" oils_persist:virtual="true ↩
    ">
    <fields>
        <field name="public_name" oils_persist:virtual="true" />
        <field name="target_url" oils_persist:virtual="true" />
        <field name="target_coverage" oils_persist:virtual="true" />
        <field name="target_embargo" oils_persist:virtual="true" />
    </fields>
</class>
```

And let's make `ResolverResolver.pm` return an array composed of our new `rhr` classes rather than raw JSON objects:

```
=== modified file 'ResolverResolver.pm'
--- ResolverResolver.pm 2009-10-24 03:07:04 +0000
+++ ResolverResolver.pm 2009-10-27 14:31:28 +0000
@@ -74,6 +74,9 @@
 # ... and this manages cached results for us ...
 use OpenSRF::Utils::Cache;

+# ... and this gives us access to the Fieldmapper
+use OpenILS::Utils::Fieldmapper;
+
 my $prefix = "open-ils.resolver_"; # Prefix for caching values
 my $cache;
 my $cache_timeout;
@@ -147,14 +150,12 @@

     my @sfx_result;
     foreach my $target (@targets) {
```

```
-         my $public_name = $target->findvalue('./target_public_name');
-         my $target_url = $target->findvalue('.//target_url');
-         my $target_coverage = $target->findvalue('.//coverage_statement');
-         my $target_embargo = $target->findvalue('.//embargo_statement');
-         push @sfx_result, {
-             public_name => $public_name, coverage => $target_coverage,
-             embargo => $target_embargo, url => $target_url
-         };
+        my $rhr = Fieldmapper::resolver::holdings_record->new;
+         $rhr->public_name($target->findvalue('./target_public_name'));
+         $rhr->target_url($target->findvalue('.//target_url'));
+         $rhr->target_coverage($target->findvalue('.//coverage_statement'));
+         $rhr->target_embargo($target->findvalue('.//embargo_statement'));
+         push @sfx_result, $rhr;
     }

     # Stuff this into the cache
```

Once we add the new entry to the IDL and copy the revised `ResolverResolver.pm` Perl module to `/openils/lib/perl5/Ope` we need to:

1. Copy the updated IDL to both the `/openils/conf/` and `/openils/var/web/reports/` directories. The Dojo approach to parsing the IDL uses the IDL stored in the reports directory.
2. Restart the Perl services to make the new IDL visible to the services and refresh the `open-ils.resolver` implementation
3. Rerun `/openils/bin/autogen.sh` to regenerate the JavaScript versions of the IDL required by the HTTP translator and gateway.

We also need to adjust our JavaScript client to use the nifty new objects that `open-ils.resolver.resolve_holdings` now returns. The best approach is to use the support in Evergreen's Dojo extensions to generate the JavaScript classes directly from the IDL XML file.

```
<html>
<head>
<title>Basic call to the OpenSRF-over-HTTP translator</title>

<script type='text/javascript' src='/js/dojo/dojo/dojo.js'
    djConfig='parseOnLoad: true, isDebug:false'></script>        <!-- ❶ -->

<script type='text/javascript'>
dojo.require('fieldmapper.AutoIDL');                              // ❷
dojo.require('fieldmapper.dojoData');                             // ❸
dojo.require('fieldmapper.Fieldmapper');                          // ❹

var holdings = fieldmapper.standardRequest(                       // ❺
    ['open-ils.resolver', 'open-ils.resolver.resolve_holdings'],  // ❻
    ['issn', '0022-362X']);                                       // ❼

for (i = 0; i < holdings.length; i++) {
    alert(holdings[i].target_coverage());                         // ❽
}
</script>
</head>
<body>
<h1>Basic call to the OpenSRF-over-HTTP translator</h1>
</body>
</html>
```

❶ Load the Dojo core.

2. `fieldmapper.AutoIDL` reads `/openils/var/reports/fm_IDL.xml` to generate a list of class properties.
3. `fieldmapper.dojoData` seems to provide a store for Evergreen data accessed via Dojo.
4. `fieldmapper.Fieldmapper` converts the list of class properties into actual classes.
5. `fieldmapper.standardRequest` invokes an OpenSRF method and returns an array of objects.
6. The first argument to `fieldmapper.standardRequest` is an array containing the OpenSRF service name and method name.
7. The second argument to `fieldmapper.standardRequest` is an array containing the arguments to pass to the OpenSRF method.
8. As Fieldmapper has instantiated the returned objects based on their class hints, we can invoke getter/setter methods on the objects.

6 License