Advanced Customisation: Scripting EPrints
Part 2: Scripting Techniques
Roadmap

Core API

- manipulating your data
- accessing data collections
- searching your data

Scripting techniques

- essentials – putting it all together
- writing export plugins
- writing screen plugins
- writing command-line tools
- writing CGI scripts
Scripting Techniques: Essentials
Putting it all together

Two essential objects

- **Session**
  - connects to the repository
  - many useful methods

- **Repository**
  - provides access to
    - datasets
      - `session->get_repository->get_dataset("archive")`
  - configuration settings

Explore using `perldoc`
API provides lots of methods to help you build Web pages and display (render) data

- these methods return (X)HTML
  - but not strings!
- XML DOM objects
  - DocumentFragment, Element, TextNode...

Build pages from these nodes

- node1->appendChild(node2)

why? it’s easier to manipulate a tree than to manipulate a large string
XML DOM vs. Strings

p = make_element("p")
text = make_text("Hello World")
p->appendChild(text)

p = "<p>"
p += "Hello World"
p += "</p>"

Can manipulate tree to add extra text, elements etc.

Hello World

Difficult to make changes to the string – would need to find the right position first
Render Methods: Session

- Session provides many useful Web page building blocks
  - `make_doc_fragment()`
    - create an empty XHTML document
    - fill it with things!
  - `make_text(text)`
    - create an XML TextNode
  - `make_element(name, attrs)`
    - create an XHTML Element
      ```
      make_element("p", align => "right")
      <p align="right" />
      ```
render_link(uri, target)

create an XHTML link

link = session->render_link("http://www.eprints.org")

text = session->make_text("EPrints")

link->appendChild(text)

<a href="http://www.eprints.org">EPrints</a>
html_phrase(phraseid, inserts)

- render an XHTML phrase in the current language
- looks up phraseid from the phrases files
- inserts can be used to add extra information to the phrase
  - must be a corresponding <epc:pin> in the phrase
- <epp:phrase>Number of results: <epc:pin name="count"/></epp:phrase>
Many methods for building input forms, including:

- `render_form(method, dest)`
- `render_option_list(params)`
- `render_hidden_field(name, value)`
- `render_upload_field(name)`
- `render_action_buttons(buttons)`
- ...

Render Methods: Session (4)
Rendering Methods: Data Objects

- `render_citation(style)`
  - create an XHTML citation for the object
  - if `style` is set then use the named citation style
- `render_citation_link(style)`
- `render_value(fieldname)`
  - get an XHTML fragment containing the rendered version of the value of the named field
  - in the current language
Rendering Methods: MetaFields

- `render_name(session)`
- `render_help(session)`

- get an XHTML fragment containing the name/description of the field in the current language
Rendering Methods: Searches

- `render_description()`
  - get some XHTML describing the search parameters

- `render_search_form(help)`
  - render an input form for the search
  - if help is true then this also renders the help for each search field in current language
Session object also provides useful methods for getting user input
  e.g. from an input form
  `have_parameters`
    true if parameters (POST or GET) are available
  `param(name)`
    get the value of a named parameter
Scripting Techniques: Writing Export Plugins
EPrints provides a framework for plugins
- registration of plugin capabilities
- standard interface which plugins need to implement

Several types of plugin interface provided
- import and export
  - get data in and out of the repository
- interface screens
  - add new tools and reports to UI
- input components
  - add new ways for users to enter data
Not just a plugin framework for 3rd party extensions!

Used extensively by EPrints itself

- majority of (dynamic) Web pages you see are screen plugins
  - search, deposit workflow, editorial review, item control page, user profile, saved searches, administration tools...
- all import/export options implemented as plugins
- all input components in deposit workflow are plugins
  - subject browser input, file upload...
EPrints is really a generic plugin framework with a set of plugins that implement the functions of a repository.

Gives plugin developers many examples to work from:
- find a plugin that does something similar to what you want to achieve and explore how it works.

Diagram:
- Plugins
- Plugin Framework
- Backend (data model)
Writing Export Plugins

Typically a standalone Perl module in
perl_lib/EPrints/Plugin/Export/

Writing export plugins

- register plugin
- define how to convert data objects to an output/interchange format
Register

- name
  - the name of the plugin
- visible
  - who can use it
- accept
  - what the plugin can convert
    - lists of data objects or single data objects (or both)
    - type of record (eprint, user...)
- suffix and mimetype
  - file extension and MIME type of format plugin converts to
Registration Example: BibTeX

$self->{name} = "BibTeX";
$self->{accept} = [ 'list/eprint',
                   'dataobj/eprint' ];
$self->{visible} = "all";
$self->{suffix} = ".bib";
$self->{mimetype} = "text/plain";

- Converts lists or single EPrint objects
- Available to all users
- Produces plain text file with .bib extension
Registration Example: FOAF

$self->{name} = "FOAF Export";
$self->{accept} = [ 'dataobj/user' ];
$self->{visible} = "all";
$self->{suffix} = ".rdf";
$self->{mimetype} = "text/xml";

- Converts single User objects
- Available to all users
- Produces XML file with .rdf extension
Registration Example: XML

$self->{name} = "EP3 XML";
$self->{accept} = [ 'list/*', 'dataobj/*' ];
$self->{visible} = "all";
$self->{suffix} = "xml";
$self->{mimetype} = "text/xml";

- Converts any data object
- Available to all users
- Produces XML file with .xml extension
For a straight conversion plugin, this usually includes:

- mapping data objects to output/interchange format
- serialising the output/interchange format

E.g. EndNote conversion section:

```php
$data->{K} = $dataobj->get_value( "keywords" );
$data->{T} = $dataobj->get_value( "title" );
$data->{U} = $dataobj->get_url;`
But export plugins aren’t limited to straight conversions!

Explore:

Google Maps export plugin
- plot location data on map
  - http://files.eprints.org/224/

Timeline export plugin
- plot date data on timeline
  - http://files.eprints.org/225/
Register

- subclass `EPrints::Plugin::Export`
  - inherits all the mechanics so you don’t have to worry about them
  - could subclass existing plugin e.g. XML, Feed
- define name, accept, visible etc.
  - in constructor `new()` of plugin module

Conversion

- define `output_dataobj` function
  - will be called by plugin subsystem for every data object that needs to be converted
Writing Import Plugins

- Typically a standalone Perl module in `perl_lib/EPrints/Plugin/Import/`
- Reading input can be harder than writing output
  - Need to detect and handle errors in input
  - Many existing libraries available for parsing a wide variety of file formats
- Writing import plugins
  1. Register
  2. Define how to convert input/interchange format into data objects
     - Reverse of export
Scripting Techniques:
Writing Screen Plugins
Plugins: Writing Screen Plugins

One or more Perl modules in

```
perl_lib/EPrints/Plugin/Screen/
```

may be bundled with phrases, config files, stylesheets etc.

Writing screen plugins

- register
  - where it appears in UI
  - who can use it
- define functionality
Register

- actions
  - the actions the plugin can carry out (if any)
- appears
  - where abouts in the interface the plugin and/or actions will appear
    - named list
    - position in list
- will be displayed as link, button or tab
Registration Example: Manage Deposits

$self->{appears} = [
    { place => "key_tools", position => 100, },
];
Registration Example: EPrint Details

$self->{appears} = [
    { place => "eprint_view_tabs", position => 100, },
];
Registration Example: New Item

```
$self->{appears} = [
    { place => "item_tools", position => 100,
      action => "create", },
];
```

`item_tools list (create action will be invoked when button pressed)`
3 types of screen plugin

- Render only
  - define how to produce output display
  - examples: Admin::Status, EPrint::Details

- Action only (no output display)
  - define how to carry out action(s)
  - examples: Admin::IndexerControl, EPrint::Move, EPrint::NewVersion

- Combined (interactive)
  - define how to produce output/carry out action(s)
  - examples: EPrint::RejectWithEmail, EPrint::Edit, User::Edit
Action plugins produce no output display but can still display messages to user

```
add_message(type, message)
```

register a message that will be displayed to the user on the next screen they see

- type can be
  - error
  - warning
  - message (informational)
Register
- subclass EPrints::Plugin::Screen
  - inherits all the mechanics so you don’t have to worry about them
  - could subclass existing plugin e.g. EPrint, User
- define where plugin appears
  - in constructor new() of plugin module
- define who can view plugin (if required)
  - can_be_viewed function
    - e.g. check user privileges

Define functionality
- define render function
  - produce output display using API render_methods
Screen Plugin Template: Action Only

Register
- subclass EPrints::Plugin::Screen
- define actions supported
- define where actions appear
- define who can use actions
  - allow_ACTION function(s)

Define functionality
- define action_ACTION function(s)
  - carry out the action
  - use add_message to show result/error
  - redirect to a different screen when done
render function usually displays links/buttons which invoke the plugin’s actions

- e.g. EPrint::Remove
- registers remove and cancel actions
- render function displays *Are you sure?*
- screen
  - OK/Cancel buttons invoke remove/cancel actions
Scripting Techniques: Writing Command Line Scripts
Command Line Scripts

- Usually stored in `bin` directory
- Add batch/offline processes to your repository
  - e.g. duplicate detection – compare each record to every other record
  - e.g. file integrity - check stored MD5 sums against actual MD5 sums
Command line scripts (and CGI scripts) must explicitly connect to the repository by creating a new Session object:

- `new(mode, repositoryid)`
  - set mode to 1 for command line scripts
  - set mode to 0 for CGI scripts

And disconnect from the repository when complete:

- `terminate()`
  - performs necessary cleanup
XHTML is good for building Web pages

but not so good for command line output!

often no string equivalent

use tree_to_utf8()

extracts a string from the result of any rendering method

tree_to_utf8(
    eprint-&gt;render_citation)
Search and Modify Template

Common pattern for command line tools

- Connect to repository
- Get desired dataset
- Search dataset
- Apply function to matching results
  - modify result
  - commit changes
- Disconnect from repository
Example: lift_embargos

- Removes access restrictions on documents with expired embargos
  - Connect to repository
  - Get document dataset
  - Search dataset
    - embargo date field earlier than today’s date
  - Apply function to matching results
    - remove access restriction
    - clear embargo date
    - commit changes
  - Disconnect from repository
Scripting Techniques: Writing CGI Scripts
**CGI Scripts**

- Usually stored in `cgi` directory
- Largely superceded by screen plugins but can still be used to add e.g. custom reports to your repository
- Similar template to command-line scripts but build Web page output using API `render__methods`
In Screen plugins, mechanics of sending Web pages to the user’s browser are handled by the plugin subsystem.

- need to do this yourself with CGI scripts
- methods provided by the Session object

```
build_page(title, body)
```

- wraps your XHTML document in the archive template

```
send_page()
```

- flatten page and send it to the user
Summary

✔ Use the core API to manipulate data in the API
  ✔ individual data objects
    ✔ EPrint, Document, User
  ✔ sets of data objects
    ✔ DataSet, List, Search

✔ Wrap this in a plugin or script
  ✔ Session, Repository
  ✔ Web output using render_ methods
  ✔ templates

⇒ Next: **hands-on** exercises designed to get you started with these techniques