Rapido Documentation

Release 1.0

Makina Corpus

Oct 25, 2018

Contents

Happy hacking with Plone

What for?

Creating a small form able to send an email, or to store some data, generating some extra information about a page and inserting it wherever we want: with Plone these kind of tasks are complex for experts, and almost impossible for beginners.

rapido.plone allows any developer having a little knowledge of HTML and a little knowledge of Python to implement custom elements and insert them anywhere they want in their Plone site.

How?

The unique interface to build applications with rapido.plone is the **Plone theming tool**.

It implies it can be achieved in the **file system** or through the **inline theming editor**.

A Rapido application is just a part of our current theme; it can be imported, exported, copied, modified, etc. like the rest of the theme.

Moreover, we can use Diazo extensively to inject our application in the Plone layout easily.

Contents:

2.1 Plone the easy way

Creating a site with Plone is easy, but developing with Plone might seem daunting.

In fact it can be easy, and it does not necessarily involve learning a lot about various complex frameworks.

Here are the **basics of easy developing with Plone**.

2.1.1 Install it

Installing Plone is very easy (see the installation documentation).

2.1.2 Theme it

Diazo is the Plone theming engine. Diazo has a brilliant approach to theming: **it applies on top of Plone, not inside**.

Indeed, Plone produces content pages and Diazo can apply any theme on-the-fly to those pages. So we do not need to know anything about Plone's internal mechanisms to theme it.

Diazo only requires a **regular static theme** (HTML files, CSS, JS, etc.) and **some mapping rules** (specified in `rules.xml`) which allows to specify where each part of our Plone content pages must fit into our static design.

The Diazo theme can be built directly from the Plone interface in the Theming editor. The Plone 5 default theme (named Barceloneta) can be copied and we can modify whatever we want in this copy.

The copy can also be exported as a `.zip` file, and imported back into the same site (to restore a previous version), or on another site (for instance to deploy a new theme from development to production).

If we are not comfortable with managing our theme implementation in a web-based interface, we might also store it on our server in the Plone installation folder:

```
$INSTALL_FOLDER/resources/theme/my-theme
```

2.1.3 Extend it

Plone can be extended in two ways.

We can install add-ons developed by the Plone community.

And we can also create our own specific content types using Dexterity.

Dexterity is the Plone **content-type framework** and it allows to create new content-types through the Plone web interface.

Like with Diazo, we are able to export what has been created online, so we can import it again later or import it on another server.

2.1.4 Customize it

Once we have changed the design with Diazo, we might want to re-organize or enrich the content layout themselves.

Mosaic is the **perfect solution to manipulate the content layout**: we can move existing elements (like the title, description, etc.), but also add new ones.

Once a layout is created, it can be exported and copied in our Diazo `manifest.cfg` file so it can be available as a new layout for our users.

Diazo and Mosaic allows us to entirely control how information is displayed in our web site, but they do not allow changing the behavior of Plone, like adding new features, new dynamically computed information, etc.

It can be achieved with **Rapido** (as explained in *Tutorial*), with a very basic knowledge of HTML and Python (so, still, no need to learn about the different Plone frameworks).

Our Rapido developments are managed in our existing theme folder, so here again we can work online in the Plone theming editor, or in `/resources/theme` folder.

Rapido provides easy access to the Plone API. The Plone API gathers in one unique module many different Plone tools allowing to search for contents, create contents, access user profiles information, etc. It makes Plone internal features much more approachable, and developing with Rapido might be a good opportunity to discover Plone through its API.

2.1.5 And if we want. . .

It might be sufficient to cover almost anything we might need to implement in our Plone site.

But if at some point we feel comfortable enough with the Plone technical environment, and if we want to learn more, then we might consider creating our own Plone add-on.

Our add-on will handle our Diazo theme (including our Rapido developments), our Dexterity content-type definitions, and all our configuration settings.

It is properly documented in the Plone documentation, and the Plone training might also be very helpful.

2.2 Installation

Install Plone, then modify `buildout.cfg` to add Rapido as a dependency:

```
eggs =
    ...
    rapido.plone

[versions]
plone.resource = 1.2
```

Then, run your buildout:

```
$ bin/buildout -N
```

2.3 Principles

2.3.1 Creating a Rapido app

Here are the basic steps to create a Rapido app:

- go to the theme folder (in Plone setup / Theme if we prefer to work inline, or, if your prefer to work in the filesystem, it might be either in the `static` folder in your theme package, or in the `resources` folder in your Plone install if you do not have a custom package),

- add a new folder named `rapido`,

- in `rapido`, add a new folder named `myapp`.

That's it.

Now, we can implement our application in this folder. Here is a typical layout for a rapido app:

```
/rapido
    /myapp
        settings.yaml
        /blocks
            stats.html
            stats.py
            stats.yaml
            tags.html
            tags.py
            tags.yaml
```

Note: `settings.yaml` is not mandatory, but it allows defining access rights if needed.

Note: A Rapido application can also be located in a non-active theme (see *Application*)

The app components are `blocks`. A block is defined by a set of 3 files (HTML, Python, and YAML) located in the `blocks` folder.

The **YAML file** defines the elements. An element is any dynamically generated element in a block: it can be a form field (input, select, etc.), but also a button (`ACTION`), or even just a piece of generated HTML (`BASIC`).

The **HTML file** contains the layout of the block. The templating mechanism is super simple, elements are just enclosed in brackets, like this: `{my_element}`.

The **Python file** contains the application logic. It is a set of functions, each named for the element or the event it corresponds to.

For a `BASIC` element, for instance, we need to provide a function with the same name as the element; its return-value replaces the element in the block.

For an `ACTION`, we are supposed to provide a function with the same name as the element; in this case, it will be *executed* when a user clicks on the action button.

Here is a basic example:

- `rapido/myapp/blocks/simpleblock.yaml`:

```
elements:
    result: BASIC
    do_something:
        type: ACTION
        label: Do something
```

- `rapido/myapp/blocks/simpleblock.html`:

```
<p>the answer to life, the universe, and everything is {result}</p>
{do_something}
```

- `rapido/myapp/blocks/simpleblock.py`:

```
def result(context):
    return "<strong>42</strong>"

def do_something(context):
    context.app.log('Hello')
```

We can see our block by visiting the following URL:

http://localhost:8080/Plone/@@rapido/myapp/blocks/simpleblock

It works fine, but where is our Plone site now??

2.3.2 Inserting our block in a Plone page

To put our block somewhere in the Plone site, we use a Diazo rule:

```
<before css:content="#content-core">
    <include css:content="form" href="/@@rapido/myapp/blocks/simpleblock" />
</before>
```

Now, if we visit any page of our site, we will see our block.

Note: If we want to display it only in the _News_ folder, we would use `css:if-content`:

```
<before css:content="#content-core" css:if-content=".section-news">
    <include css:content="form" href="/@@rapido/myapp/blocks/simpleblock" />
</before>
```

See the Diazo documentation for more details.

But unfortunately, when we click on our "Do something" button, we are redirected to the original bare block.

To remain in the Plone page, we need to activate the `ajax` target in `rapido/myapp/blocks/simpleblock.yaml`:

```
target: ajax
elements:
    result: BASIC
    do_something:
        type: ACTION
        label: Do something
```

Now, when we click our button, the rapido block is reloaded inside the Plone page.

Instead of adding a block to an existing Plone view, we might need to provide a new rendering, mapped to a specific URL. We can do that by declaring our block as a Plone view in its YAML file:

```
view:
  id: my-custom-view
  with_theme: true
```

And then we call `@@my-custom-view` on any content, like:

> http://localhost:8080/Plone/news/@@my-custom-view

We can create as many views as we might need (like `@@subscribe`, `@@unsubscribe`, `@@stats,...`).

Note: Adding a lot of rapido rules in our main `rules.xml` is not ideal.

We might prefer to create a `rules.xml` file in our `rapido/myapp` folder, and include it in our main `rules.xml` file like this:

```
<xi:include href="rapido/myapp/rules.xml" />
```

2.3.3 Running Python code

Every function in our Python files takes a parameter named `context`. The context gives access to useful objects:

- `context.app`: the current rapido app,
- `context.block`: (if executed in a block context) the current block,
- `context.record`: (if executed in a record context) the current record,
- `context.request`: the current request to rapido (the sub-request, if called from Diazo),
- `context.parent_request`: the current page request (when called from Diazo),
- `context.portal`: the Plone portal object,
- `context.content`: the current Plone content object,
- `context.api`: the Plone API.

> **Warning:** context is not the usual context we know in Plone (like context in a ZPT template or a PythonScript, or self.context in a BrowserView).
>
> The Plone context is usually the current content. In Rapido we can obtain it using context.content.

This allows us to interact with Plone in many ways, for instance we can run catalog queries, create contents, change workflow status, etc.

Nevertheless, it will behave as expected:

- the code will always be executed with the current user's access right, so the appropriate Plone access restrictions will be applied,

- the CSRF policy will also be applied (for instance, a Plone operation marked as PostOnly would fail if performed in a GET request).

> **Note:** The code we put in our Python files is compiled and executed in a sandboxed environment (provided by zope.untrustedpython.interpreter).

To help us debugging our code, we can add:

```
debug: true
```

in our app settings.yaml file. Then we can add some log message in our code:

```
context.app.log("OK")
context.app.log({"something": 1)
```

and they will be display in both the server log and the browser's javascript console.

2.3.4 Storing and retrieving data

A rapido app provides a builtin storage service, based on Souper.

> **Note:** Souper is designed to store (and index) huge amounts of small data (it can easily store survey results, comments, ratings, etc., but it will not be appropriate for attached files for instance).

The Rapido storage service stores **records**, and records contain **items**.

There are 3 ways to create records in Rapido:

- we can create records by submitting a block: if a block contain some fields elements (like TEXT or NUMBER elements for instance), and if the block contains a *save* button (by adding {_save} in its layout), every time the user enters values in the fields and clicks save, the submitted values will be saved in a new record,

- we can create records by code:

```
record = context.app.create_record(id='myrecord')
```

- we can create records using the Rapido JSON REST API:

```
POST /:site_id/@@rapido/:app_id
Accept: application/json
{"item1": "value1"}
```

or:

```
PUT /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
{"item1": "value1"}
```

The same goes for accessing data:

- we can display records by calling their URL, and they will be rendered using the block they were created with:

```
/@@rapido/myapp/record/myrecord
```

- we can get a record by code:

```
record = context.app.get_record(id='myrecord')
some_records = context.app.search('author=="JOSEPH CONRAD"')
```

- we can get records using the Rapido JSON REST API:

```
GET /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
```

2.3.5 Integration with Plone

In addition to the Diazo injection of Rapido blocks in our theme, we can also integrate our Rapido developments in Plone using:

- Mosaic: Rapido provides a Mosaic tile which enables us to insert a Rapido block in our page layout.
- Content Rules: Rapido provides a Plone *content rule action* allowing us to call a Python function from a block when a given Plone event happens.
- Mockup patterns: the *modal* and the *content loader* patterns can load and display Rapido blocks.

See *Displaying Rapido in Plone*.

2.4 Tutorial

How to build a content rating system in Plone in few minutes.

2.4.1 Objective

We want to offer to our visitors the ability to click on a *"Like"* button on any Plone content, and the total of votes must be displayed next to the button.

Note: There is a screencast covering the first steps of the Rapido tutorial.

2.4.2 Prerequisites

Run buildout to deploy Rapido and its dependencies (see *Installation*).

Install the `rapido.plone` add-on from Plone site setup.

2.4.3 Initializing the Rapido app

We go to Plone *Site setup*, and then *Theming*.

If our current active theme is not editable inline through the Plone web interface (i.e. there is no *"Modify theme"* button), we will first need to create an editable copy:
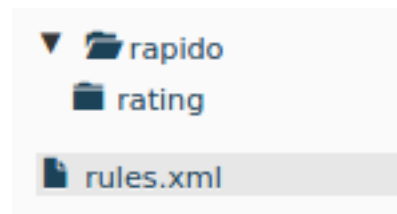
- click on *"Copy"*,

- enter a name, for example *"tutorial"*.

- check *"Immediately enable new theme"*.

Else, we just click on the *"Modify theme"* button.

We can see our theme structure, containing CSS files, images, HTML, and Diazo rules.

To initialize our Rapido app named "rating", we need to:

- create a folder named `rapido` in the theme root,
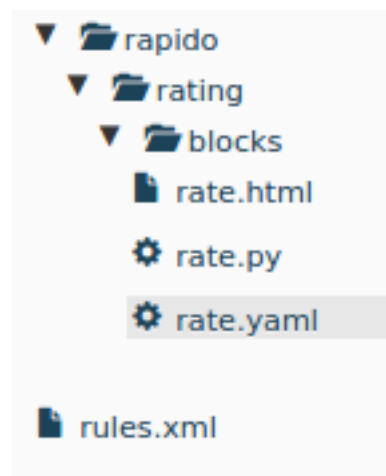
- in this `rapido` folder, create a folder named `rating`.

The application is now ready.

2.4.4 Creating the "Like" button

Rapido apps are composed of **blocks**. Let's create a block that will render our button:

- go to the `rating` folder and create a new folder named `blocks`,

- in this `blocks` folder, let's create a new block named `rate`. To do this, we need to create 3 files:

The `rate.html` file:

```
<i>If you like what you read, say it! {like}</i>
```

This allows us to implement the block layout. It is a regular HTML file, but it may contain Rapido **elements**, enclosed in brackets. In our case, we have one element, namely {like}, in charge of rendering the *"Like"* button.

The rate.py file

```python
def like(context):
    # nothing for now
    pass
```

It provides the element's implementation. Each element in the block has a corresponding Python function having the same id. In our case, that is the code that will be executed when a user clicks on *"Like"*. Right now, it does nothing, but we will change it later.

The rate.yaml file:

```yaml
elements:
    like:
        type: ACTION
        label: Like
```

This file contains all the needed settings for our block. Here we declare that our block contains one element named like, which is an **action** (i.e. it will be rendered as a button), and its displayed label is *"Like"*.

Now that our block is ready, we can see it using the following URL:

http://localhost:8080/Plone/@@rapido/rating/blocks/rate
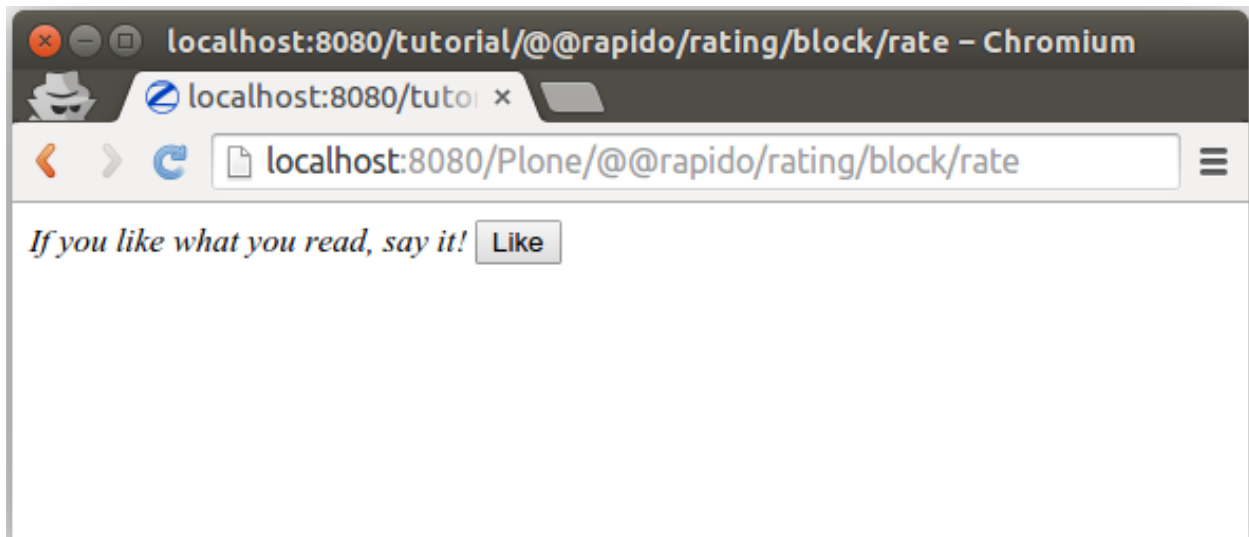

The next step is to embed our block in our Plone pages.

2.4.5 Inserting the block in Plone pages

To include our block somewhere in Plone, we will use a Diazo rule. Let's open our rules.xml file in the root of our theme, and add the following lines:

```xml
<after css:content=".documentFirstHeading">
    <include css:content="form" href="/@@rapido/rating/blocks/rate" />
</after>
```

The `include` directive allows us to retrieve a piece of content; in our case, the HTML form produced by our block. And the `after` directive inserts it after the main title in our page.

So, now if we visit any page of our Plone site, we see our block displayed just under the title.
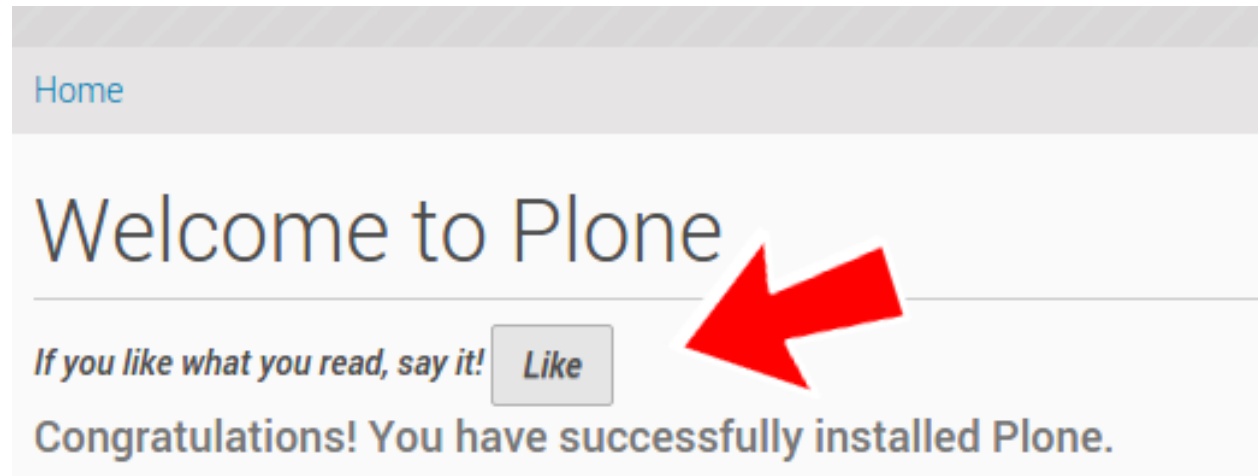
That is nice, but there is a small problem: when we click on the *"Like"* button, we are redirected to the raw block content, and we lose our current Plone page.

Let's fix that.

2.4.6 Staying in our Plone page

If we want to stay on our current page after submitting our block, we need to enable **AJAX** mode.

To do this, let's change our `rate.yaml` file like this:

```
target: ajax
elements:
    like:
        type: ACTION
        label: Like
```

Now, if we click on the *"Like"* button, the block is just reloaded dynamically, and we stay in our current page.

2.4.7 Counting the votes

Let's go back to `rate.py`, and focus on the `like` function implementation.

When a user clicks on the *"Like"* button, we need to get the current content the user voted for, check how many votes it already has, and add one new vote.

Rapido allows to create **records**, so we will create a record for each content item, and we will use the content path as an id.

So let's replace our current implementation with:

```
def like(context):
    content_path = context.content.absolute_url_path()
    record = context.app.get_record(content_path)
    if not record:
```

<div align="right">(continues on next page)</div>

```
        record = context.app.create_record(id=content_path)
    total = record.get('total', 0)
    total += 1
    record['total'] = total
```

`context.content` returns the current Plone content, and `absolute_url_path` is a Plone method returning the path of a Plone object.

`context.app` allows to access to the current Rapido app, so we can easily use the Rapido API, like `create_record` or `get_record`.

A Rapido record contains **items**. The `get(item, default=none)` method returns the value of the requested item or the default value if the item does not exist.

2.4.8 Displaying the votes

Now we are able to store votes, we also want to display the *total* of votes.

Fist, let's change the block layout in `rate.html`:

```
<p>{display}</p>
<p><i>If you like what you read, say it! {like}</i></p>
```

So now we have a new `display` element in our block.

We must declare it in `rate.yaml`:

```
target: ajax
elements:
    like:
        type: ACTION
        label: Like
    display: BASIC
```

And let's implement it in `rate.py`:

```
def display(context):
    content_path = context.content.absolute_url_path()
    record = context.app.get_record(content_path)
    if not record:
        return ''
    return "&#10084;" * record.get('total', 0)
```

We get the record corresponding to the current content, and we return as many as votes we have stored.
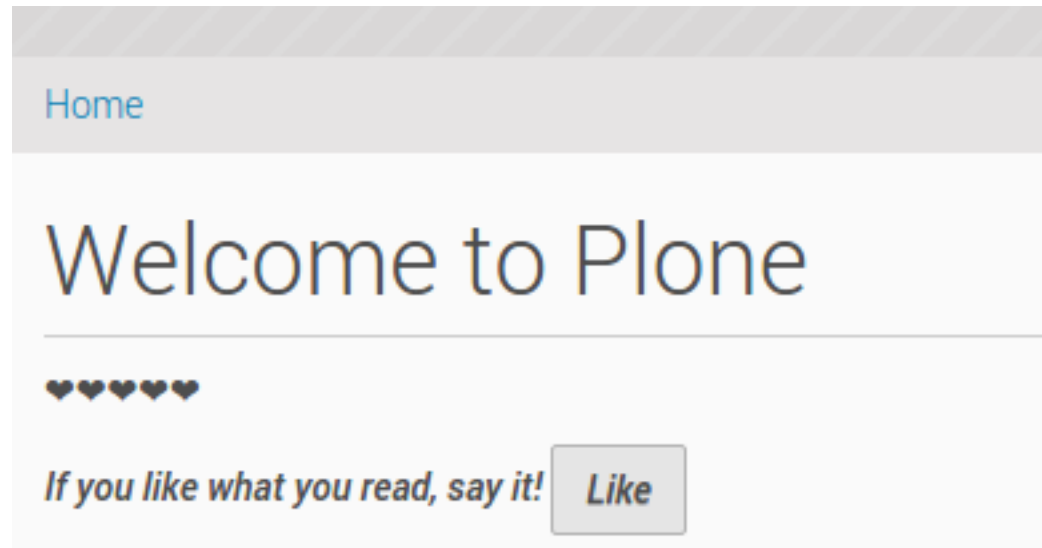
That's it! Our rating feature is ready to be used.

2.4.9 Debugging

As we are writing code, we might (we will) make mistakes. In this case, , it is always helpul to read the error messages returned by the system.

It is also very helpful to be able to log messages from our code, so we understand what is going on exactly when it is executed.

Rapido provides the `context.app.log()` method which will log string messages or any serializable object (dictionnaries, arrays, etc.).

The log messages and the error messages are visible in the server log (but we might not be able to access it), but also in our browser's **javascript console**.

First thing to do is to enable the **debug mode** in our app. To do that, we need to create a `settings.yaml` file in `/rapido/rating`:

```
debug: true
```

And now, let's change our `display` function:

```python
def display(context):
    content_path = context.content.absolute_url_path()
    record = context.app.get_record(content_path)
    if not record:
        return ''
    context.app.log(record.items())
    return "&#10084;" * record.get('total', 0)
```

We will see the following in our browser's console:

Let's imagine now we made an error like forgetting the colon at the end of the `if` statement:

```
def display(context):
    content_path = context.content.absolute_url_path()
    record = context.app.get_record(content_path)
    if not record
        return ''
    return "&#10084;" * record.get('total', 0)
```

Then we get this:

```
Rapido compilation error - rating:
in rate, at line 14: invalid syntax
    if not record
-----------------^

>
```

2.4.10 Listing the top 5 items

We would also like to see the top 5 rated content items on the site home page.

The first thing we need is to index the `total` element.

We declare its indexing mode in `rate.yaml`:

```
target: ajax
elements:
    like:
        type: ACTION
        label: Like
    display: BASIC
    total:
        type: NUMBER
        index_type: field
```

To index the previously stored values, we have to refresh the storage index by calling the following URL:

http://localhost:8080/Plone/@@rapido/rating/refresh

And to make sure future changes will be indexed, we need to fix the `like` function in the `rate` block: the indexing is triggered when we call the record's `save` method:

```
def like(context):
    content_path = context.content.absolute_url_path()
    record = context.app.get_record(content_path)
    if not record:
        record = context.app.create_record(id=content_path)
    total = record.get('total', 0)
    total += 1
    record['total'] = total
    record.save(block_id='rate')
```

We are now able to build a block to display the top 5 contents:

- `top5.html`:

```
<h3>Our current Top 5!</h3>
{top}
```

- `top5.yaml`:

```
elements:
    top: BASIC
```

- `top5.py`:

```python
def top(context):
    search = context.app.search("total>0", sort_index="total", reverse=True)[:5]
    html = "<ul>"
    for record in search:
        content = context.api.content.get(path=record["id"])
        html += '<li><a href="%s">%s</a> %d &#10084;</li>' % (
            content.absolute_url(),
            content.title,
            record["total"])
    html += "</ul>"
    return html
```

The `search` method allows us to query our stored records. The record ids are the content paths, so using the Plone API (`context.api`), we can easily get the corresponding contents, and then obtain their URLs and titles.

Our block works now:

> http://localhost:8080/Plone/@@rapido/rating/blocks/top5

Finally, we have to insert our block in the home page. That will be done in `rules.xml`:

```xml
<rules css:if-content=".section-front-page">
    <before css:content=".documentFirstHeading">
        <include css:content="form" href="/@@rapido/rating/blocks/top5" />
    </before>
</rules>
```

2.4.11 Creating a new page for reports

For now, we have just added small chunks of HTML in existing pages. But Rapido also allows us to create a new page (a Plone developer would name it a new *view*).

Let's pretend we want to create a report page about the votes on the content of a folder.

First, we need a block, `report.html`:

```html
<h2>Rating report</h2>
<div id="chart"></div>
```

We want this block to be the main content of a new view.

We need to declare it in a new YAML file called `report.yaml`:

```yaml
view:
    id: show-report
    with_theme: true
```

Now if we visit for instance:

> http://localhost:8080/Plone/@@show-report

we do see our block as main page content.

Now we need to implement our report content. We could do it with a Rapido element like we did in the Top 5 block.

Let's change our approach and implement a fancy pie chart using the amazing D3js library and the *Rapido REST API*.

We need to create a Javascript file (`report.js`) in the `/rapido/rating` folder:

```javascript
// Source: http://rapidoplone.readthedocs.io/en/latest/tutorial.html#creating-a-new-
↪page-for-reports
/* It is a feature of the RequireJS library
 * (provided with Plone by default) to load
 * our dependencies like:
 * - mockup-utils, which is a Plone internal resource,
 * - D3js (and we load it by passing its remote URL to RequireJS).
 */
require(['mockup-utils', '//d3js.org/d3.v3.min.js'], function(utils, d3) {
    /* Get the Plone getAuthenticator method
     * mockup-utils allows us to get the authenticator token
     * (with the getAuthenticator method), we need it to use
     * the Rapido REST API.
     */
    var authenticator = utils.getAuthenticator();
    // Get the local folders path
    var local_folder_path = location.pathname.split('/@@rapido')[0];
    // Get SVG element from the rapido block html named 'report.html'
    var width = 960,
        height = 500,
        radius = Math.min(width, height) / 2;

    /* d3.js Arc Generator
     * Generates path data for an arc (typically for pie charts).
     */
    var arc = d3.svg.arc()
        .outerRadius(radius - 10)
        .innerRadius(0);
```

(continues on next page)

```
    /* d3.js Pie Chart Generator
     * Generates data from an array of data.
     */
    var pie = d3.layout.pie()
        .sort(null)
        .value(function(d) { return d.value; });

    var svg = d3.select("#chart").append("svg")
        .attr("width", width)
        .attr("height", height)
        .append("g")
        .attr("transform", "translate(" + width / 2 + "," + height / 2 + ")");

    // d3.json() calls the Rapido endpoint @@rapido/rating/search (a rest api
→endpoint)
    d3.json("@@rapido/rating/search")
        // d3.json() puts the authenticator token in the X-Csrf-Token header,
        .header("X-Csrf-Token", authenticator)
        // and d3.json() passes the search query in the request BODY.
        .post(
            JSON.stringify({"query": "total>0"}),
            function(err, results) {
                var data = [];
                var color = d3.scale.linear().domain([0,results.length]).range(["
→#005880","#9abdd6"]);
                var index = 0;
                results.forEach(function(d) {
                    var label = d.items.id.split('/')[d.items.id.split('/').length -
→1];
                    data.push({
                        'i': index,
                        'value': d.items.total,
                        'label': label
                    });
                    index += 1;
                });

                // add arc element
                var g = svg.selectAll(".arc")
                    // call pie() function
                    .data(pie(data))
                    // add g element
                    .enter().append("g")
                    .attr("class", "arc");

                // add path element
                g.append("path")
                    .attr("d", arc)
                    .style("fill", function(d) { return color(d.data.i); });

                // add text element
                g.append("text")
                    .attr("transform", function(d) { return "translate(" + arc.
→centroid(d) + ")"; })
                    .attr("dy", ".35em")
                    .style("text-anchor", "middle")
```

```
                    .text(function(d) { return d.data.label; })
                    .style("fill", "white");
            }
        );
});
```

That is quite a complex script, and we will not detail here the D3js-related aspects (it is just a typical example to draw a pie chart); we will focus on the way we obtain the data.

The first thing to notice is the `require` function. It is a feature of the RequireJS library (provided with Plone by default) to load our dependencies.

We have 2 dependencies:

- `mockup-utils`, which is a Plone internal resource,

- D3js (and we load it by passing its remote URL to RequireJS).

`mockup-utils` allows us to get the authenticator token (with the `getAuthenticator` method), we need it to use the Rapido REST API.

Note:

- RequireJS or `mockup-utils` are not mandatory to use the Rapido REST API, if we were outside of Plone (using Rapido as a remote backend), we would have made a call to `../@@rapido/rating` which returns the token in an HTTP header. We just use them because they are provided by Plone by default, and they make our work easier.

- Instead of loading D3 directly form its CDN, we could have put the `d3.v3.min.js` in the `/rapido/rating` folder, and serve it locally.

The second interesting part is the `d3.json()` call:

- it calls the `@@rapido/rating/search` endpoint,

- it puts the authenticator token in the `X-Csrf-Token` header,

- and it passes the search query in the request BODY.

That is basically what we need to do whatever JS framework we use (here we use D3, but it could be a generalist framework like Angular, Backbone, Ember, etc.).

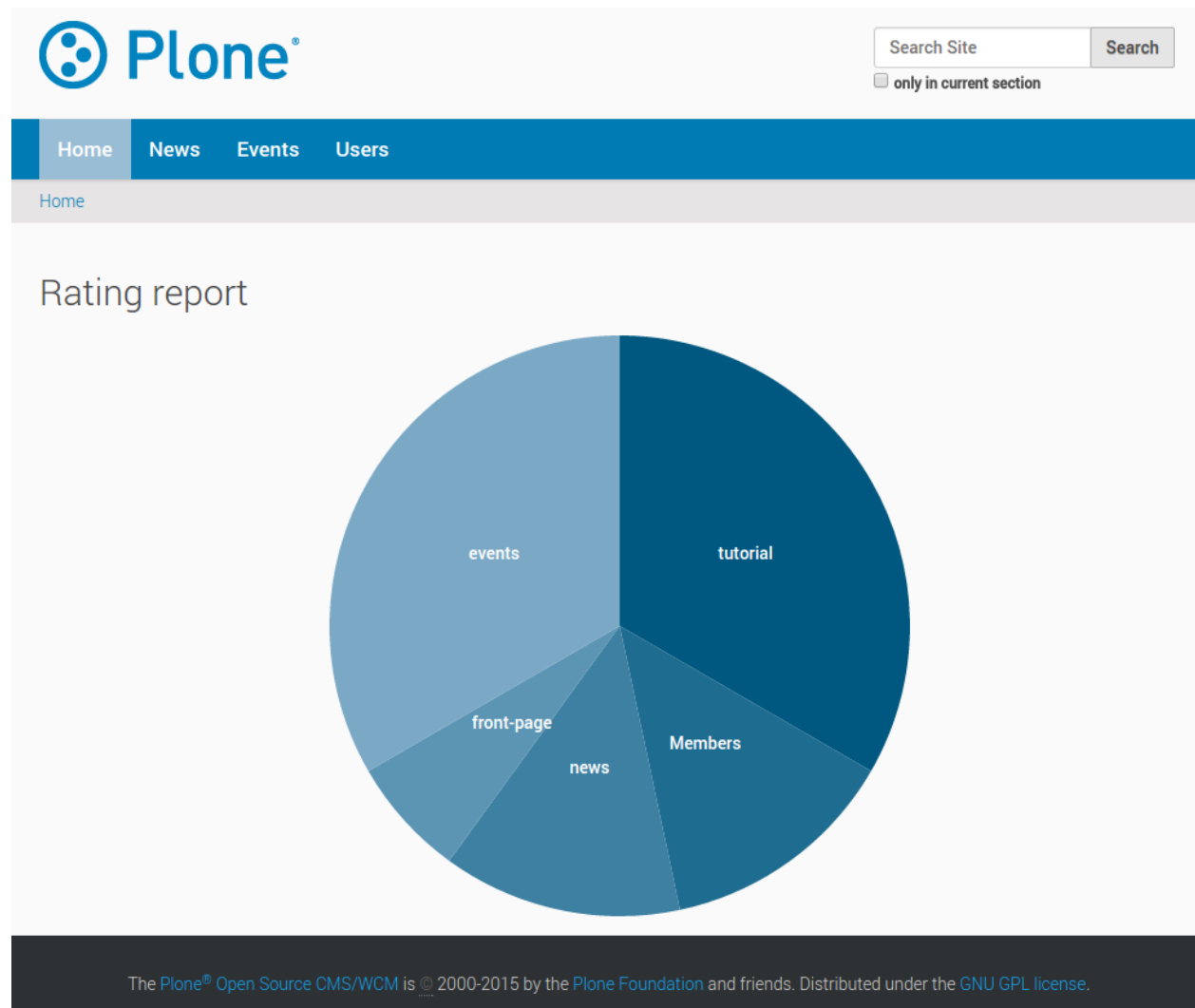Now we just need to load this script from our block:

```
<h2>Rating report</h2>
<div id="chart"></div>
<script src="++theme++tutorial/rapido/rating/report.js"></script>
```

And we can visit:

http://localhost:8080/Plone/news/@@show-report

to see a pie chart about votes on the *News* items!!

Download the `source files of this tutorial`.

Note: This .zip file can be imported in the theming editor, but it cannot be activated as a regular theme as it just contains our Rapido app. The app can be used from our main theme by adding a *rating.lnk* file in our current theme's `rapido` folder, containing:

```
tutorial
```

indicating the Rapido app named `rating` is stored in the theme named `tutorial`. And then we can activate our specific rules by adding:

```
<after css:content=".documentFirstHeading">
    <include css:content="form" href="/@@rapido/rating/blocks/rate" />
</after>

<rules css:if-content=".section-front-page">
    <before css:content=".documentFirstHeading">
        <include css:content="form" href="/@@rapido/rating/blocks/top5" />
    </before>
</rules>
```

in our main theme's `rules.xml`.

2.5 Reference

2.5.1 Application

A Rapido application is defined by a folder in the `rapido` folder in the current theme.

The application folder might contain a `settings.yaml` file in its root but that is not mandatory. It allows to define the access control settings (see *Access control*), or to enable the `debug` mode.

It always contains a `blocks` folder containing its blocks (see *Blocks*).

It might also contain regular theme items (`rules.xml`, CSS, Javascript, etc.).

Locating a Rapido application outside the current theme

If we use a lot of Rapido applications, or if the theme and the Rapido apps are managed by different persons, it might be preferable to locate the Rapido apps in a dedicated theme.

To do so, we just need to reference it using a `.lnk` text file in the current theme. The filename should be the app id, and its content must be the theme id.

For instance, our active theme would be structured like this:

```
/rapido
    myapp.lnk
```

The `myapp.lnk` content would be just:

```
dev-theme
```

The `dev-theme` theme would contain the full `myapp` Rapido app:

```
/rapido
    /myapp
        settings.yaml
        /blocks
            stats.html
```

And everything will work just like if the `myapp` folder was in our active theme.

2.5.2 Blocks

A block is defined by 3 files stored in the `blocks` folder of the application. Those files have the same filename (which is the block id) with the extensions `.html`, `.py` and `.yaml`.

The HTML file

The `.html` file contains the layout of the block. It is regular HTML. Dynamic elements are enclosed in curly brackets. Example:

```
<p>This is a dynamic message: {message}</p>
```

The curly brackets will be replaced by the corresponding element value.

If the element is a BASIC element and returns an object, we can access its properties. Example:

```
<h1>{my_doc.title}</h1>
```

Similarly, if a BASIC element returns a dictionnary, we can access its items. Example:

```
<p>{info[user]} said: {info[comment]}</p>
```

When rendered, the block layout is wrapped in an HTML `<form>` element.

The layout can contain Mockup patterns markup, they will be rendered as expected.

Some Mockup patterns might need to render actual curly brackets in the output. Double them to escape them:

```
<a href="#modal" class="pat-plone-modal"
    data-pat-modal='{{"content": "form"}}'>Display modal</a>
```

Once rendered, if the block contains some links with an `ajax` target:

```
<a href="@@rapido/record/1234" target="ajax">Open</a>
```

the request will be loaded in AJAX mode and its content will replace the current block content.

TAL template

The HTML template only offers element insertion. If we need more templating features, the `.html` file can be replaced by a `.pt` file, and we can use the TAL commands.

In the context of a Page Template, the block elements are available in the `elements` object:

```
def my_title(context):
    return "Chapter 1"
```

```
<h1 tal:content="elements/my_title"></h1>
```

Elements can be used as conditions:

```
def is_footer(context):
    return True
```

```
<footer tal:condition="elements/is_footer">My footer</footer>
```

If an element returns an iterable object (list, dictionary), we can make a loop:

```
def links(context):
    return [
        {'url': 'https://validator.w3.org/', 'title': 'Markup Validation Service'},
        {'url': 'https://www.w3.org/Style/CSS/', 'title': 'CSS'},
    ]
```

```
<ul>
    <li tal:repeat="link elements/links">
        <a tal:attributes="link/url"
            tal:content="link/title"></a>
    </li>
</ul>
```

The current Rapido context is available in the `context` object:

```
<h1 tal:content="context/content/title"></h1>
```

The YAML file

The `.yaml` file contains: - the elements settings (see below),

- the `target` option: if set to `ajax`, any action in the block resulting in a form submission will not redirect the current page, it will just refresh the block content through an AJAX call,

- the `view_permission` to manage who can see the block (see *Access control*).

The Python file

The `.py` file contains the implementation of each element as a Python function which name is the element id, and taking `context` as parameter.

2.5.3 Elements

Declaration

Elements must be declared in the YAML file under the `elements` entry. Every element is declared by its identifier, and its definition is:

- either a list of parameters, e.g.:

  ```
  elements:
      do_something:
          type: ACTION
          label: Do something
  ```

- or just a string, in which case Rapido will assume it is the `type` parameter, e.g.:

  ```
  elements:
      message: BASIC
  ```

  is equivalent to:

  ```
  elements:
      message:
          type: BASIC
  ```

Types

There are different types of elements (defined by the `type` parameter):

- `BASIC`: a piece of HTML returned by its implementation function.
- `ACTION`: a button that will execute the implementation function when clicked. Its label is provided by the `label` parameter.
- `TEXT`: a text input field.
- `NUMBER`: a number input field.
- `DATETIME`: a date/time input field.

Input elements

Input elements (i.e. `TEXT`, `NUMBER`, or `DATETIME`) can be indexed as `field` or `text`. Indexing is indicated using the `index_type` parameter.

By default, input elements are editable but they might also have a different `mode`:

- `COMPUTED_ON_SAVE`: the value is computed every time the record is saved,
- `COMPUTED_ON_CREATION`: the value is computed when the record is created.

Action elements

Action elements are rendered as submit buttons and allow to trigger a call to an associated Python function.

If the function returns a value, it must be a string, and it will be used as a redirection URL for the current request.

This is the way to redirect to another location once the action has been executed.

Builtin actions

The following actions can be included in our block HTML layout, and do not require an associated Python function:

- `_save`: creates a record based on the field elements' submitted values, and then redirects to the record display in read mode;
- `_edit`: opens the current record in edit mode;
- `_delete`: deletes the current record.

Direct HTTP call to elements

We usually want to display blocks, but we can also call an element by its URL:

> http://localhost:8080/Plone/@@rapido/myapp/blocks/block1/element1

Both GET and POST requests are supported.

If the element is an action, its Python function will be executed; the returned value is supposed to be a string and will be used as a redirection URL. When building an application, it allows us to create links that will redirect the user to the proper location depending on our business criteria (e.g. if the user belongs to group A, go to `page1`, else go to `page2`).

If the element is not an action, its Python function will be executed, and the result is returned as a response.

Note: We can change the response content type like this:

```python
def my_element(context):
    context.request.reponse.setHeader('content-type', 'text/csv')
    return "one,two,three\n1,2,3"
```

2.5.4 Records

Rapido records can be created by saving a block containing field elements. The value of each submitted element will be stored in a corresponding item.

In that case, the record has an associated block (the block id is stored in an item named `block`). When the record is rendered for display (when we load its URL in our browser), it uses the layout of the named block.

Records can also be created manually (without any associated block) using the Python API or the REST API. Such records cannot be rendered automatically by calling their URL, but their item values can be used in a block if we know how to find the record (in the *Tutorial* for instance, our records are created manually from the `like` function, they are not associated with the `rate` block, but we use the stored items to produce our element contents).

2.5.5 Associated Python functions

For a `BASIC` element, the associated Python function (having the same id) will return the content of the element.

For field elements (`TEXT`, `NUMBER`, `DATETIME`), the associated Python function will return its default value.

For an `ACTION` element, the associated Python function will be executed when the action is triggered.

Special Python functions

on_save Executed when a record is saved with the block. If it returns a value, it must be a string, and it will be used as a redirection URL for the current request.

on_display Executed when a block is displayed. It will be executed before all the element functions. It can be used to do some computation and put the result in the `context` so it can be accessed by the different elements. It can also be used to redirect to another page (using `context.request.response.redirect()`).

on_delete Executed when a record is deleted. If it returns a value, it must be a string, and it will be used as a redirection URL for the current request.

record_id Executed at creation time to compute the record id.

2.5.6 Indexing and searching

The Rapido storage system (souper) supports indexing.

Any block element can be indexed by adding a `index_type` setting in its YAML definition.

The `index_type` setting can have two possible values:

- `field`: such indexes match exact values, and support comparison queries, range queries, and sorting.
- `text`: such index matches contained words (applicable for text values only).

Queries use the (CQE format.

Example (assuming `author`, `title` and `price` are existing indexes):

```
context.app.search(
    "author == 'Conrad' and 'Lord Jim' in title",
    sort_index="price")
```

Records are indexed at the time they are saved. We can force reindexing using the Python API:

```
myrecord.reindex()
```

We can also reindex all the records using the `refresh` URL command:

> http://localhost:8080/Plone/@@rapido/<app-id>/refresh?_authenticator=<valid token>

or using the REST API (see *REST API*).

2.5.7 Displaying Rapido in Plone

We can see a block by visiting its URL:

> http://localhost:8080/Plone/@@rapido/myapp/blocks/simpleblock

Similarly for a record:

> http://localhost:8080/Plone/@@rapido/myapp/record/my-record-id

But it just returns the HTML generated by the block.

To show our blocks in our Plone site, there are 4 possibilities:

Diazo rules

We can use the Diazo `include` directive to get the Rapido block content and inject it in our pages with `before`, `after` or `replace` directives.

Example:

```
<before css:content="#content-core">
    <include css:content="form" href="/@@rapido/myapp/blocks/simpleblock" />
</before>
```

Extra views

> **Warning:** Since rapido.plone 1.1, we can declare first-class Plone views from Rapido, but it requires plone.resources 1.2.

If we do not want to just inject a small piece of HTML in existing pages, but use a Rapido block as the main part of the page, we can declare a block as a view in its YAML file:

```
view:
    id: my-custom-view
    with_theme: true
```

And then we call `@@my-custom-view` on any content, like:

> http://localhost:8080/Plone/news/@@my-custom-view

and it displays our block as main page content.

If the `with_theme` property is `false`, the page is rendered without the Plone theme (we just get the block by itself).

Extra views before 1.1

DEPRECATED since rapido.plone 1.1

If we do not want to just inject a small piece of HTML in existing pages, but create a new view for our contents, we can use the Rapido **neutral views**.

Neutral views are obtained by adding `@@rapido/view/<any-name>` to a content URL. It will just return the content's default view (that is why we call them neutral).

For instance, all those URLs display the same thing:

```
http://localhost:8080/Plone/front-page
http://localhost:8080/Plone/front-page/@@rapido/view/
http://localhost:8080/Plone/front-page/@@rapido/view/abc
http://localhost:8080/Plone/front-page/@@rapido/view/123
```

So we are able to call a content with a URL we control, and that allows us to create specific Diazo rules for it using the `if-path` attribute.

Hard-coded injection

```
<rules if-path="@@rapido/view/show-report">
    <replace css:content="#content">
        <include css:content="form" href="/@@rapido/stats/blocks/report" />
    </replace>
</rules>
```

In this example, if we open:

> http://localhost:8080/Plone/@@rapido/view/show-report

we will see our page main content replaced with our `report` block.

Dynamic injection

We can also dynamically display a Rapido resource specified in the URL. Rapido provides an URL injection pattern which allows to refer to the parent request in our Diazo rule.

The pattern is: `$<integer>`, where the integer specifies the starting position after `@@rapido` to get the path to inject.

For instance:

- with `http://localhost:8080/Plone/@@rapido/view/show-report/5654654`, `$3` gets the part of the path starting at the 3rd element after `@@rapido`, which is: `5654654`,

- with `http://localhost:8080/Plone/@@rapido/view/show-report/myapp/record/5654654`, `$3` gets the part of the path starting at the 3rd element after `@@rapido`, which is: `myapp/record/5654654`,

- with `http://localhost:8080/Plone/@@rapido/view/show-report/myapp/record/5654654/edit`, `$5` gets the part of the path starting at the 5th element after `@@rapido`, which is: `5654654/edit`.

Examples:

```
<rules if-path="@@rapido/view/show-report">
    <replace css:content="#content-core">
        <include css:content="form" href="/@@rapido/$3" />
    </replace>
</rules>
```

if we open:

> http://localhost:8080/Plone/@@rapido/view/show-report/myapp/record/my-record-id

we will render `myapp/record/my-record-id` in our page main content.

We could also do:

```
<rules if-path="@@rapido/view/show-report">
    <replace css:content="#content-core">
        <include css:content="form" href="/@@rapido/myapp/record/$3" />
    </replace>
</rules>
```

if we open:

> http://localhost:8080/Plone/@@rapido/view/show-report/my-record-id

we will get the very same rendering as in our previous example.

Mosaic

Mosaic is a layout editor.

It allows to add and manipulate *tiles* in our content layouts.

Rapido provides a Mosaic tile, so any Rapido block can be added as a tile to our layouts.

To enable it, we need to install Mosaic and then to import a specific Rapido Generic Setup profile named **"rapido.plone mosaic tile"** from the *ZMI >>> portal_setup >>> Import* and click on **"Import all steps"** button.

Here the **"Import"** page link from **portal_setup** tool for run Generic Setup profile:

> http://localhost:8080/Plone/portal_setup/manage_fullImport

Mockup patterns

Some Mockup patterns can display contents provided by an URL. The two main use cases are:

- **Display a Rapido block in a modal**: we use the `plone-modal` pattern on a `<a>` element, the Rapido block URL will be provided in its `href` attribute, and we just need to specify `form.rapido-block` as content selector (because `plone-modal` default content selector is `#content`, which is accurate for a Plone page but not for a Rapido block). Example:

  We create a block named `my-content` containing whatever we might need, and we create a block named `menu` containing the following HTML:

  ```
  <a href="@@rapido/my-app/blocks/my-content"
      class="plone-btn pat-plone-modal"
      data-pat-plone-modal="content: form.rapido-block">
          Open in a modal
  </a>
  ```

And then we just need to insert `menu` in our Plone page (using a Diazo rule).

See Mockup modal documentation for more details about the options.

- **Load a Rapido block dynamically in the current page**: we use the `plone-contentloader` to inject our Rapido block wherever we want. In our previous example, we would change the `menu` HTML to:

```
<a href="#" class="pat-contentloader"
   data-pat-contentloader="url:@@rapido/my-app/blocks/my-content#form.rapido-
↪block;">
   Load content</a>
```

It would replace the "Load content" link with our `my-content` block when we click on the link.

> **Warning:** with `plone-contentloader`, the content selector is passed directly as an hash at the end of the URL.

`plone-contentloader` also allows us to target a specific element for the injection (instead of replacing the link):

```
<a href="#" class="pat-contentloader"
   data-pat-contentloader="url:@@rapido/my-app/blocks/my-content#form.rapido-
↪block;target:#here;">
   Load content</a>
<p id="here">Insert my content here.</p>
```

By default, the injection is triggered by a click, but we can choose any DOM event (`mouseover` for instance), and we can even perform the injection immediately (using the `immediate` trigger):

```
<a href="#" class="pat-contentloader"
   data-pat-contentloader="url:@@rapido/my-app/blocks/my-content#form.rapido-
↪block;trigger:immediate">
   Load content</a>
```

2.5.8 Javascript

We can add Javascript files to our theme that will interact with our Rapido blocks.

There are no specific constraints on these scripts. Nevertheless, it might be handy to use the Javascript dependencies already provided by Plone, such as `jQuery` and `require`.

As Rapido allows to load block content dynamically (using the `ajax` mode), we might need to know when a Rapido block has been loaded dynamically.

To do that we can use the `rapidoLoad` event, which receives the block id as parameter. Example:

```
require(['jquery'], function($) {
    $(document).on('rapidoLoad', function(event, block_id) {
        console.log(block_id + ' has been loaded!');
    });
});
```

2.5.9 Import/export and source management

Rapido applications are implemented in the */rapido* folder of a Diazo theme. All the known development procedures for theming apply to Rapido development.

ZIP import/export

The Plone theming editor allows to export a Diazo theme as a ZIP file, or to import a new theme from a ZIP file.

That is the way we will import/export our Rapido applications between our sites.

Direct source editing

We might also store our Diazo themes on our server in the Plone installation folder:

```
$INSTALL_FOLDER/resources/theme/my-theme
```

That way, we can develop our Rapido applications using our usual development tools (text editor or IDE, Git, etc.).

Plone add-on

We can also create our own Plone add-on (see Plone documentation, and Plone training) and manage our Rapido applications in its theme folder.

2.5.10 Access control

Access control list

The ACL defined in the app applies to records, not to blocks. Blocks are always accessible, if we do not want a block to render some information, we have to implement this in its Python file or use the `view_permission` setting.

Moreover, access control only impacts direct HTTP access to records (like opening a record URL, deleting a record via the JSON API, etc.), and it does **not** impact what happens in block Python files.

For instance in the *Tutorial*, if an anonymous visitor clicks on the *"Like"* button on a page nobody had voted for yet, the `like` function will create a record.

But an anonymous visitor would not be able to modify this record or to delete it using the JSON API.

The expected format is:

```
acl:
  rights:
    reader: [<list of users or groups>]
    author: [<list of users or groups>]
    editor: [<list of users or groups>]
  roles: {<role_id>: [<list of users or groups>]}
```

In the list of users or groups, `'*'` means everyone.

Access levels

The access levels are:

- `reader`: can read all the records,
- `author`: can read all the records, can create records, can modify/delete his own records,
- `editor`: can read/modify/delete any record, can create records.

The access control settings are managed in the `settings.yaml` file in the app root folder.

Roles

Roles do not grant any specific rights on records, they can be defined freely. They are used in our Python functions to change the app behavior depending on the user.

For instance, we might have a role named 'PurchaseManager', and in our block we would display a "Validate purchase" button if the current user has the 'PurchaseManager' role.

Permissions on blocks

By default, blocks are accessible by anyone (including anonymous visitors).

By setting the `view_permission` attribute in a block's YAML file, we can control access to this block.

Its value is a list of users or groups.

Example:

```
elements:
  whatever: BASIC
view_permission:
  PurchaseDepartment
  eric
```

This block will be accessible only by the 'PurchaseDepartment' group members and Eric.

This restriction applies to direct block rendering and element calls, including REST calls.

2.5.11 Content rules

Content rules allows to trigger specific actions (for instance, send an email) when an given event (for instance, when new content is created in such and such a folder) happens in our Plone site.

Rapido provides a content rule action, so we can execute a Rapido function when an given event happens.

The action to take is defined in the Plone content rules editor (see the Plone content rules documentation), and requires the following parameters:

- the app id,
- the block id,
- the function name.

The `context.content` received by the function will be the content where the event happened.

2.5.12 External call to Rapido

By traversing to `@@rapido-call`, we can call a Rapido element as a Python function.

It might be very useful when we want to use Rapido from a PythonScript, a Plone page template, or any Plone mechanism offering to run a small script (Plone workflow, `collective.easyform`, etc.).

`@@rapido-call` accepts the following parameters:

- `path` (mandatory, string): Rapido path to the element to call (format: `app/blocks/element`),
- `content` (optional, object): the content to provide to the Rapido context,
- any other named parameters: those named parameters will be available to the Python implementation of the element in the `context.params` dictionary.

Example:

PythonScript:

```
visitors = container.restrictedTraverse('@@rapido-call')(
    'myapp/stats/analyse',
    content=portal.news,
    min_duration=2,
    client='smartphone')
```

Rapido element in `myapp/stats.py`:

```
def analyse(context):
    filtered = get_filtered_visitors(
        duration=context.params['min_duration'],
        type=context.params['client'])
    return len(filtered)
```

2.6 Python API

Any Rapido Python function receives `context` as parameter.

The `context` provides the following properties:

- *context.app*
- *context.request* and *context.parent_request*
- *context.portal*
- *context.content*
- *context.record*
- *context.api*
- *context.rapido*
- *context.modules*
- *Record*
- *Access control list*

2.6.1 `context.app`

This property gives access to the Rapido application object.

Properties

acl Returns the Rapido application's access control list object (see below).

blocks Returns the existing block ids.

indexes Returns the existing index ids.

url Returns the application URL.

Methods

create_record(self, id=None) Creates and return a new record. If id is not provided, a default one is generated. If id already exists, it is replaced with another one (like ...-1, ...-2).

delete_record(self, id=None, record=None, ondelete=True) Delete the record (which can be passed as object or id). If ondelete is truthy, the on_delete function will be called (if it exists) before deleting the record.

get_block(self, block_id) Returns a block.

get_record(self, id) Returns the record corresponding to the id, or None if it does not exist.

log(self, message) Logs a message in the server log. If the app is in *debug* mode, logs the same message in the browser's javascript console. Messages can be strings or any other serializable object.

records(self) Returns all the records as a list.

_records(self) Returns all the records as a Python generator.

search(self, query, sort_index=None, reverse=False) Performs a search and returns records as a list.

_search(self, query, sort_index=None, reverse=False) Performs a search and returns records as a Python generator.

2.6.2 `context.request` and `context.parent_request`

`context.request` is the actual request to Rapido, like:

> http://localhost:8080/Plone/@@rapido/rating/blocks/rate

When a block is embedded in a Plone page, `context.request` was issued by the user's browser, it was issued by Diazo.

To get the request issued by the user's browser, we use `context.parent_request`.

Both of them are HTTP requests objects, see the reference documentation.

Examples:

- Reading submitted values:

```
val1 = context.request.get('field_1')  # returns None if key doesn't exist
val1 = context.request['field_2']  # fail if key doesn't exist
```

- Reading the `BODY`:

```
request.get('BODY')
```

2.6.3 `context.portal`

Return the Plone portal object.

It is equivalent to:

```
context.api.portal.get()
```

The most common task we will perform through the portal object is to get its contents:

```
folder = context.portal['my-folder']
```

2.6.4 `context.content`

It returns the current Plone content.

The most common tasks we will perform on the content are:

- reading/writing its attributes (read/write):

```
the_tile = context.content.title
context.content.title = "I prefer another title"
```

- getting its URL:

```
context.content.absolute_url()
```

To manipulate the content, refer to the Plone API documentation about contents.

Note: Depending on its content type, the content object might have very different methods and properties.

2.6.5 `context.record`

It returns the current Rapido record if any.

See *Record* for more information.

2.6.6 `context.api`

It gives access to the full Plone API.

Warning: There is no need to import the API, as shown in all the Plone API examples:

```python
from plone import api  # WRONG
```

because the API is already available in the Rapido *context*:

```
catalog = context.api.portal.get().portal_catalog
```

This API mainly allows:

- to search contents; for example:

```
folders = context.api.content.find(portal_type="Folder")
# be careful, the find() method returns Brain objects, not real objects
# so only indexed attributes are available.
desc = folders[0].Description # OK
folders[0].objectIds() # WRONG!
folder = folders[0].getObject()
folder.objectIds() # OK!
```

- to manipulate contents (create / delete / move / publish / etc.), example:

```
new_page = context.api.content.create(
    type='Document',
    title='My Content',
    container=context.content)
context.api.content.transition(obj=new_page, transition='publish')
```

- to access or manage the users and groups information, and send emails. Example:

```
current_user = context.api.user.get_current()
context.api.portal.send_email(
    recipient=current_user.getProperty("email"),
    sender="noreply@plone.org",
    subject="Hello",
    body="World",
)
```

For more detailed examples, refer to the Plone API documentation.

2.6.7 `context.rapido`

`context.rapido` is a function able to obtain another Rapido application in our current script.

It takes as mandatory parameter the id of the Rapido application. Example:

```
purchase_app = context.rapido('purchase')
new_purchase_order = purchase_app.create_record()
```

It might also accept a `content` parameter to provide a specific content context to the app (if not provided, it will take the current content). Example:

```
stat_app = context.rapido('stats', content=context.portal.news)
```

2.6.8 `context.modules`

> **Warning:** For security reason, it is not allowed to import a Python module in a Rapido Python file.

Rapido provides some safe modules through `context.modules`:

- `context.modules.datetime`: Basic date and time types,
- `context.modules.random`: Generate pseudo-random numbers,
- `context.modules.time`: Time access and conversions.

If we need to add extra modules to `context.modules`, we can do it by adding in our own add-on something like:

```python
import re
from rapido.core import app

app.safe_modules.re = re
```

In this example, we allow to access `context.modules.re` from our Rapido Python files.

2.6.9 Record

Properties

url Returns the record URL.

id Returns the record identifier.

Methods

display(self, edit=False) Render the record using its associated block (if any).

get(self, name, default=None) Returns the value of the item (and defaults to `default` if the item does not exist).

items(self) Returns all the stored items.

reindex(self) Re-index the record.

save(self, request=None, block=None, block_id=None, creation=False) Update the record with the provided items and index it.

> `request` can be an actual HTTP request or a dictionnary.
>
> If a block is mentionned, formulas (`on_save`, computed elements, etc.) will be executed.
>
> If no block (and `request` is a dict), we just save the items values.

set_block(self, block_id) Assign a block to the record. The block will be then used to render the record or to save it.

Python dictionary-like interface

The record's items can be accessed and manipulated like dictionary items:

```python
myrecord['fruit'] = "banana"
for key in myrecord:
    context.app.log(myrecord[key])
if 'vegetable' in myrecord:
    del myrecord['fruit']
```

Note: When setting an item value, the record is not reindexed.

2.6.10 Access control list

Note: The application access control list can be obtain by `context.app.acl`.

Methods

---

current_user(self) Returns the current user id. Equivalent to:

```
context.api.user.get_current().getUserName()
```

current_user_groups(self) Returns the groups the current user belongs to. Equivalent to:

```
api.user.get_current().getGroups()
```

has_access_right(self, access_right) Returns `True` if the current user has the specified access right (Rapido access rights are `reader`, `author`, `editor`, `manager`)

has_role(self, role_id) Returns `True` if the current user has the specified role.

roles(self) Returns the existing roles.

2.7 REST API

2.7.1 Get the application settings

Request

```
GET /:site_id/@@rapido/:app_id
Accept: application/json
```

Response

```
{"no_settings": {}}
```

HTTP Response Headers

```
x-csrf-token: token
```

Returns the Rapido application settings and sets a token in the `X-CSRF-TOKEN` HTTP header value.

This HTTP header will have to be reused in all the requests made to the API (except for GET requests).

2.7.2 Authentication

Some of the operations below require authentication before they will run successfully. You will need to generate an Authorization String (A Base64 encoded version of your username and password separated by a dot).

Basic Authorization String

If your username is "john" and your password is "password", you can quickly generate the basic authorization string on the python prompt as follows:

```
>>> "john.password".encode('base64','strict').strip()
'am9obi5wYXNzd29yZA=='
```

Now you can use this header in all your requests:

```
Authorization: Basic am9obi5wYXNzd29yZA==
```

Note: The expected X-CSRF-TOKEN will be change when you use a Basic Authorization header.

2.7.3 Compute an element

Request

```
GET /:site_id/@@rapido/:app_id/blocks/:block_id/:element_id
Accept: application/json
X-CSRF-TOKEN: :token
```

or:

```
POST /:site_id/@@rapido/:app_id/blocks/:block_id/:element_id
Accept: application/json
X-CSRF-TOKEN: :token
```

Response

```
{"something": "bla"}
```

Returns the value returned by the element computation. The X-CSRF-TOKEN is not needed for a GET if the computation does not produce any change.

2.7.4 Get a record

Request

```
GET /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
X-CSRF-TOKEN: :token
```

Response

```
{"item1": "value1", "id": "boom"}
```

Returns the record items.

2.7.5 Get all the records

Request

```
GET /:site_id/@@rapido/:app_id/records
Accept: application/json
X-CSRF-TOKEN: :token
```

Response

```
[{"path": "http://localhost:8080/demo/@@rapido/test2/record/boom", "id": "boom",
→"items": {"bla": "bla", "id": "boom"}},
 {"path": "http://localhost:8080/demo/@@rapido/test2/record/10025657", "id": "10025657
→", "items": {"id": "10025657"}},
 {"path": "http://localhost:8080/demo/@@rapido/test2/record/9755269", "id": "9755269",
→ "items": {"bla": "bli", "id": "9755269"}},
```

```
 {"path": "http://localhost:8080/demo/@@rapido/test2/record/8742197835653", "id":
→"8742197835653", "items": {"bla": "bli", "id": "8742197835653"}},
 {"path": "http://localhost:8080/demo/@@rapido/test2/record/9755345", "id": "9755345",
→ "items": {"id": "9755345"}}]
```

Returns all the records.

2.7.6 Create a new record

Request

```
POST /:site_id/@@rapido/:app_id
Accept: application/json
X-CSRF-TOKEN: :token
{"item1": "value1"}
```

Response

```
{"path": "http://localhost:8080/demo/@@rapido/test2/record/9755269", "id": "9755269",
→"success": "created"}
```

Creates a new record with the provided items.

2.7.7 Create many records

Request

```
POST /:site_id/@@rapido/:app_id/records
Accept: application/json
X-CSRF-TOKEN: :token
[{"item1": "a"}, {"item1": "b", "item2": "c"}]
```

Response

```
{"total": 2, "success": "created"}
```

Bulk creation of records.

2.7.8 Create a new record by id

Request

```
PUT /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
X-CSRF-TOKEN: :token
{"item1": "value1"}
```

Response

```
{"path": "http://localhost:8080/demo/@@rapido/test2/record/boom", "id": "boom",
→"success": "created"}
```

Creates a new record with the provided items and having the specified id.

2.7.9 Delete a record

Request

```
DELETE /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
X-CSRF-TOKEN: :token
```

Response

```
{"success": "deleted"}
```

Deletes the record.

2.7.10 Remove all records

Request

```
DELETE /:site_id/@@rapido/:app_id/records
Accept: application/json
X-CSRF-TOKEN: :token
```

Response

```
{"success": "deleted"}
```

Remove all the records and delete the indexes.

2.7.11 Update a record

Request

```
POST /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
X-CSRF-TOKEN: :token
{"item1": "newvalue1"}
```

or:

```
PATCH /:site_id/@@rapido/:app_id/record/:record_id
Accept: application/json
X-CSRF-TOKEN: :token
{"item1": "newvalue1"}
```

Response

```
{"success": "updated"}
```

Updates the record with provided items.

2.7.12 Search for records

Request

```
POST /:site_id/@@rapido/:app_id/search
Accept: application/json
X-CSRF-TOKEN: :token
{"query": "total>0", "sort_index": "total"}
```

Response

```
[{"path": "http://localhost:8080/tutorial/@@rapido/rating/record//tutorial/news", "id
↪": "/tutorial/news", "items": {"total": 5, "id": "/tutorial/news"}},
 {"path": "http://localhost:8080/tutorial/@@rapido/rating/record//tutorial", "id": "/
↪tutorial", "items": {"total": 8, "id": "/tutorial"}}]
```

Search for records.

2.7.13 Re-index

Request

```
POST /:site_id/@@rapido/:app_id/refresh
Accept: application/json
X-CSRF-TOKEN: :token
```

Response

```
{"success": "refresh", "indexes": ["id", "total"]}
```
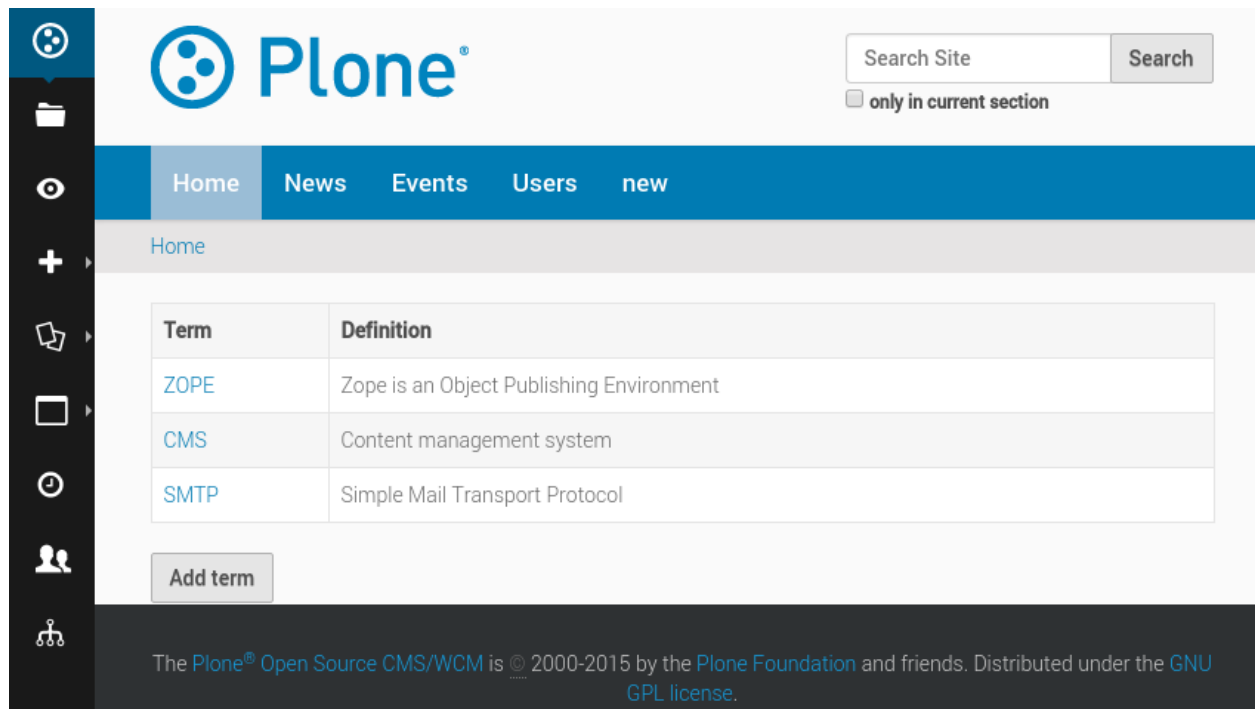
Re-declare the indexes and re-index all the records.

2.8 Use cases

2.8.1 Glossary use case

Objective

We want to provide a tool to manage a list of terms and their definitions:

Every time one of these terms appears on a page of our site, it will be wrapped into a *<abbr>* tag, the title of which will be the definition, so when we hover a term, we get a small popup indicating its definition:

Application structure

```
rapido/
    glossary/
        blocks/
            all.html
            all.py
            all.yaml
            term.html
            term.py
            term.yaml
        glossary.js
rules.xml
```

Rules.xml

```
<after css:theme-children="body">
  <script src="/tutorial/++theme++test/rapido/glossary/glossary.js"></script>
</after>
```

This rule inserts in all our pages a javascript file in charge of replacing matching words with <abbr> tags.

The `term` block

This block is a form allowing to create/edit/delete a glossary term. It contains two field elements and three actions.

- term.html

```
<p><label>Term</label> {term}</p>
<p><label>Definition</label> {definition}</p>
{_save} {_delete} {close}
```

- term.yaml

```
target: ajax
elements:
    term: TEXT
    definition: TEXT
    close:
        type: ACTION
        label: Close
    _save:
        type: ACTION
        label: Save
    _delete:
        type: ACTION
        label: Delete
```

- term.py

```
def close(context):
    return context.app.get_block('all').url

def on_save(context):
    return context.app.get_block('all').url

def on_delete(context):
    return context.app.get_block('all').url
```

If we click on any action in this block, we want to be redirected to the main management page. We do that by returning the `all` block URL (when an action returns a string, it is used as a redirection URL).

The `all` block

This block lists all the existing terms in a table. When we click on a term, we open it in the `term` block in edit mode, and a button allows opening a blank `term` block to create a new term.

- all.html

```html
<table class="listing"><tr><th>Term</th><th>Definition</th></tr>
{list}
</table>
{new_term}
```

- `all.yaml`

```yaml
target: ajax
view:
    id: glossary
    with_theme: true
elements:
    list: BASIC
    new_term:
        type: ACTION
        label: Add term
```

The `view` setting allows to render the `all` block as Plone view named `@@glossary`, so we can call http://localhost:8080/Plone/@@glossary to see it.

- `all.py`

```python
def list(context):
    html = u""
    for record in context.app.records():
        html += """<tr><td><a href="%s/edit" target="ajax">%s</a></td><td>%s</td>
↪</tr>""" % (
            record.url,
            record['term'],
            record['definition'],
        )
    return html


def new_term(context):
    return context.app.get_block('term').url
```

The `list` function builds a table row for each existing record, displaying the *term* value and the *definition* value. The link we put on the term targets the record URL (plus */edit* to open it in edit mode) and we have added *target="ajax"* so the resulting page is not displayed as a full page, it is just loaded into the current block in AJAX mode.

The Javascript

- `glossary.js`

```javascript
require(['jquery'], function($) {
    if($('.template-edit').length > 0) {
        return
    }
    $.getJSON('/tutorial/@@rapido/glossary/records', function(data) {
        var keys = [];
        var values = {};
        for(var i=0; i<data.length; i++) {
            term = data[i].items.term;
            definition = data[i].items.definition;
            keys.push(term);
            values[term] = definition;
```

(continues on next page)

```
        }
        var re = RegExp("(\\W)(" + keys.join("|") + ")(\\W)", "g");
        function replaceNodeText() {
         if (this.nodeType === 3) {
                var parent = $(this).parent();
                var html = parent.html();
                if(html) {
                    var newvalue = html.replace(re, function(){
                        var term = arguments[2],
                            before = arguments[1],
                            after = arguments[3];
                        term = '<abbr title="'+values[term]+'">'+term+'</abbr>';
                        return before + term + after;
                    });
                    parent.html(newvalue);
                }
            } else {
                $(this).contents().each(replaceNodeText);
            }
        }
        $("#content-core").contents().each(replaceNodeText);
    });
});
```

First thing we do is to check if we are in edit mode, and if we are, we stop, as we do not want to manipulate the HTML currently being edited in TinyMCE or in any input field.

Then we load the glossary terms with the following JSON call: `/tutorial/@@rapido/glossary/records`

Using the term values we have loaded, we build a regular expression able to match those terms in any text.

Then we iterate on the page main content (`#content-core`) elements, and every time we find a text node, we use our regular expression to replace the matching words with an `<abbr>` tag where the `title` attribute is the associated definition.

2.8.2 Use *Rapido* to create custom *SearchableText* field

Objective

Create TTW (Through The Web) a *Book* content type where fields are indexed for a full-text search.

Content type

Add a new content type *Book* on the *Dexterity Content Types* control panel (http://localhost:8080/Plone/@@ dexterity-types).

Edit the *Book* configuration (http://localhost:8080/Plone/dexterity-types/book). In the *Behaviors* tab, uncheck the *Dublin Core metadata* behavior and *Save*. In the *Fields* tab, click on the *Edit XML Field Model* button and replace the XML model by:

```xml
<model xmlns:form="http://namespaces.plone.org/supermodel/form"
       xmlns:i18n="http://xml.zope.org/namespaces/i18n"
       xmlns:lingua="http://namespaces.plone.org/supermodel/lingua"
       xmlns:marshal="http://namespaces.plone.org/supermodel/marshal"
       xmlns:security="http://namespaces.plone.org/supermodel/security"
       xmlns:users="http://namespaces.plone.org/supermodel/users"
       xmlns="http://namespaces.plone.org/supermodel/schema">
  <schema>
    <field name="title" type="zope.schema.TextLine">
      <description/>
      <title>Title</title>
    </field>
    <field name="authors" type="zope.schema.Text">
      <description/>
      <required>False</required>
      <title>Authors</title>
    </field>
    <field name="year" type="zope.schema.Int">
      <description/>
      <required>False</required>
      <title>Year</title>
    </field>
    <field name="isbn_13" type="zope.schema.TextLine">
      <description/>
      <max_length>13</max_length>
      <min_length>13</min_length>
      <required>False</required>
      <title>ISBN 13</title>
    </field>
    <field name="image" type="plone.namedfile.field.NamedBlobImage">
      <description/>
      <required>False</required>
      <title>Image</title>
```

(continues on next page)

```xml
      </field>
      <field name="back_cover" type="plone.app.textfield.RichText">
        <description/>
        <required>False</required>
        <title>Back cover</title>
      </field>
    </schema>
</model>
```

You can now add *Book* content in your web site (http://localhost:8080/Plone/++add++book).

Full text search field

If you have a lot of books in your site, you would like to search on the author's name or the back cover content. To do this we have to provide a `SearchableText` method or field which give the content of the full-text index. We'll use

a *rapido block* and a *rapido content rule action* to compute this `SearchableText` field.

Rapido block

Go to the *Theming* control panel (http://localhost:8080/Plone/@@theming-controlpanel). Create a new theme *MyTheme* with this structure.

```
index.html
manifest.cfg
rapido/
    book/
        blocks/
            fields.py
rules.xml
```

Look for example at the Inheriting a new theme from Barceloneta section of the Plone documentation for the content of `index.html`, `manifest.cfg` and `rules.xml` files.

Content of `fields.py` file:

```python
def update_searchabletext_field(context):
    transforms = context.api.portal.get_tool(name='portal_transforms')
    book = context.content
    back_cover_html = book.back_cover.output if book.back_cover else ""
    back_cover_plain = transforms.convertTo(
        'text/plain', back_cover_html, mimetype='text/html').getData()
    book.SearchableText = " ".join([
        book.title if book.title else "",
        book.authors if book.authors else "",
        str(book.year) if book.year else "",
        book.isbn_13 if book.isbn_13 else "",
        back_cover_plain
    ])
    book.reindexObject(idxs=['SearchableText'])
```

We use the `portal_transforms` tool to convert the `back_cover` HTML field to plain text. We also need to reindex the content.

Rapido content rule action

The last thing we need is a *rapido content rule action* which is used on each book modification.

Go to the *Content Rules* (http://localhost:8080/Plone/@@rules-controlpanel) and add a rule which is triggered on *Object modified* event.

Add a *Content type* condition on *Book*. Add a *Rapido action*.

Add Rapido Action   ✕

↵ Site Setup

A Rapido action executes a Rapido method.

Rapido application ● *The targeted Rapido application.*

book

Block ● *The block providing the method.*

fields

Method ● *The name of the method to execute.*

update_searchabletext_field

Save   Cancel

Assign the content rule on the whole site and *Save*.

Edit content rule

↵ **Up to rule management**

Rules execute when a triggering event occurs. Rule actions will only be invoked if all the rule's conditions are met. You can add new actions and conditions using the buttons below.

If all of the following conditions are met:

Perform the following actions:

⚠️   Edit   Remove   ↑   ↓

Content type Content types are: Book

⚠️   Edit   Remove   ↑   ↓

Rapido action Call Rapido method update_searchabletext_field from book/fields

Add condition

Content type ▾   **Add**

Add action

Logger ▾   **Add**

Assignments

ℹ️   **Info** This rule is assigned to the following locations: Site

Configure rule

Title

Please set a descriptive title for the rule.

Update book SearchableText field

Description

Enter a short description of the rule and its purpose.

Exercise

Modify the code above to compute a *Description* field which will be used in Plone listings.

Custom book view

To build a custom book view, the simplest solution is to use a *Diazo* rule.

For example, you can add in the `rules.xml` file of your theme the following `diazo rule`:

```xml
<rules css:if-content="body.template-view.portaltype-book">
  <replace css:content="#content-core" method="raw">
    <xsl:variable name="image_url">
      <xsl:value-of select="substring-before(//span[@id='form-widgets-image']/img/
→@src,'view')"/>
    </xsl:variable>
    <div class="row">
        <div class="col-xs-12 col-sm-4">
            <xsl:if test="$image_url">
                <img src="{$image_url}@@images/image/large" style="width: 100%;" />
            </xsl:if>
        </div>
        <div class="col-xs-12 col-sm-8">
            <div><strong>Author(s) : </strong><xsl:copy-of css:select="#form-widgets-
→authors" /></div>
            <div><strong>ISBN-13(s) : </strong><xsl:copy-of css:select="#form-widgets-
→isbn_13" /></div>
            <div><strong>Year : </strong><xsl:copy-of css:select="#form-widgets-year"␣
→/></div>
            <div><xsl:copy-of css:select="#formfield-form-widgets-back_cover" /></div>
        </div>
    </div>
  </replace>
</rules>
```

Our new custom book views:

2.9 License

Rapido Copyright 2015, Makina Corpus - Eric BREHAULT

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., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.