I.J. Modern Education and Computer Science, **2017, 11, 1-12**
Published Online November 2017 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2017.11.01

Generating Code for Simple Dynamic Web Applications via Routing Configurations

Kazuaki Kajitori
Department of Fisheries Distribution and Management, National Fisheries University, Shimonoseki, Yamaguchi, Japan
Email: kajitori@fish-u.ac.jp

Received: 04 August 2017; Accepted: 04 September 2017; Published: 08 November 2017

***Abstract*—**A dynamic web application tends to have many sorts of routine code which are essentially common in most dynamic web applications.

By the routing for a web application we mean mapping URLs (requests) to actions (responses) of the application. In this paper, we show that by configuring the routing for a web application together with preparing static libraries and modifying dynamic templates to generate the modules which are needed by the application, we can generate most of the routine code necessary for the application. Configurations for the routing and other settings for an application are written uniformly in JSON format. Then, reading the JSON configurations our scripts set up static libraries and generate necessary modules from templates and write files which conduct the routing.

Our system sets up everything you need to run a web application from the web directories for the application and the basic libraries to the modules for the application and the files for routing. You can add another web application to the web setting for the web application you have generated.

To show by example the applicability of our web code generating system we first construct a book request page with a very little configuration on the application. Next as a more complicated example, we apply our system to a web application which involves a specific function i.e. reading a marksheet for which the system cannot generate code, showing how the system can generate a web application which utilizes the code not generated by the system.

***Index Terms*—**Web application, Code generation, JSON, Routing, Configuration

I. Introduction

When we write a dynamic web application, we have to code a lot of routine processes which are essentially common in most dynamic web applications. Examples of such routine processes include authentication, session handling, constructions of some of CRUD (Create, Read, Update, Delete) interfaces to databases, uploading and downloading files and so on.

These common processes can be implemented by using static libraries or by code generated dynamically depending on parameters of the application. The former type of 'common code' often are provided as modules or packages in many languages. But those existing libraries are written rather in low level e.g. (for Perl) HttpAgent, CGI, RDB drivers, etc. Higher level helpers for web programming come as CMS (Contents Management System) like Movable Type, WordPress, etc, and web frameworks like CakePHP, Ruby on Rails, Spring, etc and other web code generation methods (including the method of this paper).

CMSs are handy to make a visual web site and are quite effective for static web sites whose main purposes are providing with static information.

For making a dynamic web site, one might choose a web framework. But for using a web framework, one needs to learn an elaborate mechanism for handling the framework which might as well be thought as yet another complex higher level language (which is valid only for the framework). Web frameworks may be indispensable for some big programming projects conducted by a group of programmers where the versatility of the programming scheme and a common background to every programmer are vital where the cost of learning the complex framework is regarded as reasonable.

For rather simple dynamic web applications like those the author and others have developed [1][2] for educational purposes where visual effects do not have the first priority and data model is simple, one may prefer to a more inexpensive scheme that may be restricted to simple web applications but allow simpler handling of it. Compared with other web code generation systems we will review in the next section, our system is web specific and is simpler and more practical in the sense that it constructs a concrete and complete environment and code for a dynamic web application.

By the routing for a web application we mean mapping URLs (requests) to actions (responses) of the application. The routing is the place where user requests, inner methods to handle requests and response methods (usually printing methods of an HTML page) gather and incorporate and thus for a web programmer can be the best place to overview the application. In this paper, we show a way to configure the routing of a dynamic web application in JSON (JavaScript Object Notation [10]) format to generate code for a dynamic web application.

The paper is organized as follows. In section II we review some of the related works. In section III we propose our scheme of generating code for a dynamic web application. In section IV we present an application of our code generating system to show how simple the configuration for a web code generation can be. In section V, as a more complicated example, we apply our code generating system to a web application which involves a specific function i.e. reading a marksheet. We state concluding remarks in section VI.

II. Related Works

In this section, we review papers on methods to generate web code which are usable in practice because we aim at a really usable method to generate web code.

We mentioned about web frameworks in the previous introduction and since their presence are not of academic nature, we here just refer to one of the pioneering works [3]. In [3], the authors introduced web design frameworks as a conceptual approach to maximize reuse in web applications which is a common task for all web programming tools.

In [4] the authors point out the difficulties of security implementation of web applications and presents a framework for developing secure web applications. Also the priority and importance of use case in UML are pointed out.

In [5] the authors propose a machinery for a code generator which is based on dynamic frames consisting of SCT (Specification, Configuration and Templates) elements. SCT generators dynamically generate code templates which contain source code in the target programming language together with connections (replacing marks for insertion of variable code parts). As template examples the authors show an HTML template [5] and an SQL template [6], in the former case PHP and other web frameworks have similar (and more practical) mechanisms. As [7] shows for example, the SCT generator model can be used for applications other than web applications and the presentation of SCT generators are rather abstract and showing only vital technical ideas. For example, authentication, session handling and interaction with database and web servers seem to be out of the authors' sight.

In [8] the authors proposed a code generating machinery which maps a declarative specification language onto an imperative target language. As a specification language the authors use SDL (Specification and Description Language) whose description of specification is very comprehensive [9]. In [9], this method and SCT are compared to see when one is a better choice than the other. In [9] the author also pointed out that specification avoids human errors in programming and also is an ideal starting point for automatic code generation.

These works referred so far are conceptual in nature and don't provide with or even mention much about implementation details. For example, although some of these works use templates as our web code generator does, but the necessary details are not presented in their works to see the differences between their templates and ours. Our web code generator which will be described in the following sections is (in contrast to the previous works) web specific and constructs the directories and the static libraries and modules and routing files, all you need to run a web application.

III. A Scheme of Generating Web Code

As mentioned our code generation system sets everything you need to run a web application. So, to organize our presentation, we separate the system into the three parts: the system setting, the application setting, and the routing setting. All the three parts of the settings are done based on configurations written in JSON.

We prepare the directory for our code generation system, say 'User_Home/Gen' and call it 'Gen_home'. Every process of our system will be executed from on this directory.

For showing concrete examples of generating web applications we need concrete scripts for the generating processes and the generated code. In our case we write them in Perl though a basic idea of our code generation system can be applied to other languages as well.

A. System Setting

In this part we set the basic environments for web applications we will produce. First we determine the name of the environment as we wish. So let's call it 'pp' and web applications constructed under it 'pp applications'. Then we write a JSON configuration file for pp, say 'Gen_home/CONFpp/conf_pp_data.json' and the file content is:

```
{
    "ppData":
    {
        "DbModel": "mymysql",
        "FullWebHomePath": "/var/www/html",
        "WebReadPath": "pp_d",
        "JsPath": "pp_d/js",
        "CssPath": "pp_d/css",
        "ImgPath": "pp_d/img",
        "WebExecPath": "pp",
        "WebLibPath": "pp/mylib",
        "DbUser": "kajitori",
        "WebServerUser": "apache",
        "ppOwner": "kajitori"
    }
}
```

This expresses an object in JSON which determines the database engine model be MySQL (or MariaDB, 'mymysql' is the name of the module manipulating MySQL in our language) and the web read-path be '/var/www/html/pp_d' and the web execution-path be '/var/www/html/pp' and so on. We chose JSON as our configuration language throughout our code generating system because its notation is simple [12].

The data model of user accounts and session data are common to every pp application. If as above our database engine is MySQL (MariaDB) then we write an SQL file 'Gen_home/CONFpp/pp.sql' like:

```
DROP TABLE IF EXISTS `account`;
CREATE TABLE `account` (
  `account` varchar(10) NOT NULL,
  `password` varchar(10) NOT NULL,
  PRIMARY KEY (`account`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `session`;
CREATE TABLE `session` (
  `session_id` varchar(40) NOT NULL DEFAULT '',
  `account` varchar(10) DEFAULT NULL,
  `login_datetime` datetime DEFAULT NULL,
  PRIMARY KEY (`session_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `session_data`;
CREATE TABLE `session_data` (
  `session_id` varchar(40) NOT NULL DEFAULT '',
  `name` varchar(20) DEFAULT NULL,
  `value` text,
  `dumped` tinyint(1) DEFAULT '0',
  `updated_time` datetime,
  KEY `session_id` (`session_id`),
  FOREIGN KEY (`session_id`) REFERENCES
 `session`(`session_id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `pp_data`;
CREATE TABLE `pp_data` (
  `name` varchar(20) NOT NULL,
  `value` text character set utf8,
  PRIMARY KEY (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The table 'account' and 'session' store account info and session id, respectively. The table 'session_data' stores data used throughout the session. The table 'pp_data' stores data specified in the file 'conf_pp_data.json'.

To register pp applications we also write another SQL file 'Gen_home/CONFpp/app.sql' :

```
DROP TABLE IF EXISTS `app`;
CREATE TABLE `app` (
  `name` varchar(20) DEFAULT NULL,
  `code` int(11) NOT NULL AUTO_INCREMENT,
  `register_datetime` datetime,
  PRIMARY KEY (`code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `app_data`;
CREATE TABLE `app_data` (
  `app_code` int(11) NOT NULL DEFAULT '0',
  `name` varchar(20) DEFAULT NULL,
  `value` text character set utf8,
  `dumped` tinyint(1) DEFAULT '0',
  PRIMARY KEY (`app_code`,`name`),
  KEY (`name`),
 FOREIGN KEY (`app_code`) REFERENCES `app`(`code`)
ON UPDATE CASCADE ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `role`;
CREATE TABLE `role` (
  `account` varchar(10) NOT NULL,
  `app_code` int(11) NOT NULL DEFAULT '0',
  `uadmin` tinyint(1) NOT NULL DEFAULT '0',
  `admin` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`account`,`app_code`),
 FOREIGN KEY (`account`) REFERENCES
`account`(`account`) ON UPDATE CASCADE ON DELETE
CASCADE,
 FOREIGN KEY (`app_code`) REFERENCES `app`(`code`)
ON UPDATE CASCADE ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `user_data`;
CREATE TABLE `user_data` (
  `account` varchar(10) NOT NULL,
  `app_code` int(11) NOT NULL DEFAULT '0',
  `name` varchar(20) NOT NULL,
  `value` text character set utf8,
  `dumped` tinyint(1) DEFAULT '0',
  PRIMARY KEY (`account`,`app_code`,`name`),
 FOREIGN KEY (`account`) REFERENCES
`account`(`account`) ON UPDATE CASCADE ON DELETE
CASCADE,
 FOREIGN KEY (`app_code`) REFERENCES `app`(`code`)
ON UPDATE CASCADE ON DELETE CASCADE,
 FOREIGN KEY (`name`) REFERENCES
`app_data`(`name`) ON UPDATE CASCADE ON DELETE
CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

These tables are created here because they have to exist before generating a pp application and we postpone explanation of them until the next subsection.

Then we run a Perl script 'generate_pp.pl' on Gen_home to do the following:

(1) to create a database instance 'pp' in MySQL and execute 'pp.sql' to create the tables above and register 'DbUser' as a superuser for 'pp' and grant update, insert, delete to 'WebServerUser' and store the data for pp in 'pp_data'.

(2) to make the directories under the web home and copy the static libraries and javascript files and css files and image files which are common to every pp application to the directories 'the_web_home/pp/mylib', 'the_web_home/pp_d/js', 'the_web_home/pp_d/css', 'the_web_home/pp_d/img', respectively.

(3) to just execute 'app.sql'.The static libraries under the directory 'mylib' include mysession.pm which deals with session control and myview.pm which gives basic HTML printing methods and mymodel/mymysql.pm which gives basic access to MySQL (or MariaDB). These libraries are basic to pp applications as we explain later.

B. Application Setting

In this part we set up the environment of an individual pp application. Let the name of a pp application we are going to configure be 'test'. Then first we write a JSON configuration file Gen_home/CONFapp/conf_test_data.json like:

```
{
  "AppData":
  {
    "AppName": "test",
    "AppOwner": "kajitori",
    "MenuItem": "logout,config",
    "AppModel":
    {
      "ModelName": "requested books",
      "ModelType": "mymysql"
    }
  }
}
```

This is an example of a configuration of a simple pp application 'test' which accepts and stores a request for books to be bought by a library and will be explained further and demonstrated in the next section. For 'test' we also need a model definition to be executed by a database engine of “ModelType”. The model type for 'test' is configured above to be 'mymysql' and so we use 'mymysql.pm' as the module for database tasks of 'test' and we write a model definition in the language of MySQL (MariaDB) :

```
DROP TABLE IF EXISTS `test_request`;
CREATE TABLE test_request (
  `id` varchar(50) NOT NULL PRIMARY KEY,
  `book_info` varchar(60) character set utf8,
  `requester` varchar(150) character set utf8,
  `department`  varchar(30)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

At this point, we run the script 'generate_app.pl' by:

```
$ ./generate_app.pl  test
```

To do the following:

(1) to create the directory for the application 'the_web_home/pp/test'.

(2) to register the application to the table 'app' and set the application data in the above configuration file to the table 'app_data' in the database 'pp'.

(3) according to MVC (Model, View, Control) model for applications, to generate from templates 'model_template', 'view_template', 'control_template' under 'Gen_home/CONFpp/mylib/TEMPLATE' the modules 'test_model.pm', 'test_view.pm' and 'test_control.pm' under 'the_web_home/pp/test'.

(4) to just execute 'test.sql'. The module 'test_model.pm' defines CRUD routines for the model of the application which are higher level routines than those of the basic modules under 'the_web_home/mylib/mymodel' (of course 'test_model.pm' uses those basic model modules). The module 'test_view.pm' defines HTML writing routines which again use and are higher in level than those in myview.pm in 'the_web_home/pp2/mylib'. The module test_control.pm defines (by our definition of 'control' for our system) wrapper routines which utilize routines whose code cannot be generated by our system e.g. routines for reading marksheets (see section V).

C. Routing Setting

For pp applications the routing is done by a web script file (in our case a Perl program file) which we call a uibase (user interface base) file because it is organized by managing user requests and responses to users.

The function of a uibase can be expressed by an action diagram of UML as in Fig.1 below.

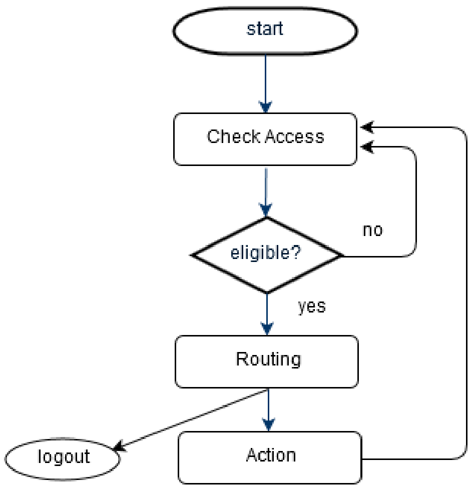


Fig.1. Action diagram for a uibase

At each time a uibase gets a request from the user, it checks the user's eligibility for the request at the 'Check

Access' node in the diagram. This check process is done by creating an object of 'mysession.pm' package in 'mylib'. The object sets in itself many parameters in 'app_data' and 'session_data' to fit the user's environment. If the check is ok, then the uibase routes the request to some actions of our library and show the next user interface as a response.

The skeleton of a uibase file is as follows (we only write comments) :

```
#!/usr/bin/perl
# Set a session object
# Check access
# Routing
```

A uibase file is automatically generated from a template 'uibase_template' under the 'TEMPLATE' directory which writes the above skeleton in Perl without the routing part and a uibase configuration file 'conf_test_uibase.json' under 'Gen_home/CONFapp':

```
{
  "UiBase": [
  {
    "UiBaseName": "user",
    "MenuItem": "logout, ...",
    "Role": "user",
    "Route": [
      {
        "RoutePath":"print_page_show_record",
        "Action": [
          {
            "ActionName":"test_model::search_table(...)",
            "ActionOutput": "recs",
            "Target": "print_page_show_record"
          },
          {
            "ActionName":"print_page_show_record(...)"
          }
        ]
      },
      {
        "RoutePath": "",
        "Action": [
          {
            "ActionName":"print_page_add_record(...)"
          }
        ]
      },
      {
        "RoutePath": "add_record",
        "Action":[
          {
            "ActionName":"test_model::add_record"
          },
          {
            "ActionName":"print_page_message(...)"
          }
        ]
      ]
    }
  ]
}
```

In the above, '…' part means the real content is omitted for brevity (the same in the followings).

The above expression is a JSON object for which the only attribute is "Uibase" whose value is an array of uibases but usually it includes only one uibase. The general form of the object is:

```
{
  "Uibase": [
    {
      "UiBaseName": "…",
      "MenuItem": "…",
      "Role": "…",
      "Route": [
        ...
      ]
    },
    {
      ...
    }
  ]
}
```

The value of "UiBaseName" will be the name of the uibase file which will be generated based on this configuration.

"MenuItem" determines the menu items to be shown on every page of the uibase. "Role" means the role the user must be admitted in the table 'role'. The 'role' table determines whether it admits each of the three roles 'user', 'uadmin' and 'admin' to each (account, application) pair.

"Route" is an array of routes each of which is an object of the form:

```
{
  "RoutePath": "….",
  "Action": [
    {…},
    {…}
  ]
}
```

If the request URL has a PATH_INFO which matches the value of "RoutePath" of a route, then the actions of "Action" of the route are invoked in the order in the array. For 'conf_test_uibase.json', for example, if the request URL is:

http://localhost/pp/test/user/print_page_show_record

, then the uibase 'user' prints an HTML page displaying all the records as explained in detail below.
if "ActionName" begins with "print_page", then the action object is of the form:

```
{
  "ActionName": "print_page_...",
  "PageOutput":
  {
    …
  }
}
```

or otherwise

```
{
  "ActionName": "...",
  "ActionOutput": "...",
  "Target": "..."
}
```

In the former case the action is printing an HTML page and "PageOutput" is the content of the page. In the above example configuration there is no "PageOutput" because all "print_page_..." methods in the above example are custom methods defined in the module 'test_view.pm' which are just inherited from the template 'view_template' under the template directory 'Gen_home/CONFpp/mylib/TEMPLATE'. The more we use custom methods the less configuration we must write and the more we can benefited by the code generation.

Examples of "PageOutput" will appear in section V.

In the latter case, "ActionOutput" is optional because the method may not have an output. "Target" means the name of the method which uses the output named by "ActionOutput". In the above example, the method 'test_model::search_table' targets and pass the output records "recs" to the method 'print_page_show_record' which of course amounts to showing the search result. The prefix 'test_model::' indicates (in Perl grammar) that the method is defined in the 'test_model.pm' module under 'the_web_home/pp/test'.

There may be more than one uibase configuration which in turn can be separated into more than one file.

Then we run the script 'augment_uibase.pl' by:

```
$ ./augment_uibase.pl  test
```

To do the following:

(1) to join the uibase configurations altogether to make an array of uibases and write it into the file 'conf_augmented_test_uibase.json'.

Actually there is another important role of the script 'augment_uibase.pl', namely to 'augment' routes for update, delete, download, but since the 'test' application does not use this augmentation, our explanation of this role will be postponed until the section V.

Now that we have all the uibase configurations in a file, we generate the all uibase files by executing:

```
$ ./generate_uibase.pl  test
```

This command actually does for each uibase cofiguration the following:

(1) to create the uibase file of the name determined by the "UiBaseName" under 'the_web_home/pp/test'.

(2) to open 'uibase_template' under 'Gen_home/CONFpp/mylib/TEMPLATE'. Replace in the template the temporary expressions "_UiBaseName", "_Role", "_AppName", "_DbUser", "_MenuData" by their real values for the uibase.

(3) to replace in the template the expression "_ROUTING" by the real routing code for the uibase.

(4) to write the content of the template into the uibase file.

In (3) the real routing code for the uibase is generated for each “Action” array of the uibase as follows:

(a) the routing of the action array is of the form:

```
elsif($path_info eq '/_route_path_'){
        ...
        ...
}
```

, where '_route_path_' is the value of "RoutePath" for which the action array is defined and the '...' parts will be fulfilled with the expressions for the actions in the action array.

(b) if the action is not a 'print_page_...', then the code for the action is of the form:

```
my $_output_=_class_::_action_;
```

, where 'my $_output_=' is omitted if the action has no "ActionOutput", otherwise the part '_output_' is the value of "ActionOutput". The part '_action_' is the value of "ActionName". If the value of "ActionName" does not include '::', then '_class_' will be 'test_control' and if "ActionName" already include '::', namely the class is already determined, then '_class_::' is omitted and we assume that the action is already defined in the module of the class (we call them 'custom methods'). The part '_action_' may include the argument part '(...)' at the end and in that case the arguments should be appropriately added '$' (if we use Perl).

(c) if the action is a 'print_page_...', then the code for the action is of the form:

```
test_view::print_page_...;
```

, where the part '...' includes not only the name of the action but also the argument part '(...)' which contains the argument determined from the value of "PageOutput". The value of "PageOutput" which is a JSON expression is transformed into the corresponding structure of Perl and written as an argument of the action. If "PageOutput" is not defined, then we assume that the action is already defined in the module 'test_view.pm', namely the action is a custom method of the module 'test_view.pm'. An example of the value of "PageOutput" and its transformation into a Perl structure will be shown in section V.

IV. The TEST Example

In the previous section, the configuration files are written for a pp application 'test'. In this section, we show how the application 'test' works and how its code can be effectively generated by our system.

The aim of 'test' is simple:

(1) to register a request for a book to be purchased by the library.

(2) to show all the requests so far registered.

We demonstrate 'test' by showing a login view in Fig.2 and a view for (1) in Fig.3 and a view for (2) in Fig.4 (below WebExecPath is 'pp2' not 'pp').

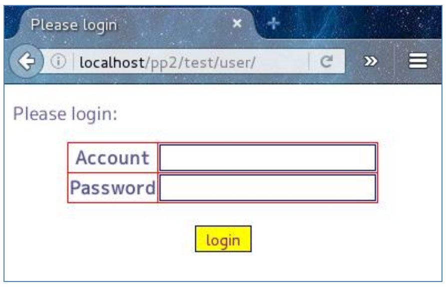


Fig.2. A login view

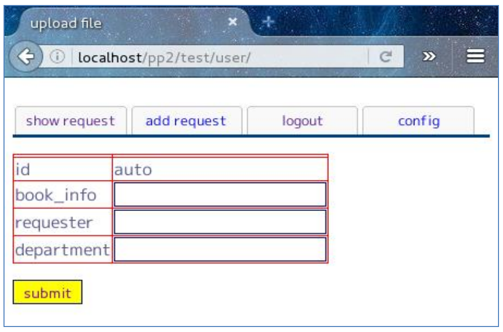
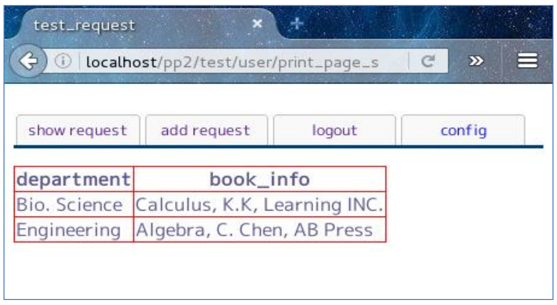


Fig.3. A view for registering a request



department	book_info
Bio. Science	Calculus, K.K, Learning INC.
Engineering	Algebra, C. Chen, AB Press

Fig.4. A view for showing all the requests

To accomplish the aim we needed just the following configuration files as we described in the previous section:

Table 1. The configuration files for 'test'

File name	# of characters	# of words
test.sql	255	31
conf_test.data.json	144	13
conf_test_uibase.json	670	52
Total	1069	96

On the other hand the following files are automatically generated for 'test' under 'the_web_home/pp2/test'.

Table 2. The generated files for 'test'

File name	# of characters	# of words
test_model.pm	3581	398
test_view.pm	3867	523
user	1998	174
Total	9446	1095

The number of characters is counted by using LibreOffice Writer (which counts space). The number of words is counted by LibreOffice Writer except that we count it by vision for JSON files. The difference between the size of the configuration files and the size of the generated files is huge (the ratio is about 1:10) because in 'test' the methods are custom methods which are prepared in the templates.

In the above figures, HTML styles are controlled by the default CSS file prepared for pp application (see the next section for details).

V. An Example using An External Library

In this section we present an example of a pp application that uses an external library for which our code generating system cannot generate code from configurations.

A. *the aim of the application*

We name this pp application (as in a configuration file below) 'reo' because it returns scanned exam answer sheets already graded to the students (**r**eturn **e**xam **o**nline). The external library we use for reo does the job of reading the student ids marked on marksheets on students' answer sheets. Note that although the external library reads ids correctly almost all the time from marksheets, we need to check its correctness anyway. To use 'reo' the teacher should prepare in advance a PDF file (or a zipped file) of the scanned answer sheets on each of which a marksheet is printed on one side for marking the student id.

The 'reo' aims at the following functions:

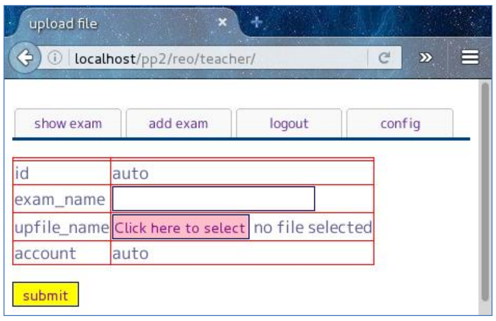
(1) A teacher registers an exam file stated above.

(2) After the exam file is uploaded, it is separated into pages and marksheets on all the pages are read (so the ids are 'identified'). Then, the images of the marksheets and the corresponding student ids are returned to be checked by the teacher.

(3) After the teacher checks the correspondence and corrects it if necessary, he or she submits the check result. Then, the application creates for each student id a PDF file consisting of the student's answer sheets.

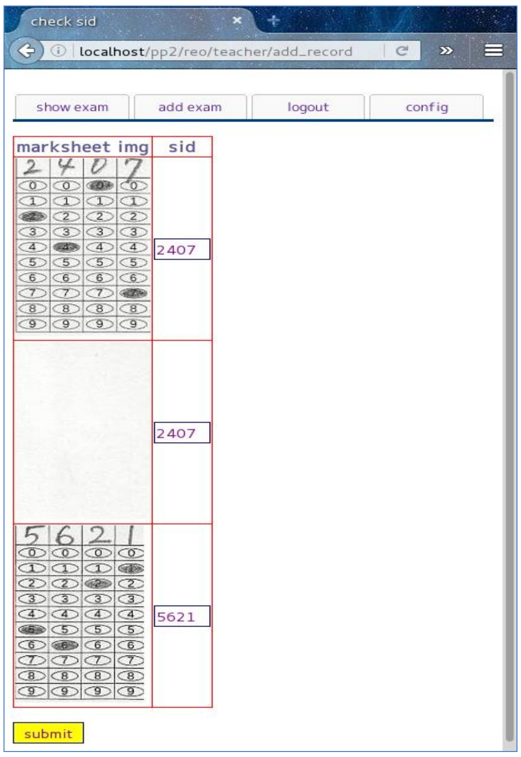
(4) A student upon login can see or download a PDF file of exam answer sheets he or she wrote.

We demonstrate views for (1) to (4) below in Fig.5 to Fig.8.



id	auto
exam_name	
upfile_name	Click here to select no file selected
account	auto

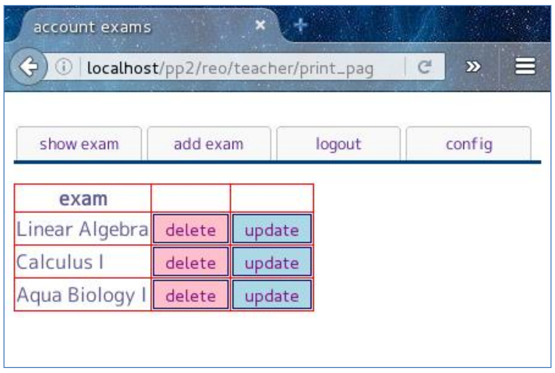
Fig.5. A teacher's view for adding an exam



marksheet img	sid
	2407
	2407
	5621

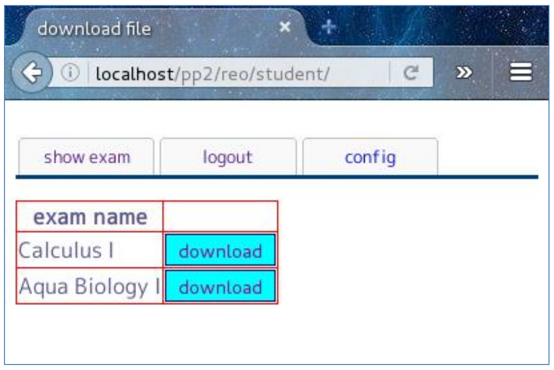
Fig.6. A teacher's view for checking the student ids.

The empty image means that it is a back of a sheet and no marksheet on it, hence the same sid (student id) is assigned as the previous page.



exam		
Linear Algebra	delete	update
Calculus I	delete	update
Aqua Biology I	delete	update

Fig.7. A teacher's view for all exams registered by the teacher.



exam name	
Calculus I	download
Aqua Biology I	download

Fig.8. A student's view for downloading

B. *Role of Control Module*

Unlike the 'test' application in the previous section, the pp application 'reo' has a control module 'reo_control.pm' (C in MVC model) other than 'reo_model.pm' and 'reo_view.pm' (M and V in MVC model). For pp applications, a control module acts as a wrapper to an external library and we do not provide with a way to generate it so we have to write it manually. We have a static library for manipulating marksheet named 'marksheet.pm' under 'mylib'. This module does the following:

(1) to separate a file into the pages (image files) and read a sequence of numbers from a marksheet on each page. The method name is '_read_marksheet'.
(2) to create a PDF file consisting of the pages of the same sequence. The method name is '_create_sid_pdf'.

Then for example, in 'reo_control.pm', a method 'read_marksheet' is defined as:

```
sub read_marksheet {
    my $ms=shift;
    …
    my $corres = marksheet::_read_marksheet(
        $ms->{session_data}->{upfile_name},
        ...
    );
    return $corres;
}
```

If the external library is written so as to be used directly in uibases, then we can dispense with a control module. Actually 'marksheet.pm' can be used directly in uibases and we can dispense with 'reo_control.pm', but we wrote it anyway because otherwise the configuration of uibases would be complicated.

C. *Configuration files for REO*

As in the case of 'test' application, we write 'conf_reo_data.json' for basic settings like "AppName" and "AppModel", and so on. But this time the file 'conf_reo_data.json' includes an attribute "Augment" of the object "AppModel" as below:

```
"AppModel":
{
  …,
  "Augment":
  [
    {
      "UiBaseName": "teacher",
      "AugmentItem":
      {
        "Update": "",
        "Delete": ""
      }
    },
    {
      "UiBaseName": "student",
      "AugmentItem":
      {
        "Download": ""
      }
    }
  ]
}
```

The 'Augment' configuration and HTML custom types defined in 'myview.pm' dispense us from very tedious coding of user interfaces for update, delete for records of database tables, and download of files. In the above configuration it is designated, for example, that the uibase 'teacher' should include the routing code for updating table, namely showing an HTML table for updating a record and committing it to the database and reporting the result to the user. To include these routing code, 'augment_uibase.pl' mentioned in section III adds JSON code so that 'generate_uibase.pl' should generate these routing code. In 'reo' application these routing are invoked by custom HTML types 'updateByIdButton', 'deleteByIdButton' and 'downloadByIdButton' which are explained further when we describe the configuration of uibase 'teacher'.

We assume that data structures of pp applications are not complicated. So, we had better model it by directly writing SQL script files. Here is our definition of the model for 'reo', 'reo.sql':

```
DROP TABLE IF EXISTS `reo_exam`;
```

```
    CREATE TABLE reo_exam (
      `id`  varchar(50) NOT NULL PRIMARY KEY,
      `exam_name`  varchar(60)  character set utf8,
      `upfile_name` varchar(150)  character set utf8,
      `account` varchar(10)
    ) ENGINE=InnoDB DEFAULT CHARSET=latin1;

    DROP TABLE IF EXISTS `reo_student_doc`;
    CREATE TABLE `reo_student_doc` (
      `id`  varchar(50) NOT NULL PRIMARY KEY,
      `sid` varchar(10) NOT NULL,
      `exam_id` varchar(50),
      `file_name` varchar(150) NOT NULL,
      UNIQUE(`sid`,`exam_id`),
      KEY `exam_id` (`exam_id`),
      FOREIGN      KEY      (`exam_id`)      REFERENCES
`reo_exam`(`id`) ON DELETE CASCADE
    ) ENGINE=InnoDB DEFAULT CHARSET=latin1;

    CREATE VIEW `reo_student_view` AS
      SELECT      reo_student_doc.id,      reo_student_doc.sid,
reo_exam.exam_name, reo_student_doc.file_name
      FROM reo_exam, reo_student_doc
      WHERE reo_exam.id=reo_student_doc.exam_id;
```

This defines indeed a simple data model with just two tables and one view. The table 'reo_exam' stores uploaded file information for each teacher account and the table 'reo_student_doc' stores PDF file information for each student id. The view 'reo_student_view' is set for viewing PDF information with exam names not with ids.

We would like to generate two uibase files, 'teacher' for teachers and 'student' for students. For teachers we write a configuration 'conf_reo_uibase.json' under 'Gen_home/CONFapp':

```
    {
    UiBase":[
      {
        "UiBaseName": "teacher",
        "MenuItem": "...",
        "Role": "uadmin",
        "Route": [
          {
            "RoutePath": "",
            "Action": [
              {
                "ActionName":"print_page_add_record(…)
              }
            ]
          },
          {
            "RoutePath": "add_record",
            "Action": [
              {
                "ActionName": "reo_model::add_record"
              },
              {
                "ActionName": "read_marksheet",
                "ActionOutput": "img_page_sid_taio",
                "Target": "print_page_check_sid"
              },
              {
                "ActionName": "print_page_check_sid",
                "PageOutput":
                {
                  "PageTitle": "check sid",
                  "Content": [
                  {
                    "ContentType": "form",
                    "FormAction":  "create_sid_pdf",
                    "Content": [
                      {
                        “ContentType": "table",
                        “Column": [
                        {
                          "ColumnTitle": "marksheet img",
                          "Content": [
                            {
                              "ContentType": "image",
                              "ImageUrl": 0
                            }
                          ]
                        },
                        {
                          "ColumnTitle": "sid",
                          "Content": [
                            {
                              "ContentType": "input",
                              "InputType":"text",
                              "InputName": 1,
                              "InputValue": 2
                            }
                          ]
                        }
                      ]
                    }
                  ]
                }
                ]
                }
              }
            ]
          },
          {
            "RoutePath": "create_sid_pdf",
            "Action": [
              {
                "ActionName": "create_sid_pdf"
              },
              {
                "ActionName": "print_page_message(…)"
              }
            ]
          },
          {
            "RoutePath": "print_page_account_exams",
            "Action": [
              {
"ActionName":"reo_model::search_table_ByAccount(...)",
                "ActionOutput": "exam_events",
                "Target": "print_page_exam_events"
              },
              {
                "ActionName": "print_page_exam_events",
                "PageOutput":
                {
                  "PageTitle": "account exams",
                  "Content": [
                  {
                    "ContentType": "table",
                    "Column": [
                    {
                      "ColumnTitle": "exam",
                      "Content": [
                        {
                          "ContentType": "text",
                          "TextType": "raw",
                          "Text": 1
                        }
                      ]
                    },
                    {
                      "ColumnTitle": "",
                      "Content": [
                        {
                          "ContentType": "deleteByIdButton",
                          "Table": "reo_exam",
                          "Id": 0
                        }
                      ]
                    },
                    {
                      "ColumnTitle": "",
                      Content": [
                        {
                          "ContentType": "updateByIdButton",
                          "Table": "reo_exam",
                          "Id": 0
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

This JSON expression consists mostly of an array "Route" of routes and each route contains an array "Action" of actions just as in 'conf_test_uibase.json'. This time each action is not necessarily a custom method already defined in a module. We colored the parentheses and the value of "RoutePath" of each route by red and boldfaced them for visibility.

In the above configuration, there are two 'print_page_...' actions which have "PageOutput" definitions (those page outputs are marked green). If a 'print_page_...' action is not already defined in the template 'view_template' from which 'reo_view.pm' is generated, then we have to define it by writing a page output. The value of a "PageOutput" is a JSON object whose "Content" array should be transformed into an HTML page content by a method in the module 'reo_view.pm' for which we have to generate code. The code we generate for a value of "PageOutput" is just a wrapper of the method 'print_ page' in 'myview.pm'. Thus the code is of the form :

```
sub print_page_… {
  my $ms=shift;
  my $menu=shift;
  my $option=shift;

  myview::print_page ( $ms,$menu,
    {
      'PageTitle' => '...',
      'Content' => [
        ...
      ]
    },
    $option
  );
}
```

Here, the Perl object

```
    {
      'PageTitle' => '...',
      'Content' => [
        ...
      ]
    }
```

is just a direct translation of the JSON object, i.e. the value of the "PageOutput" into a Perl object.

The '$option' is an object of options which may contain '$option→{Data}' which sends dynamic data for HTML code. In the above configuration, the first page output includes an HTML content like :

```
{
    "ContentType": "image",
    "ImageUrl": 0
}
```

which shows this content is '<img src="...">' type. Since in this case '$option→{Data]' is an array reference, the 'src' attribute of the 'img' tag has the value :

```
$option→{Data}→[0].
```

The '$option→{Data}' could be a Perl object of the form :

```
{ ""=>…, "..."=>…, ... }
```

in which case the above 'src' attribute has the value :

```
$option→{Data}→{""}→[0]
```

, and other assignments are reserved for other kinds of data.

Options are passed in a variable '$option' whose code is written in a uibase file as:

```
elsif($path_info eq '/add_record'){
  reo_model::add_record($ms);
  my $corres=reo_control::read_marksheet($ms);
  reo_view::print_page_check_sid (
    $ms,
    \@menu_data,
    {css_path => ['/pp_d/css/reo.css'], Data => $corres}
  );
}
```

In this routing in the uibase file 'teacher' for the 'reo' application the red highlighted part is the option. This option includes two option data. One determines HTML style of the page. The other determines the dynamic data in the page as follows. The 'targeted' action 'print_page_check_sid' accepts as an option the output '$corres' of the action 'read_marksheet' which targets 'print_page_check_sid'.

The second page output describes the content of the HTML table of the registered exams whose columns include the columns for the two HTML buttons 'updateByIdButton' and 'deleteByIdButton' (see Fig.7). These buttons are defined as custom HTML types in 'myview.pm' just as those submit buttons for HTML forms for updating or deleting a record by id.

We need for 'reo' another uibase configuration for students which we named 'conf_reo_uibase2.json', but we do not present it here because no new features are in it except an HTML custom type 'downloadByIdButton' which is similar to 'updataByIdButton' and 'deleteByIdButton' described above.

Next as the previous section we compare the size of the configuration files and the size of files under 'the_web_home/pp':

Table 3. The manually written files for 'reo'

File name	# of characters	# of words
reo.sql	803	85
conf_reo.data.json	546	54
conf_reo_uibase.json	2930	207
conf_reo_uibase2.json	1281	94
reo_control.pm	1101	67
Configuration Total	**6561**	**507**
marksheet.pm	6744	565
Total	**13305**	**1072**

Table 4. The files under 'the_web_home/pp' used by 'reo'.

File name	# of characters	# of words
reo_model.pm	3306	345
reo_view.pm	8697	643
reo_control.pm	1101	67
teacher	3042	255
student	1710	154
Produced Total	**17856**	**1464**
marksheet.pm	6744	565
'reo' files Total	**24600**	**2029**
mysession.pm	12025	1130
myview.pm	9012	745
mymysql.pm	3961	359
myfile.pm	2700	302
myroutine.pm	4395	377
MylibTotal	**32093**	**2913**
Total	**75750**	**6543**

'Produced Total' means the total of generated files and the manually written control file for 'reo'.
''reo' files Total' means the total of the files specific to the application 'reo'.

If we assume that the 'marksheet.pm' is provided as a static library before coding the 'reo' application, then the manually prepared files for 'reo' amount in size to about one third of files generated or used as is (represented altogether as 'produced' in the table) in the directory 'the_web_home/pp/reo', which is compared to the result for 'test' where the configuration files amount to one tenth of the generated files.

If we assume that we have to code marksheet.pm when we implement the 'reo' application, then the ratio decreases to about a half. But still the libraries doing authentication, session handling, file uploading & downloading, and custom HTML types and custom methods help implementing the application avoid temporal coding of basically important functions.

VI. Conclusion and Final Remarks

Our first example 'test' shows that our code generation system is quite effective if the application needs just 'routines' which our system prepares as custom methods or types. The second example 'reo' shows that our pp libraries and custom types and methods can still be useful in that it helps us concentrate on coding functions which is not generic (in the sense of our system).

To edit configuration files in JSON format, we had better use a JSON processor like jq [11] because such a JSON processor can check the syntactical correctness of JSON expressions and pretty-prints them which makes our system much easier to use. We write the contents of HTML page outputs in JSON not in HTML directly because JSON is easier to input with the assistance of those JSON processors. But it seems not hard to modify our system to be able to write HTML page outputs in HTML as well as in JSON.

The more custom methods and custom types are defined, the more useful our system will become as a way of generating web code and it will be so if we go on writing web applications by using our system because we would get more reusable code for our system.

Our method of generating web code described in this paper can be seen as an automation of the way of coding web applications which the author has used in recent years. The author wanted to treat the method in a more systematic way. For example, it is certainly desirable to have a systematic way of handling errors which occur during the code generation with suggesting ways to correct them and achieve what we want. We have written some error checking code in our code generating scripts but they are not enough because they are not exhaustive nor sufficiently suggestive at all. Ontological or semantic description of web applications have been studied [13][14]. We expect that ontological or semantic approaches would help make our automation of coding more systematic. This will be our future investigation.

References

[1] K.Kajitori, K. Aoki, S. Ito, Developing a Compact and Practical Online Quiz System, International Journal of Modern Education and Computer Science(IJMECS), Vol.6, No.9, 1-7 (2014).

[2] K.Kajitori, K. Aoki, Implementation of a Simple Document Repository System, International Journal of

Modern Education and Computer Science(IJMECS), Vol.8, No.9, 12-19 (2016).
[3] Schwabe, D., Rossi, G., Esmeraldo, L., Lyardet, F., Web design frameworks: An approach to improve reuse in web applications, Web Engineering pp.335-352 (2001).
[4] Nitish Pathak, Girish Sharma, B.M.Singh, Experimental Analysis of SPF Based Secure Web Application, International Journal of Modern Education and Computer Science(IJMECS), Vol.7, No.2, 48-55 (2015).
[5] Radošević, Danijel, and Ivan Magdalenić, Source code generator based on dynamic frames. Journal of Information and Organizational Sciences, 2011.
[6] Radošević, Danijel, and Ivan Magdalenić. "Python implementation of source code generator based on dynamic frames." MIPRO, 2011 Proceedings of the 34th International Convention. IEEE, 2011.
[7] Kvesić, A., Radošević, D., Orehovački, T.:"Dynamic Frames Based Generation of 3D Scenes and Applications", Acta Graphica, ISSN: 0353-4707, 26(1-2), 11-19, 2015.
[8] Mansurov, N. and Ragozin, A. Using declarative mappings for automatic code generation from {SDL} and asn.1. In Lahav, R. D. v. B., editor, {SDL} '99, pages 275 – 290. Elsevier Science B.V., Amsterdam, 1999.
[9] Martin Kaufleitner, Code Generation from Configuration Specification Languages -For Program Execution Environment Configuration-, Seminar aus Programmiersprachen, May 6, 2016.
[10] JSON, http://www.json.org
[11] jq, https://stedolan.github.io/jq/.
[12] Mohammed Ali, Tarek S. Sobh,*, Salwa El-Gamal, Identity Management: Lightweight SAML for Less Processing Power, International Journal of Information Technology and Computer Science(IJITCS), Vol.7, No.4, pp.42-49, 2015
[13] Amira Abdelatey, Mohamed Elkawkagy, Ashraf Elsisi, Arabi Keshk, "Improving Matching Web Service Security Policy Based on Semantics ", International Journal of Information Technology and Computer Science(IJITCS), Vol.8, No.12, pp.67-74, 2016.
[14] N. Kaur, H. Aggarwal,"Evaluation of Information Retrieval Based Ontology Development Editors for Semantic Web", International Journal of Modern Education and Computer Science (IJMECS), Vol.9, No.7, pp.63-73, 2017.

Authors' Profiles

Kazuaki Kajitori, Ph.D, is a professor of the Department of Fisheries Distribution and Management at National Fisheries University in Japan. In teaching, he has been in charge of classes of mathematics and statistics and computer sciences. In his classes, he has been utilizing IT methods extensively. He wrote online texts and courses' home pages and conducted many online exams and let students do online exercises as the preparation of online exams. In research, he has studied mathematical logic which led him computer related fields like data mining and databases and e-learning. He has developed several web applications including one treated in this paper.

How to cite this paper: Kazuaki Kajitori, "Generating Code for Simple Dynamic Web Applications via Routing Configurations", International Journal of Modern Education and Computer Science(IJMECS), Vol.9, No.11, pp. 1-12, 2017.DOI: 10.5815/ijmecs.2017.11.01