AtoM Feature Development

An intro on how to create and contribute

Steve Breker, May 2017
AtoM Camp SJC
Feature Development Process
Feature development process overview

1. Feature idea
2. Technical classification
3. Development preparation
4. Creating a feature
5. Contributing a feature
1. Feature idea

- New archival standard?
- New theme?
- New way to bring data in or out?
- ???
2. Technical classification

- Plugins
- CLI tasks
- Background jobs
- Core features
2. Technical classification

- **Plugins** <- probably the most common
- CLI tasks
- Background jobs
- Core features
3. Preparation: design

- Think of broad use-cases
- Follow known open standards whenever relevant
- Implement as simply as possible
- Change as little as possible to get the functionality you want
- Avoid breaking backwards compatibility
- Run your ideas by the community
3. Preparation: technical

- Read up on Symfony 1.4
- Explore AtoM’s codebase
- Ask questions in the AtoM user forum
4. Developing your feature

- Fork AtoM on GitHub to have your own repository to work with
- Follow AtoM coding standards: https://wiki.accelstomemory.org/Development/Coding_standard
- Review our wiki page on contributing code: https://wiki.accelstomemory.org/Development/Contribute_code
- Make sure any third-party code libraries you add are AGPL v3 compatible
5. Contributing your feature

1. Submit a pull request on GitHub with your work
2. Respond to feedback in the pull request until the pull request’s approved:
   https://wiki.accesstomemory.org/Development/Code_review
3. Fill in a Contributor’s agreement:
   https://wiki.accesstomemory.org/Development/Contribute_code
*/
* This file is part of the Symfony package.
* (c) 2004-2006 Fabien Potencier <fabien@potencier.org>
* For the full copyright and license information, please view the LICENSE
* file that was distributed with this source code.
*/

class SymfonyFormatter extends Symfony {  
  protected  
  static $styles = array(  
    'ERROR' => array('fg' => 'red', 'bg' => 'white', 'bold' => true),  
    'SUCCESS' => array('fg' => 'green', 'bold' => true),  
    'COMMENT' => array('fg' => 'yellow'),  
    'WARNING' => array('fg' => 'cyan'),  
  );  
  
  public static function setStyle($name, $options = array())  
  {  
    $this->styles[$name] = $options;  
  }  
  
  public function format($text = '', $parameters = array())  
  {  
    if (is_array($parameters))  
    {  
      return $text;  
    }  
    if (is_array($parameters) && isset($this->styles[$parameters]))  
    {  
      $parameters = $this->styles[$parameters];  
    }  
    $styles = array();  
    
    return $text;
}
Symfony 1.4 overview

- Symfony 1.4 is the web application framework AtoM is built with
- Symfony follows the MVC (model/view/controller) pattern
- Models represent types of data
- Views represent how data is rendered
- Controllers represent logic that determines what data ends up being rendered by the view
Symfony models

- AtoM’s Symfony is using an ORM called Propel
- To define models, data schemas are defined using YAML files
- A CLI tool is run that uses these YAML files as a guideline to generate PHP code that defines model classes
- These classes are referred to as “base” models and aren’t supposed to be manually altered
- Model characteristics can be added or overridden by creating child classes that extend the base models
- Migrations handle changes between schema versions
Symfony controllers

- There are two main types of controllers: actions and components
- Actions define the behaviour of pages
- Components define logic shared between actions
- Both types of controllers are represented by classes
- Actions extend `sfAction` (or a child class)
- Components extend `sfComponent` (or a child class)
- If actions or components are part of a plugin they are given a class name that includes the plugin name
- Example: `sfRadPluginEditAction`
Symfony views

- There are two main types of views: page templates and partials
- Page templates define how an action’s data is rendered
- Partialss have multiple uses:
  - They can be used to define how a component’s data is rendered
  - They can be used by page templates to render repeating data
  - They can be used by page templates to encapsulate a complex part of a page
- The `use_helper` function can be used in page templates to include functions intended to be used within templates (for rendering dates and URLs, etc.)
Symfony routing

- AtoM’s routing is largely defined in `apps/qubit/config/routing.yml`
- Plugins can dynamically add routes as well
- Example: the arRestApiPlugin’s configuration class adds routes
Symfony debugging/development tools

- The clear cache CLI task is useful during development:
  ```
  php symfony cc
  ```

- Also useful is the CLI task to purge all user-created data:
  ```
  php symfony tools:purge
  ```

- See Steve Breker’s presentation, slide 4, for how to enable debug mode
Developing Plugins
Plugin Development Overview

- Plugins can be used to implement optional features as mentioned earlier (support for individual archival standards, new themes, etc.)
- Plugins are also used in AtoM to encapsulate functionality (the arElasticSearchPlugin plugin for example)
- Plugins can also be used to add new classes that other plugins can share
Plugins have four optional subdirectories: config, lib, modules, and web.

- **config** is where plugin-related configuration files can be put.
- **lib** is where plugin code libraries, such as plugin-related classes, can be put.
- **modules** is where module-related code can be put.
- **web** is where plugin-related web assets are put.
Plugin configuration files

- Plugin configuration files are named using the convention: `<plugin name>Configuration.class.php`
- **An example:** `sfRadPluginConfiguration.class.php`
- **These files define a class named** `<plugin name>Configuration` **that inherits from Symfony's** `sfPluginConfiguration` **class**
- These files specify the plugin's name, version, etc.
- These files also can contain plugin initialization logic, etc.
Plugin library files

- Plugin library files are named with the plugin name included in the filename
- An example:
  
  `plugins/arOaiPlugin/lib/arOaiPluginComponent.class.php`
- This is merely a convention, however
- Any class that is put in the plugin's lib directory will be auto-loaded by AtoM once the cache is cleared
- The `sfHistoryPlugin` plugin is an example of a plugin whose sole functionality is encapsulated in a class in the plugin’s `lib` directory
Plugin modules

- Modules are an organizational unit used in Symfony to encapsulate a bunch of files related to a group of application web pages

    sfFacebookPlugin
    modules
    photos
    actions
    indexAction.class.php
    editAction.class.php
    templates
    indexSuccess.php
    editSuccess.php
Enabling/disabling Plugins

- Plugins that consist entirely of plugin code library, put in lib, don't need to be explicitly enabled.
- Plugins must contain a configuration file in their config subdirectory in order to be enabled by an admin.
- Enable or disable plugins using the sfPluginAdminPlugin/plugins page.
Developing Tasks
Task Development Overview

- Tasks are primarily used by advanced users and system administrators
- Tasks add command-line accessible features
- Task examples: cache clearing, bulk import/export, adding administrators, etc.
- Task code can be found in `lib/tasks`
## Developing Background Jobs

<table>
<thead>
<tr>
<th>Start date</th>
<th>End date</th>
<th>Job name</th>
<th>Job status</th>
<th>Info</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-12-12 02:23 PM</td>
<td>N/A</td>
<td>arGenerateFindingAidJob</td>
<td>Running</td>
<td></td>
<td>Artefactual</td>
</tr>
<tr>
<td>2014-12-12 02:20 PM</td>
<td>2014-12-12 02:23 PM</td>
<td>arGenerateFindingAidJob</td>
<td>Running</td>
<td></td>
<td>Artefactual</td>
</tr>
<tr>
<td>2014-12-12 02:03 PM</td>
<td>2014-12-12 02:03 PM</td>
<td>arGenerateFindingAidJob</td>
<td>Completed</td>
<td></td>
<td>Artefactual</td>
</tr>
<tr>
<td>2014-12-12 01:57 PM</td>
<td>2014-12-12 01:57 PM</td>
<td>arGenerateFindingAidJob</td>
<td>Completed</td>
<td></td>
<td>Artefactual</td>
</tr>
<tr>
<td>2014-12-12 11:22 AM</td>
<td>2014-12-12 11:22 AM</td>
<td>arGenerateFindingAidJob</td>
<td>Completed</td>
<td></td>
<td>demo</td>
</tr>
<tr>
<td>2014-12-12 11:21 AM</td>
<td>2014-12-12 11:22 AM</td>
<td>arGenerateFindingAidJob</td>
<td>Completed</td>
<td></td>
<td>demo</td>
</tr>
<tr>
<td>2014-12-09 02:15 PM</td>
<td>2014-12-09 02:15 PM</td>
<td>arGenerateFindingAidJob</td>
<td>Completed</td>
<td></td>
<td>Artefactual</td>
</tr>
</tbody>
</table>

*You may only clear jobs belonging to you.*
Background Job Development Overview

- Jobs are used to perform "heavy lifting" in the background of an AtoM instance.
- For example, if a user requests a CSV export of all descriptions the user will be informed that the export has started, but won't have to wait for the export to complete to get a web page response.
- Users can visit a webpage to see the status of their jobs, whether in-progress, completed, or failed.
- A job is analogous to a pizza delivery order: the doorbell will ring when the pizza arrives and when in doubt you can call the pizzaria to enquire about whether the pizza's done or not.