

ErlangWeb Tutorial

Erlang Training & Consulting Ltd.

December 15, 2008

www.erlang-web.org

Abstract

Erlang Web 1.2 release tutorial.



Contents

1 Erlang Inside	3
1.1 Overview	3
1.2 Easy start	3
1.3 Compiling the project	4
1.4 Adding new application	5
1.5 Running the server	5
2 Directory Structure	7
2.1 Overview	7
2.2 Directory Meaning	7
2.3 Example of a directory tree	9
3 MVC - What is What	10
4 Philosophy	11
4.1 Architecture of Types	11
4.1.1 Overview	11
4.1.2 Origin of a problem	11
4.1.3 User friendly complex types	11
4.1.4 Deeper	12
4.2 Dispatcher	15
4.2.1 Overview	15
4.2.2 Types of dispatching	15
4.2.3 Control flow	15
4.2.4 dispatch.conf	16
4.2.5 Named subpatterns	17
4.2.6 Skipping dispatcher	18
4.2.7 Example	18
4.3 Validation	20
4.3.1 Why do we need validation	20
4.3.2 Expectations	20
4.3.3 Validate_tool	23
4.4 Internationalization	24
4.4.1 Overview	24
4.4.2 Defining translations	24
4.4.3 Translating	24
4.4.4 Control flow	25
4.4.5 Example	27
4.5 Request Dictionary	28
4.5.1 Overview	28

4.5.2	Request dictionary API	28
4.5.3	Special cases	30
4.6	DBMS	31
4.6.1	Overview	31
4.6.2	e_db module	31
4.6.3	Supported DBMS	32
4.6.4	Example	32
4.7	Project configuration file	33
4.7.1	Overview	33
4.7.2	Types of options	33
4.7.3	Example	35
4.8	Data flow	37
4.8.1	Overview	37
4.8.2	Control flow	37
4.8.3	Example	38
4.9	Template engine	41
4.9.1	Overview	41
4.9.2	wtpl tags	41
4.9.3	Control flow	41
4.9.4	Example	42
4.9.5	wpart_gen	43
5	Controller. One to rule them all.	46
5.1	Overview	46
5.2	The simplest case	46
5.3	Second step: data flow and new home made controller	48
5.4	Pure dynamic	50
6	Model. What happens in model it stays in model.	52
6.1	Overview	52
6.2	Specification	52
6.3	Example	52
6.3.1	Basics	52
6.3.2	Dynamic type changes	53
7	View. Let's see it.	55
7.1	Overview	55
7.2	Example	55
7.2.1	Base and dispatcher	55
7.2.2	Content	56
7.2.3	Listing	57

8	Reference Manual	60
8.1	Basic Types	60
8.1.1	Definition options	60
8.1.2	HTML tags	66
8.2	Interesting wparts	67
8.2.1	wpart_choose	67
8.2.2	wpart_include	68
8.2.3	wpart_lang	69
8.2.4	wpart_list	69
8.2.5	wpart_lookup	71
8.2.6	wpart_paginate	72
8.3	Tuples for HTTP server	74
8.3.1	Overview	74
8.3.2	Types of tuples	74
9	Credits	76

1 Erlang Inside

1.1 Overview

The ErlangWeb is an open source framework for applications based on HTTP protocols, giving the developer better control of content management. With ErlangWeb's simple but extensible concept of including dynamic content in pages, libraries of reusable components can be built. Currently it supports *INET*¹ and *Yaws*² web servers, but others are planned in the future.

The ErlangWeb platform has been developed by *Erlang Training & Consulting Ltd.* in the past three years been used in many commercial and high profile projects.

The official website of the framework is <http://erlang-web.org/>.

1.2 Easy start

In order to make the start of the adventure with ErlangWeb easier, there is a start-up script, which can prepare all the environment for a new project. Currently only *Linux*-based systems are supported by the script.

Before we will be ready to use the framework for the first time, we should compile it. To do this, we should type in the sources root directory:

```
bin/prepare.erl
```

To create a proper directory structure, we should type

```
PATH_TO_THE_ERLANG_WEB_ROOT_FOLDER/bin/start.erl
```

inside a directory where we want to start our work.

The generated output should look like this:

```
erlang@erlangweb ~/dev/erlangweb/new_project/ $ /usr/lib/ew-1.2/bin/start.erl
Element created: config
Element created: docroot
Element created: log
Element created: pipes
Element created: bin
Element created: templates
Element created: lib
Element created: releases
Element created: releases/0.1
Element created: docroot/conf
```

¹part of Erlang/OTP, see more on <http://www.erlang.org/doc/apps/inets/index.html>

²see more on yaws.hyber.org

```
Element created: bin/heart
Element created: bin/to_erl
Element created: bin/run_erl
Element created: bin/start
Element created: bin/stop
Element created: bin/connect
Element created: bin/start_interactive
Element created: bin/start_erl
Element created: releases/start_erl.data
Element created: config/dispatch.conf
Element created: config/errors.conf
Element created: templates/404.html
Element created: templates/501.html
Element created: config/project.conf
Element created: templates/welcome.html
Element created: releases/0.1/start.rel
Element created: releases/0.1/start.script
Element created: releases/0.1/start.boot
Element created: config/yaws.conf
Element created: docroot/conf/mime.types
Element created: releases/0.1/sys.config
Element created: bin/compile.erl
Element created: bin/add.erl
```

Now we must create our new application tree in *lib* directory and we can start a work with ErlangWeb framework!

By default the main and only rule in *dispatch.conf* is set to render the *templates/welcome.html* page after entering the *http://localhost:8080/* address.

1.3 Compiling the project

As Erlang runs interpreted virtual machine code, we must compile and load changed parts of code to see the effects. To make it easier and faster, we can use *Emakefiles* - the build utility similar to *GNU Makefile*.

In simplest case we will need to type:

```
bin/compile.erl
```

inside the root directory of our project. This command will force Erlang's make to compile updated files.

In other case, when we need to compile and create a new release (for embedded system) we will have to run:

```
bin/compile.erl release Vsn
```

where *Vsn* is the version of our release (by default set to 0.1).

Details of the *bin/compile.erl* script are described after running

```
bin/compile.erl help
```

1.4 Adding new application

Our next goal is to create our own Erlang Web application. To do this, we should prepare the environment:

```
bin/add.erl
```

and follow the instructions displayed on the screen. This script will create the whole directory tree and update the main Emakefile file for us.

1.5 Running the server

After creating all needed modules and preparing proper configuration files we will be ready to start our own Erlang server.

There are two ways to achieve that:

- run it in embedded mode
- run it in interactive mode

Both ways are supported by created start scripts. Differences between those modes are described on <http://www.erlang.org/doc/>.

1. Embedded mode

- To run our system just type:

```
$ bin/start
```

To connect to Erlang shell:

```
$ bin/connect
```

To stop the system:

```
$ bin/stop
```

2. Interactive mode

- To start the system with interactive shell running on top of the INETS server:

```
$ bin/start_interactive inets
```

or

```
$ bin/start_interactive
```

In order to run it on top of the Yaws server:

```
$ bin/start_interactive yaws
```

2 Directory Structure

2.1 Overview

ErlangWeb is built upon OTP principles, which implies some conventions on it. One of them is directory structure. Project must contain at least the following set of directories:

- config
- docroot
- lib
- log
- pipes
- priv
- releases
- templates

2.2 Directory Meaning

config directory is the place where all configuration files should be placed. The must-have files are:

dispatch.conf - routing rules for dispatcher

errors.conf - error templates definition

project.conf - main configuration file for the whole project

configuration file for server - server-specific configuration file (for example for Yaws it will be *yaws.conf*)

Moreover, we should place here server certificates and keys files (for example in *config/keys*).

docroot directory is a folder for content, which should be served directly by server (without processing by our application), like CSS files, images, binary files, etc. To access it directly, we should place a static route rule in *dispatch.conf* file (or one of the included ones) with target set to **enoent**.

lib directory is the place where applications created by users should be placed. Directories *lib/*/ebin* are automatically added to virtual machine path. After the installation, it should contain either symbolic links or copies of all used Erlang application, i.e. *stdlib*, *kernel*, *mnesia*, *compiler*, *eptc*, *wpart*, *wparts*, *runtime_tools* and *yaws* or *inets*.

Each directory inside *lib* must satisfy OTP directory structure principles (contains at least folders: *src* for Erlang source code, *ebin* for Erlang object code and *.app* file, *priv* for application specific files and *include* for include files).

log directory is a container for logs created during running in embedded mode. Moreover, Yaws server default configuration file (*yaws.conf* copied from *Yaws* config directory) points this folder as a directory for its logs.

pipes directory holds the OS-level pipes for communicating with the shell of Erlang embedded system.

priv directory is a place for all project specific data: for example static html content (e.g. in *priv/static*).

releases directory contains separate subdirectories for each release version. Version folders should contain *.rel* file, boot script *start.boot* and optionally *relup* file. In the *releases* directory we have to put *sys.config* as well.

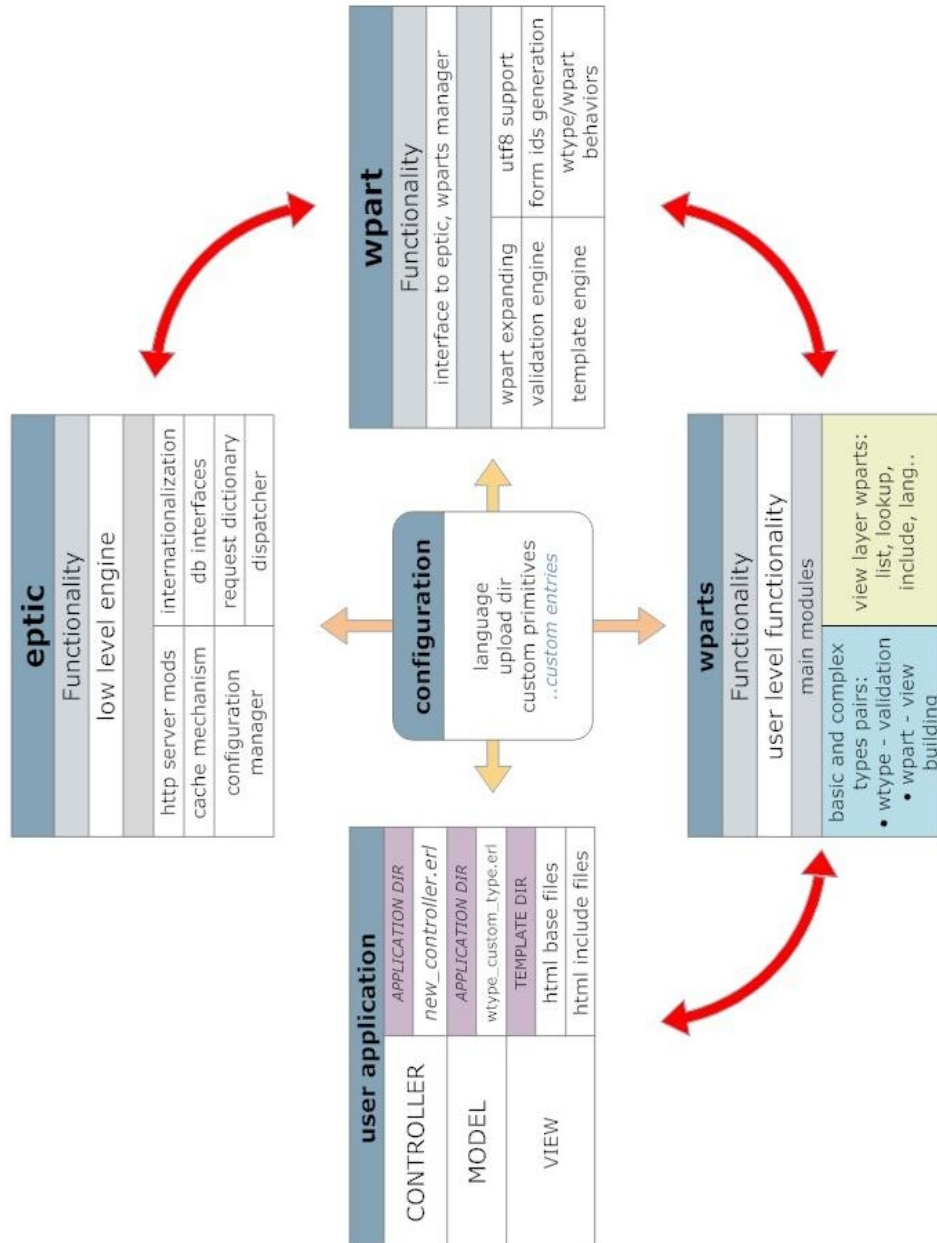
templates folder is a place for all template files used within the project. By default cache files are kept in *templates/cache* folder.

2.3 Example of a directory tree

```
.
|..config
|----dispatch.conf
|----errors.conf
|----project.conf
|----yaws.conf
|..docroot
|..lib
|----eptic
|.....|----ebin
|.....|----include
|.....|----priv
|.....|----src
%%...
|----myapp
|.....|----ebin
|.....|----include
|.....|----priv
|.....|----src
|----runtime_tools
|.....|----ebin
|.....|----include
|.....|----priv
|.....|----src
|----second_app
|.....|----ebin
|.....|----include
|.....|----priv
|.....|----src
|..log
|..pipes
|..priv
|..releases
|----sys.config
|..templates
```

3 MVC - What is What

Major idea of ErlangWeb architecture is make separate areas responsible for adequate functionality. Framework consists of *epctic*, *wpart* and *wparts*. *Epctic* communicates with *wpart* and *wpart* manages *wparts*. It is necessary to keep global configuration in separate file – it is used by many components. User's project is usually kept in separate folder and there is place for custom modules, controllers, etc.



4 Philosophy

4.1 Architecture of Types

4.1.1 Overview

One of the first steps during building any application is the design. This is a point when we make the decisions on the level of abstraction represented by modules, functions and variables. MVC pattern gives developer mayor guidelines on internal structure. This chapter will more describe Model part, however in general it is the result of portion of data required by user. So each choice we made on model will be mirrored directly to the final user and to the amount of information given to him.

4.1.2 Origin of a problem

Dynamic web services are build on the top of database. Custom types are represented by tables. E.g. *Article* could be a custom complex type stored in table *articles*. It is complex type because it consists of basic types: *string*, *text* and *date*. Working with Erlang on the edge of functional and declarative world raises some issues about typing. Mnesia³ do not store data as a specific type because Erlang is dynamically typed. Building, for example, HTML code of forms usually uses information from database fields definitions. To sum up we need a mechanism on framework side to map complex types elements to their types with minimum storage and performance requirements.

4.1.3 User friendly complex types

To solve an issue described in the previous paragraph, there is a nice and convenient way of defining your own types in ErlangWeb. Such a definition consists of two records. For *articles* it would be **article** and **article_types**. Let's look at definitions below:

```
-record(article, {
    id,
    title ,
    text
}).

-record(article_types, {
    id = {integer, [{description, "ID"},
                  {min, 1},
                  {private, true},
                  {primary_key}]},
    title = {string, [{description, "Title"},
```

³DBMS included in OTP

```

        {max_length, 255},
        {min_length, 1},
        {html, []}}},
text = {text, [{description, "News Content"},
               {max_length, 1 bsl 15},
               {rows, 10},
               {cols, 40},
               {html, ["u", "b", "i", "a", "h1", "h2", "h3",
                      "ul", "ol", "li", "br", "hr", "img",
                      "center"]}]}
}).

```

We place the record definition in *article_records.hrl* file in the **include** folder of our web application. The only place we should include them is *wtype_articles.erl* which represents the model. Names ended with *'types'* and *'records'* are obligatory for those who want to use generic functions of framework.

Now we know what is a complex type (article) and where is its definition.

What kind of benefits we can expect? Thanks to *'_records'* building HTML forms is done automatically. Validation of input values is generic. Selecting values from post and feeding controllers - generic. It is dynamic — because *'wtype_'* implements obligatory function which returns types, this is a time and place to modify and prepare dynamic types.

Complex types can be nested. To achieve that just use as a name of type – newly defined type.

In this tutorial we will try to explain each of those mechanisms to give chance to customize or improve them. Look for specification and users guide in Ref. Manual.

Derived (*wpart_derived*) is responsible for building forms. For generic validation it is **Validate** (*wpart_valid, validate_tool*).

4.1.4 Deeper

As we could see in the source code above, record *_types* consists of fields which values are by default types and options of those types. E.g.

```

title = {string, [{description, "Title"},
                 {max_length, 255},
                 {min_length, 1},
                 {html, []}}},

```

Specification:

```
Name = {Type, Options}
Type = atom()
Options = [Tuple]
Tuple = tuple()
```

List of basic types is held in *basic_types.conf*, which is loaded to ETS table at the start up application or during the configuration reload. For now, it covers almost all useful types (and we can add new at any time).

```
{integer, string, date, bool, enum, text, upload,
password, multilist, time, datetime, autocomplete, csv}.
```

Options for basic types are handled in **Wparts** application. They are in pairs *wtype_name.erl* and *wpart_name.erl*. Supported options are listed in Reference Manual. *wpart_valid* can be understood as a router delegating validation of declared value to corresponding *wtype_name.erl* which must implement functions *validate/1* and, in case of user defined type, *get_record.info/1*.

Validator is called from controller with argument saying where to look for definition of the complex type. Of course it goes to *wtype_complex_name.erl* (in application folder - where it represents model). It must implement function *validate/1* which passes included types with options to validator.

Finally, to understand it better, let's analyze use case of adding link to bookmark database (URL(*string*), title(*string*), category(*enum*)) with option of adding our own category (not only listed by enum type). Source codes of records will be shown in later examples - at the moment we will focus only on the data flow.

Simplified control flow:

1. Controller is asked to render form (we want to add some new link)
 - *wpart_form* uses *wpart_derived* to build adequate input forms
 - *wpart_derived* asks *wtype_link* for complex type definition.
 - *wtype_link* prepares feed for enum value from category store and sends complete dynamically created types to derived.
 - *wpart_derived* calls *wpart_string*, *wpart_enum* (each of them is a basic type included in link) to build HTML form tags with unique names
2. Form is rendered, we fill it in and click *submit* button
 - all values are sent as POST
 - *dispatcher* identifies function *link:add* but there is a validation required before calling the *add* function

- controller calls *wpart_valid*
- *wpart_valid* goes to model and asks for types of link; recreates unique names and takes the values from POST; sends them to adequate *wtype_some_basic_type*
- Every *wtype_some_basic_type* validates input value from HTML form with build-in rules and checks, if options defined by user are obeyed. On succes (true) sends response with *ok*. Otherwise (false) sends *error* with both *Reason* and input
- When all responses are returned, *wpart_valid* prepares a list with validation results and values; sends it to controller
- controller uses generic tools to retrieve information from list, feeds function link:add with values or builds error message

3. Function gets arguments

- values can be interpreted and are send to model
- *wpart_link* knows about record definition and can save them into mnesia

Custom types built on base of basic types are integral part of framework. Even if it is more popular now to declare whole form as a Java Script with built in validation, it is still usefull to have the internal check of HTML types with pre-defined attributes. On the other hand, it is very simple to build and add our own basic type hypothetically *JS_form* and use it inside custom type – example of that is type *autocomplete* which is fed by controller with necessary values to autocomplete.

4.2 Dispatcher

4.2.1 Overview

Routing requests is one of the most important aspects in each web service. Two main reasons for that are providing user-friendly URLs and having the control on access to the service content. *E_dispatcher* has both of them.

In order to provide mapping URL - view/controller we must fill dispatcher configuration file (*dispatch.conf*) properly. This file (placed in *config* directory) is read during the start of the application, so after making any change inside *dispatch.conf* (or configuration files included in it) we must force dispatcher to reload its routing table. It could be done by typing

```
e_dispatcher:reinstall().
```

4.2.2 Types of dispatching

There are two basic types of dispatching: **static** and **dynamic**.

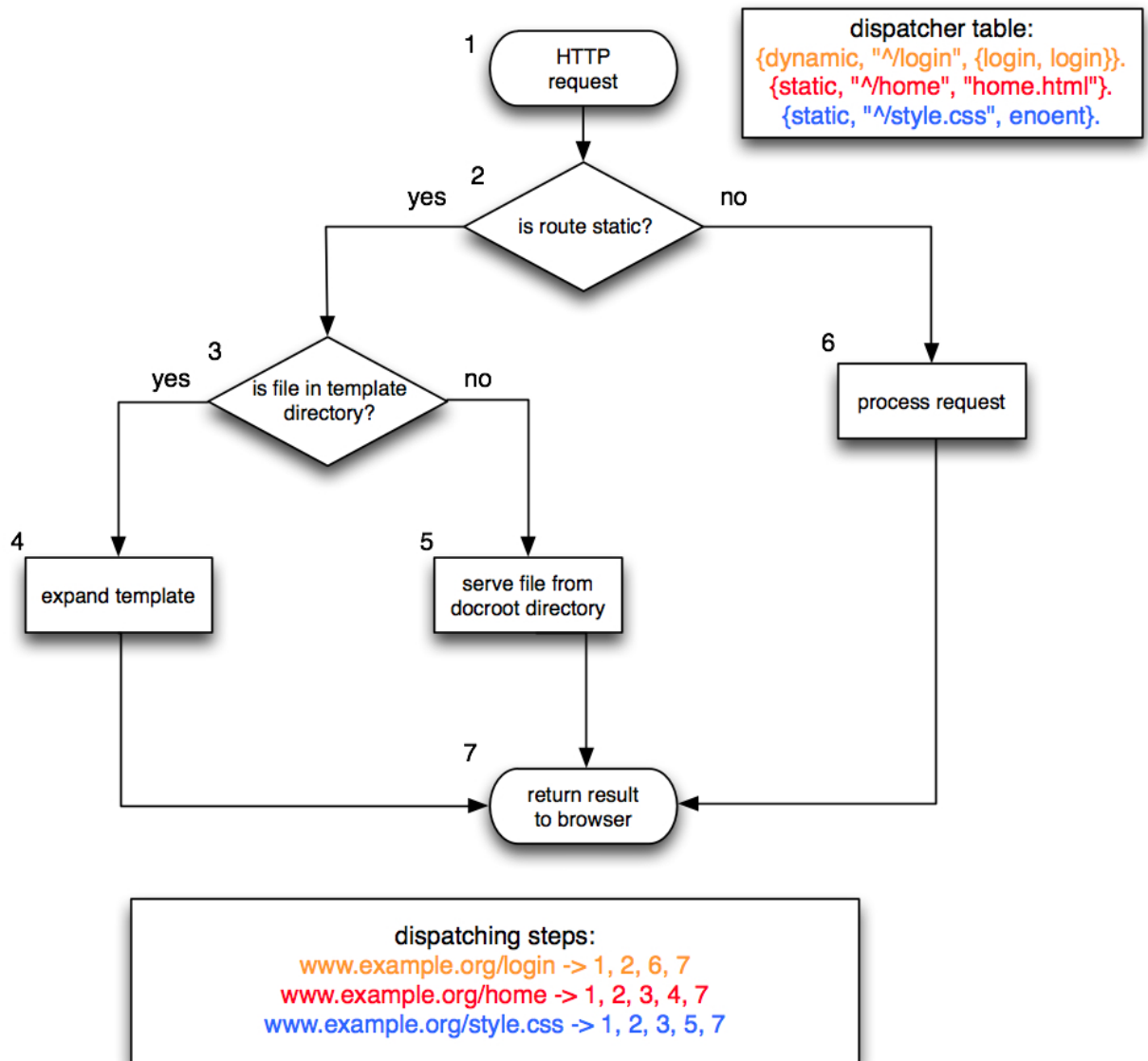
Static dispatching Static dispatching has been designed for accessing the content which doesn't need to be generated for request (which is *static*). We can serve files directly as they are (for example some media: images, css and so): without inspecting the content and changing the access path, or we can point the template which should be expanded.

In both cases we don't hit controller at all: we must know exactly what is the response for the request in the moment of creation of the dispatcher entry.

Dynamic dispatching Dynamic dispatching involves controller in handling requests. The URL user entered in his browser is mapped onto the call of the controller function: so we can build the content on the fly (fetch data from database, control the access, validate input data).

4.2.3 Control flow

After receiving request from client, application must handle it in some way. At first dispatcher checks if the URL matches any of the static entries. If so, the dispatcher tries to get target file from *templates* directory. If it fails, it looks for it in *docroot* folder. In other case - when route is dynamic - dispatcher passes the request to the target controller.



4.2.4 dispatch.conf

dispatch.conf file is placed under *config* directory - is the one of the must-have files in our application. It consists of Erlang tuples which tell dispatcher how to handle incoming request.

Each entry contains **Regexp** element - dispatcher checks if it matches the request URL. First matching entry will be used to handle the request.

The following types of entries are allowed in *dispatch.conf* file:

- {static, Regexp, File}

Statically dispatches the request to **File**. **File** should be placed in *templates* directory. If there is no file with given name, server tries to find it in *docroot* folder.

To make file serving faster, we can set **File** to **enoent** atom - it means that server should not look for it in *templates* directory.

- **{dynamic, Regexp, {Module, Function}}**

Module:Function will be called in order to satisfy the request.

- **{dynamic, delegate, Regexp, File}** To make *dispatch.conf* file clearer, easier to read and maintain we can use **delegate** entry. At first dispatcher reads the **File** and suffixes with **Regexp** all **Regexps** in entries in read file.

This kind of entry does not have any practical meaning - it only helps to structuralize the application. The same effect can be achieved by explicitly placing all the entries from **File** in original one (with suffixed **Regexp** element).

4.2.5 Named subpatterns

Named subpatterns is the mechanism used for extracting the certain information from the input string. Having regular expression we are able to specify the semantic of the incoming URL and get its value. Moreover, the named subpatterns allows us to do a simple syntax validation.

The syntax is the same as in dynamic dispatcher rules, but the regular expression is a little bit modified. If we want to bind some part of the input string to the given name, we should use the following syntax:

```
"HeadOfRegexp(?<NameOfThePattern>RegexpPart)TailOfRegexp"
```

So when the whole regular expression matches the request URL, it will extract its selected part and tag it with the specified name. The list of extracted values will be passed to the first function on dataflow list (if we are not using the named subpatterns mechanisms we will get the empty list there). The passed list will have format:

```
[{name_of_the_pattern, Value} | ...]
```

For example, when we run a blog and we want to display our post by entering the address */show/post/Post_no*, we can do it using named subpatterns:

```
{dynamic, "^/blog/post/(?<post_no>[0-9]+)$", {blog, display_post}}.
```

So when the incoming URL, let's say */blog/post/23*, could be described by the regular expression: *^/blog/post/[0-9]+\$*, then the part responsible for post number will be extracted from it. The result of the named subpatterns will be a property list:

```
[{post_no, "23"}]
```

and it will be passed to the first function on the dataflow list.

We can place more than one named subpatterns in the regular expressions, for example:

```
{dynamic, "~/blog/post/(?<post_no>[0-9]+)/comment/(?<comment_title>.+)$}.
```

for `"/blog/post/5/comment/first_comment"` URL, dispatcher will pass

```
[{post_no, "5"}, {comment_title, "first_comment"}]
```

to the first function on the dataflow list.

4.2.6 Skipping dispatcher

Because sometimes our service provides very simple API, we do not need dispatcher support. Moreover, *regexp* module is not as fast as Perl equivalent, so for efficiency reasons we will decide that we want uglier URL's instead of many regular expression matches. The last reason for providing the dispatcher omission functionality is to keep the backward compatibility with the prior version of *ErlangWeb*.

If we want to skip dispatcher, we must prefix the URL with *app/*. The meaning is as follows:

```
http://example.org/app/Module/Function/View.part1/View.part2/.../View.partN
```

The call path is split with */* and the *Module:Function* is called (of course following the dataflow rules). In case *Module:Function* call returns the *template* atom, the *View* will be expanded (path to the template will be created by joining the *View.partN*).

4.2.7 Example

This is a simple example of nested dispatcher configuration files:

```
_____ dispatch.conf _____
1 {dynamic, "[/]*$", {main, home}}.
2 {dynamic, "~/index.html$", {main, home}}.
3
4 {dynamic, delegate, "~/user", "config/dispatcher/user.conf"}.
5
6 {static, "~/about$", "about.html"}.
7 {static, "^style.css$", enoent}.
```

and the included file:

```
_____ user.conf _____
1 {dynamic, "/create$", {users, create}}.
2 {dynamic, "/delete$", {users, delete}}.
3
4 {static, "/welcome$", "users/welcome.html"}.
```

The following services will be accessible on corresponding URLs:

- "/" ⇒ function call **main:home**
- "index.html" ⇒ function call **main:home**
- "style.css" ⇒ direct access to *style.css* file in *docroot* directory
- "about" ⇒ expanding template *about.html*
- "user/create" ⇒ function call **users:create**
- "user/delete" ⇒ function call **users:delete**
- "user/welcome" ⇒ expanding template *users/welcome.html*

4.3 Validation

4.3.1 Why do we need validation

ErlangWeb like any other framework is a natural front end to existing systems that communicates with human beings. It results with information exchange. All in all we need to know type of data in the system. Even if we believe it is well formed and comes from secure source, we need to render it somehow. Secondly, more important is validating input values coming over HTTP from user.

4.3.2 Expectations

Set of types of the expected data from final user is declared in header file representing *custom type*. We already know that *Validator* collects it and sends to proper functions validating *basic types*. Let's see it by example of code that is built around the bookmark links.

Links records

```
-record(link,{
    id,
    title,
    uri,
    text,
    category,
    new_cat
}).

-record(link_types,{
    id = {integer, [{description, "Link ID"},
                  {private, true},
                  {primary_key}]},
    title = {string, [{description, "Title"},
                    {min_length, 1}]},
    uri = {string, [{description, "URI http://"},
                  {min_length, 1}]},
    text = {text, [{description, "Description"},
                  {max_length, 255},
                  {rows, 5},
                  {cols, 40}]},
    category = {enum, [{description, "Category" },
                     {optional, ""}]},
    new_cat = {string, [{description, "New Category"},
                      {optional, ""}]}
}).
```

Including records The simplest form of wtype file below.
Obligatory function `validate/1`

```
1 -module (wtype_link).
2
3 -export([validate/1, get_record_info/1]).
4
5 -include("../include/link_records.hrl").
6
7 get_record_info(link_types) -> #link_types{}
8 get_record_info(link) -> record_info(fields, link).
9
10
11 validate(From) ->
12     SuperField = get_record_info(link),
13     SuperType = get_record_info(link_types),
14
15     wpart_valid:validate(SuperField, SuperType, From ++ ["link"]).
```

String in line 15 is quite important for building default unique names. We use name of custom type to let generic tools to recreate them. Function `get_record_info/1` is a space for dynamic changing types (e.g. on base of authentication).

Validating an integer Our *Link* type was analyzed and now incoming values from user are send to validate functions of basic types. They are on framework side, but in case of adding new type let's see example of *wtype.integer*.

Obligatory function `validate/1` has to be exported.

```
1 validate({Types,Input}) ->
2     case wpart_valid:is_private(Types) of
3         true ->
4             {ok, Input};
5         false ->
6             case catch list_to_integer(Input) of
7                 Int when is_integer(Int) ->
8                     case check_min(Int, Types) of
9                         {ok, Int} ->
10                             case check_max(Int, Types) of
11                                 {ok, Int} -> {ok, Int};
12                                 ErrorMax -> ErrorMax
13                             end;
14                             ErrorMin -> ErrorMin
15                         end;
16                 _ -> {error, {not_integer, Input}}
```

```

17         end
18     end.

```

In line 1 function takes arguments. They are sent by validator. It is always a tuple {Types, Input}. 'Types' is a rewritten list from record *link_types*. Last important thing is specification of *ok* and *error* result. Only format from lines 11 and 16 is valid.

Private field. As a careful reader can see, there is a special case for private field in line 3. *Id* field in record *link* is private. It will not be visible or editable for user and it will not be validated.

Back to controller Finally let's see how to call generic validation from controller. Just code of date flow function and validate function.

```

1     %...
2     dataflow(create) -> [authenticate, validate, validate_logic];
3     %...
4     validate(create,_) ->
5         validate_tool:validate_cu(link, create);
6     %...
7     error(create, not_valid) ->
8         Err = wpart:fget("__error"),
9         Message = "ERROR: Incomplete input or wrong type in form!" ++
10        " Reason: " ++ Err,
11        wpart:fset("error_message",Message),
12
13        Not_validated = wtype_link:prepare_initial(),
14        wpart:fset("__edit", Not_validated),
15        {template, "templates/link/add_link.html"};
16     %...

```

Any of calls for framework functions is not obligatory (we use in line 5 validation). We can easily create our own validator and call it. What happens above? *dataflow* makes function *validate* to be called [3]. It uses framework tool to call and analyze response from *wpart_valid* [6]. On error *dataflow* makes sure that function *error/2* is called. It forms Error Message and fills form with initial values (original input by user) - wrong rounded with red frame (style on CSS side - obligatory class name *form_error*).

Special cases What if we want to use one record definition for many controllers and models, which will interpret data differently? To achieve that we need to export from model (*wtype_custom_type*) *get_parent_info/0* function.

```

-module(wtype_download).
-export([get_parent_type/0]).
%...some code
get_parent_type() -> link.

```

4.3.3 Validate_tool

In the example above we could see *validate_tool* in action. Let's take a closer look at engine inside this module.

Validate_tool was created to cooperate closely with *validator*. It is connected with data format returned by *validator*. Below is an example of such a list:

```

{error, [{ok, []}, "link_new_cat"],
        {{ok, []}, "link_category"},
        {error, {empty_input, undefined}}, "link_text"},
        {ok, "http://asd"}, "link_uri"},
        {ok, "VeryInterestingTitle"}, "link_title"},
        {ok, undefined}, "link_id"]]}

```

Specification:

```
{GlobalResult, ListResult}
```

```
ListResult = [ItemResult]
```

```
ItemResult = {Result, UniqueName}
```

```
GlobalResult = error | ok
```

```
Result = {ok, Input} | {error, ErrorCode}
```

```
ErrorCode = atom()
```

```
UniqueName = list()
```

To have full understanding of the input data *validator* passes almost all extracted and analyzed data. Handling it in each controller would be difficult. That is why we need *validate_tool*. Its main functionality is to build record and send it to controller. Next issue is handling *error* 'branch'. *Validate_tool* inserts all failed long unique names into request dictionary. They are caught by *wpart_derived* and then error frame is built. In the same time error atoms are collected and error message string is set in dictionary. All of those information is handled by proper clause of local *error/2* function inside of controller.

4.4 Internationalization

4.4.1 Overview

The bigger your web service is, the more attention you should pay to make it more accessible to other users. One of the main barrier is language. Because of MVC model we split the application into logic, behavior and view, but there is sometimes a need for a next partition.

Creating multi-lingual service we should only focus on changing the content we want to display - the view should remain always the same. E_lang provides a very convenient API to keep views the same and switch only the dictionary which application uses for each request.

E_lang has been built upon a key-value mapping. Basic idea is to use only the keys in our application instead of language-specific elements and place all the translation in the separate files.

Keys for translation used within application are strings, which can be parsed in two ways:

string contains ":" - string is split by **:** character and the actual key is a tuple of string - tokens created by splitting the original string

string doesn't contain ":" - string is the key

Language files are read into memory during the start of the application, so after any change in those files there is a need to reload them. It could be done by calling

```
e_lang:reinstall().
```

4.4.2 Defining translations

Preparing *project.conf* First step of translating process is to put *language_files* tuple in *project.conf* file. This two-element tuple should be in the following format:

```
{language_files, [LanguageFileSpec|...|...]}
```

where **LanguageFileSpec** is **{LangCode, PathToTranslationFile}** **LangCode** is an atom representing the language of the translation file.

Preparing translation files Secondly, we should create all files we specified in *project.conf* and place there translations for all the keys we used in our application.

Translation file contains Erlang tuples in format: **{Key, Translation}** where **Key** is either a single string or a tuple of strings (look at *Overview* section).

4.4.3 Translating

There are three ways of accessing translated strings:

- We can use `<wpart:lang key=Key />` inside the *html* file. During the process of tags expanding this tag will be replaced with the proper translation.

- Call `wpart_lang:get_translation/1` with key we want to translate.
- Use tuple `{key, Key}` in *description* option in *.hrl* record definition file.

4.4.4 Control flow

All types of translating (using `wpart`, specifying description in *.hrl* files and explicitly calling *get_translation* function) has the same control flow.

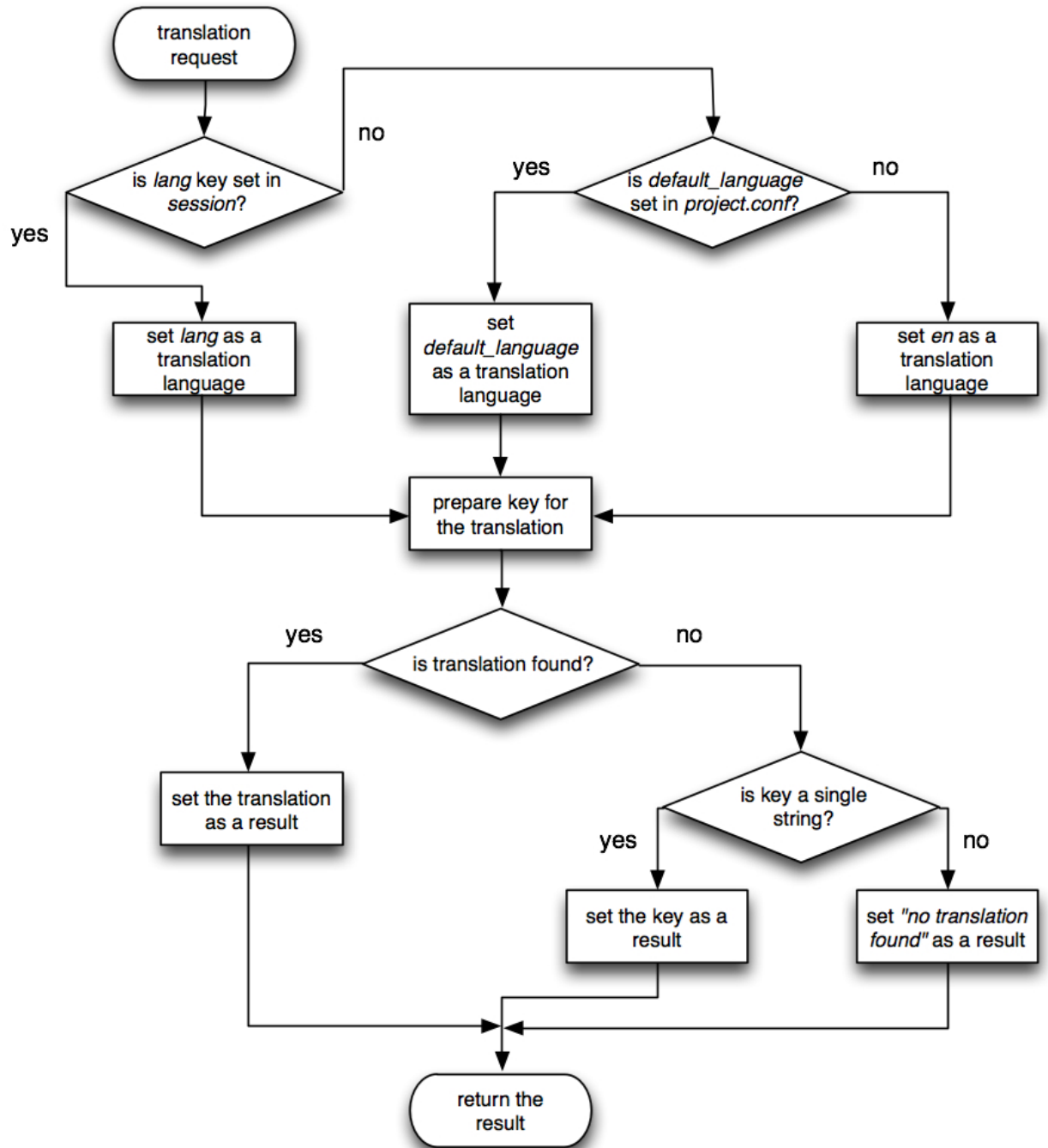
The target language is chosen in the following order:

1. from **lang** key kept in **session** in **e_dict**
2. from **default_language** option in *project.conf* file
3. if none of this options is set, `e_lang` assumes we want to use English (*en* language code)

Language codes used in application must be the same as those declared in *project.conf*.

Translation process behaves as follows:

1. if translation for given key is found, it is returned
2. otherwise, if key is a single string (not a tuple), the key is returned
3. in other cases "no translation found" string is returned



4.4.5 Example

html embedding

```
1 ...
2 <h1><wpart:lang key="contact:header"/></h1>
3 erlangweb@example.org<br/>
4 <wpart:lang key="back"/>
5 ...
```

In this example two translations will be used: one with key {*"contact", "header"*} and one with *"back"*.

get_translation call

```
1 ...
2 ErrorMessage = wpart_lang:get_translation("errors:no_such_login"),
3 ...
```

ErrorMessage variable will be bound to the translation corresponding to the {*"errors", "no_such_login"*} key.

description option in record definition file

```
1 ...
2 -record(login_types,
3     {login = {string, [{description, {key, "login:login"}}]},
4     password = {password, [{description, {key, "login:password"}}]}}).
5 ...
```

Descriptions for *#login_types.login* and *#login_types.password* will be found under the keys {*"login", "login"*} and {*"login", "password"*} respectively.

language configuration file: "en.conf"

```
1 {"contact", "header"}, "My contact details".
2 {"back", "Go Back"}.
3 {"errors", "no_such_login"}, "There is no such user in the system".
4 {"login", "login"}, "Login".
5 {"login", "password"}, "Password".
```

This file includes all the translations for the examples quoted above.

4.5 Request Dictionary

4.5.1 Overview

Each request to HTTP server mod sets up *request dictionary*. Its life time is limited only to this one request processing time and after that it is cleaned up. Module which provides an API to it is *wparts*, however it was introduced only to improve handling of dictionary values. It is developers decision to brake framework idea of calling *request dictionary* through *wpart* application and work directly with *aptic* — it can give some advantages in specific situations.

Usage

In general, it is possible to set {Key, Value} and later get those values from dictionary (they will be stored under the *Key*). Below is presented a list of functions operating on request dictionary.

4.5.2 Request dictionary API

Wpart API

Dictionary API covers functions: *fset/2*, *finsert/2*, *fget/1*. Although *fset* takes only two arguments { *Key*, *Value* } it is possible to create deeper structures. In version 1.1 it is only 2-level tree. To separate parent node from leaves in *Key* ":" (colon) is used. *Wpart* API uses only string as keys, thanks to that it is possible to call *dictionary* from *View* application level (*wpart:lookup* look for spec in Manual).

How to put values into dictionary?

```
fset/2      wpart:fset("SomeKey", SomeVal),
```

All different values stored under *Key* are in dictionary during request time.

fset/2 - possible to create family of values in one table

```
wpart:fset("List:Key", Val),
```

finsert/2 - analogical structure like *fset* but replaces values if the key is the same

```
wpart:finsert("key", Val),
```

How to get values from dict?

```
fget/1      wpart:fset("SomeKey"),
```

```
fget/1      wpart:fget("People:Developers"),
```

If there is more values under one key function returns a list of them.

Low level calls API

Setting values

```
fset/2      eptic:fset("some key", SomeVal),
```

Even if some values are set under one key they are not missing. **THIS IS MAJOR CHANGE FROM VERSION 1.0.** It has been implemented because HTML has some tag constructions which do not allow to set name parameters in dependent tags - just the cover tag has it.

fset/3 - possible to create family of values in one table

```
eptic:fset("list","key",Val),
```

finsert/2 - analogical structure like fset but replaces values if the key is the same

```
eptic:finsert("key", Val),
```

Recovering values

```
fget/1      eptic:fset("some key"),
```

```
fget/2      eptic:fget("post","name1"),
```

If there is more values under one key function returns a list of them

4.5.3 Special cases

Dictionary is also used by framework. E.g. POST messages from browser go over it. Each message on framework side passed by request dictionary should start with double underscore. Because of backward compatibility issues some of them still are not. Below is the list of restricted keys.

key	description
__https	Bool()
__controller	current controller
post	POST
get	GET
__not_validated	record from form which failed validation
__error	reason of validation failure
__type	used by wpart_page - information about whelp which feeds wpart_page list
__types	dynamic feed for types addressed to wpart_derived
__edit	default or initial values for derived to fill up form
session	talks to session table in ETS via <i>e_session</i> which synchronize them
__path	holds URL of current request
__primary_key	identifies values during update
__cookies	list of tuples { <i>CookieName</i> , <i>CookieVal</i> } related to our service

4.6 DBMS

4.6.1 Overview

During the build of your application we will notice there is a very narrow set of most common database operations:

- saving the element
- reading the element with the given ID
- reading all the elements from the given domain
- updating the existing element
- deleting the element with the given ID
- obtaining the next available ID within the given domain

The *e_db* module provides the convenient API to all those operations.

4.6.2 e_db module

e_db module is a transparent API to the DBMS laying at the bottom of the system. It allows us to run the following operations on the database:

install() - installs the selected database. When we are using mnesia it creates the table for used ID's. Otherwise, when we are using CouchDB, creates two new databases: one for our project (with our project name - it is specified in *project.conf* file) and one for our project's ID's.

write(Domain, Element) - saves the *Element* to the database in the specified *Domain*.

read(Domain) - reads and returns a list of all the entities from the given *Domain*.

read(Domain, Id) - reads the element from the *Domain* with the given *Id*.

update(Domain, Element) - updates the *Element* in the given *Domain*.

size(Domain) - returns the number of elements stored in the given *Domain*.

delete(Domain, Element) - deletes *Element* from the *Domain*.

get_next_id(Domain) - returns the next, unique ID within the *Domain*.

4.6.3 Supported DBMS

Currently only *Mnesia* and *CouchDB* DBMS are supported. Moreover, *CouchDB* support is in incubation phase, so it is not recommended to use it in the production systems. To set the desired DBMS we should specify its name in the *project.conf* file:

```
{dbms, DBMS}.
```

where **DBMS** is either *mnesia* for Mnesia support or *couchdb* for CouchDB.

After setting this option, the access to the selected DBMS is completely transparent when we are using *e_db* module.

4.6.4 Example

Let's check out the basic use cases of *e_db* module:

```
%% creating new item
ID = e_db:get_next_id(blog_post),
Post = blog_post:create_post(ID),
e_db:write(blog_post, Post),
...
%% reading all the items
Posts = e_db:read(blog_post),
...
%% reading the particular item
Post = e_db:read(blog_post, 10),
...
%% updating the existing item
Post = e_db:read(blog_post, 3),
NewPost = blog_post:update_post(Post),
e_db:update(blog_post, NewPost),
...
%% removing the item
Post = e_db:read(blog_post, 5),
e_db:delete(blog_post, Post),
...
%% getting the domain size
Size = e_db:size(blog_post),
...
```

4.7 Project configuration file

4.7.1 Overview

project.conf file (placed in *config* directory) is the place where the settings for application are being held. Having all the things in one configuration file makes it easier to understand, maintain and develop.

Configuration file contains Erlang tuples: first element of each one is the option name; second - option value.

All the settings are read during the start of the application, so after every change we must reload them manually, by typing

```
e_conf:reinstall().
```

4.7.2 Types of options

The following options are being used in the application:

1. {upload_dir, Dir}

Dir specifies the directory, where user uploaded files will be stored. The set directory will be placed inside *docroot* folder. By default it is set to *"upload"*.

This option could be accessed with command:

```
e_conf:upload_dir().
```

2. {default_language, LanguageCode}

LanguageCode specifies the default language of translation: if none is set in *session:lang* key in **e_dict**, this one will be used. By default it is set to *en*.

This option could be accessed with command:

```
e_conf:default_language().
```

3. {language_files, [LanguageFilesSpecs]}

LanguageFilesSpecs specifies the language files with translations. This option has been described deeper in *e_lang* section.

4. {cache_dir, Dir}

Dir is the directory, where the cached templates are stored. By default it set to *"templates/cache"*.

This option could be accessed with command:

```
e_conf:cache_dir().
```

5. {host, Host}

Host is the absolute address of our service. It could be helpful in building absolute links. By default it is set to *"localhost"*.

This option could be accessed with command:

```
e_conf:host().
```

6. {primitive_types, ListOfPrimitiveTypes}

ListOfPrimitiveTypes defines user-prepared primitive types, which can be used in building application models. We have to provide both *wpart_NameOfTheType* and *wtype_NameOfTheType* modules with corresponding **wpart** and **wtype** behaviours. By default it is set to `[]`.

This option could be accessed with command:

```
e_conf:primitive_types().
```

7. {debug_mode, Bool}

Bool specifies if debugging mode is enabled. If so, all the Erlang errors will be rendered as a error 501 (with explanations) instead of displaying user-specified template. By default it is set to *false*.

This option could be accessed with command:

```
e_conf:debug_mode().
```

8. {http_port, PortNo}.
{https_port, PortNo}.

PortNo specifies the port number for incoming http and https connections. The numbers should be the same as those in server configuration file. They can be used in redirection between protocols running on different than default ports (80 for http and 443 for https).

These options could be accessed with commands:

```
e_conf:http_port().  
e_conf:https_port().
```

To easily redirect user from http to https connection, just return:

```
{redirect, "https://" ++ e_conf:host() ++ ":" ++ e_conf:https_port()  
++ "/" ++ wpart:fget("__path")}.
```

If server is running on default ports we can return:

```
{redirect, "https://" ++ e_conf:host() ++ "/" ++ wpart:fget("__path")}
```

9. `{project_name, Name}`.

Name specifies the string representing the project name. By default is set to *"erlang-web"*.

This option could be accessed with command:

```
e_conf:project_name().
```

10. `{couchdb_address, URL}`.

URL specifies the CouchDB address. This address is used only when our DBMS is set to CouchDB for communicating with CouchDB server. By default is set to *"http://localhost:5984/"*.

This option could be accessed with command:

```
e_conf:couchdb_address().
```

11. `{dbms, DBMS}`.

DBMS is the type of the DB engine used in our project. It could either *mnesia* or *couchdb*. This option has been described deeper in `e_db` section. By default it is set to *mnesia*.

We can also place our own options inside the *project.conf* file. These settings could be found in *e_conf* ets table under our own defined key.

4.7.3 Example

This is the simple example of *project.conf* file

```
1 {upload_dir, "user_upload"}.
2 {host, "example.org"}.
3 {default_language, de}.
4 {language_files, [{en, "config/languages/en.conf"},
5                  {de, "config/languages/de.conf"}]}.
6 {admin_logins, [adam, michael]}.
7 {primitive_types, [embedded_video]}.
```

The configuration of our application will be as follows:

- upload directory will be set to *"user_upload"*

- hostname will be set to *"example.org"*
- default language of translation will be *de*
- there will be two translation files: for *en* (*"config/languages/en.conf"*) and for *de* (*"config/languages/de.conf"*)
- cache directory will be set to *"templates/cache"* (by default)
- user-defined setting, **admin_logins** will be set to *[adam, michael]*
- there will be new primitive type: *embedded_video*

4.8 Data flow

4.8.1 Overview

There are a lot of cases when we want to make several actions in response of one HTTP request. Moreover, subset of those actions is the same for more than one function and the requirement to proceed to next step (fire next action) is the successful return from the current one. To achieve that dataflow mechanism has been created.

The basic idea is to provide a tool for defining flow of data from beginning of the request to its end. For example, we can specify, that before calling **user:create** we want to check the permissions of the caller, validate input data, check for uniqueness, log the activity and only if every of them has ended successfully - proceed to **create** function. It is possible also to declare some functions to execute after fuction requested by HTTP server (syntax in the next paragraph). In that case we do not have to worry about arguments requested by HTTP server mode – it is captured and values passed by "after request" functions are up to developer's idea.

4.8.2 Control flow

To enable the mechanism of dataflow, controller module, which we want to use dataflow within, must export **dataflow/1** and **error/2** functions.

dataflow/1 - Function **dataflow/1** is responsible for providing the list of functions inside its module, which will be sequentially called, one after another. Each of that function must return either `{ok, Args}` or `{error, Reason}` tuple.

Specification:

```
dataflow(ControllerFun) -> {BeforeList, AfterList} | BeforeList
```

```
BeforeList = AfterList = [Function]
```

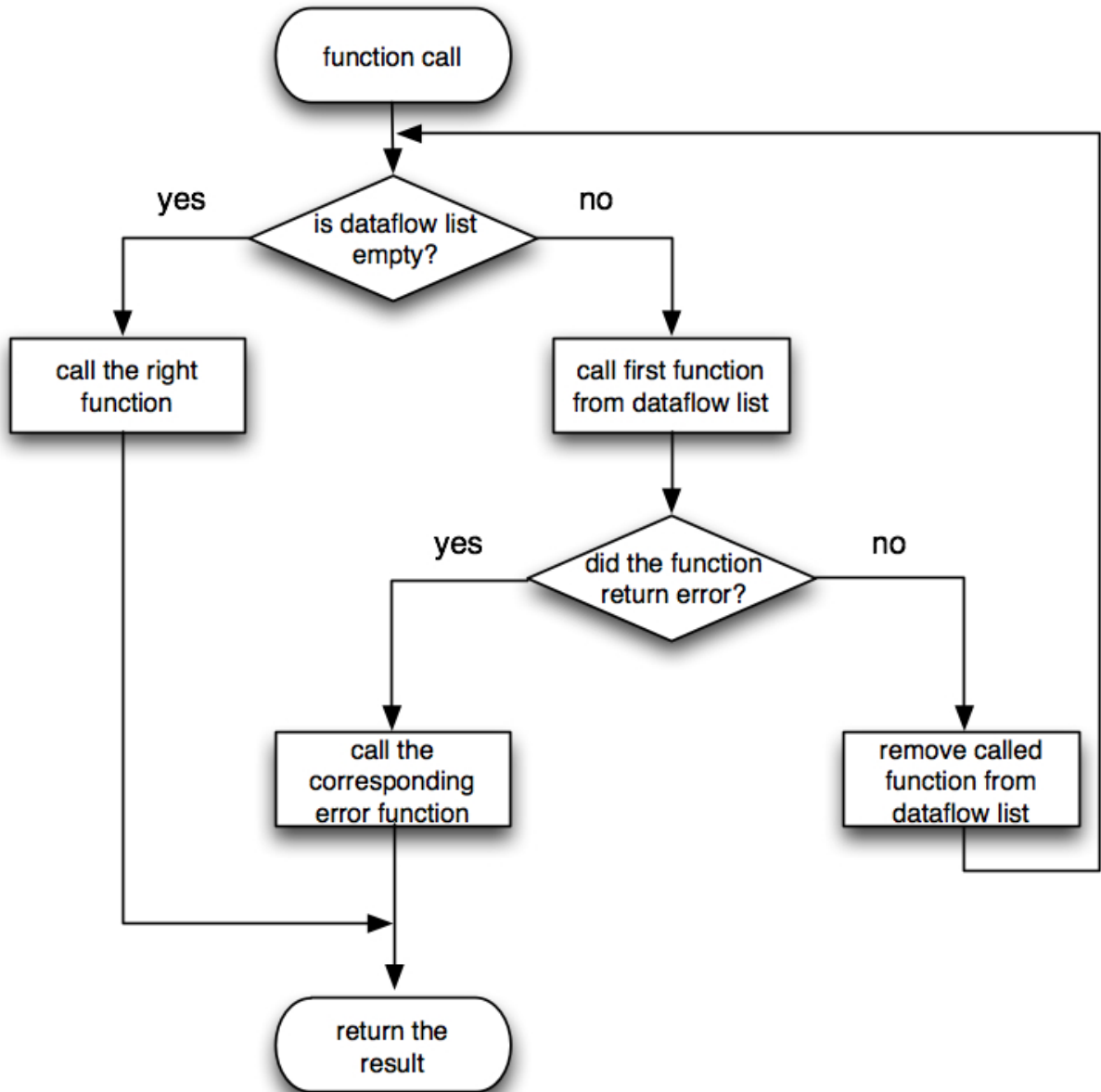
```
ControllerFun = Function = atom()
```

Args will be used as the arguments for the next function in dataflow list or as the arguments for the proper function call.

The proper function will be called only when all the functions specified in **dataflow** will succeed (and it will be called with arguments returned from the last function on the dataflow's list).

error/2 - If something goes wrong and one of the functions will return an error, the calling sequence is stopped and the **error(FunName, Reason)** is called (**FunName** is the name of the function where error occurred)

If one of the obligatory dataflow functions is not present (exported), the standard procedure is held: at first the **validate/1** function is called, then the proper one is fired.



4.8.3 Example

Let's assume we want to create the following scenario of removing user from system (**users:remove/1** function - argument is an ID of the user):

1. check if user connects *via* https

2. check if user has right permissions
3. validate passed parameters (passed in URL, like: *users/delete/3* to remove user with ID=3)
4. log the activity
5. proceed the removal function

To achieve this **users** module must at least look like that:

```

1  ...
2  -export([dataflow/1, error/2]).
3  ...
4  dataflow(remove) -> [check_https, check_permissions,
5                        validate_number, log].
6  ...
7  check_https(_, _) ->
8      case wpart:fget("__https") of
9          true -> {ok, ok};
10         false -> {error, no_https}
11     end.
12
13  check_permissions(_, ok) ->
14      %% check if user is logged in and has the good permissions
15      case Result of
16          not_logged_in          -> {error, not_logged_in};
17          not_enough_permissions -> {error, not_enough_permissions};
18          _                      -> {ok, ok}
19      end.
20
21  validate_number(_, ok) ->
22      case validate_tool:validate_number() of
23          {error, default_val} -> {ok, 1};
24          {error, nan}         -> {error, invalid_url};
25          {ok, N}              -> {ok, N}
26      end.
27
28  log(Action, Number) ->
29      %% log activity to the file
30      {ok, Number}.
31
32  error(check_https, no_https) ->

```

```
33     {redirect, "https://" ++ e_conf:host() ++ wpart:fget("__path")};
34 error(check_permissions, not_logged_in) ->
35     {redirect, "/login"};
36 error(check_permissions, not_enough_permissions) ->
37     {template, "/errors/not_enough_permissions.html"};
38 error(validate_number, invalid_url) ->
39     {error, 404}.
40
41 remove(Number) ->
42     %% proceed with user removal
43     {template, "users/list.html"}.
44
45     ...
```

4.9 Template engine

4.9.1 Overview

Template engine is a tool for faster and easier building views for our application. It allows us to create one big view from little "bricks" - parts of html code. The most common situation is when we have page header, menu page, page content and page footer. Writing each html file with header and footer can be very boring job. Moreover what if at some point of time we decide to change the header? We will have to change it's every occurrence in the hundreds of html files (in case of big service).

Template engine is in fact a set of three *wtpl* tags: *wtpl:parent*, *wtpl:include* and *wtpl:content*.

4.9.2 wtpl tags

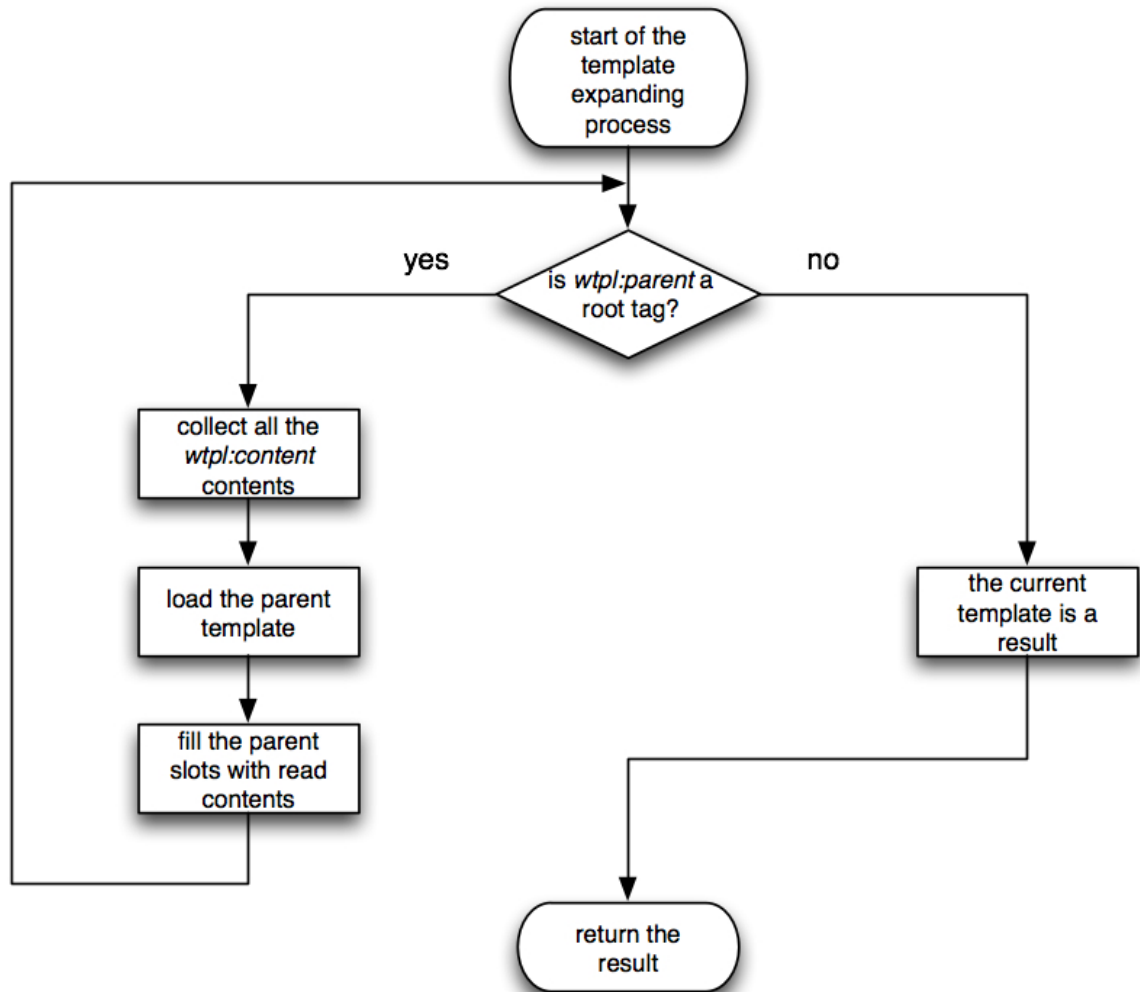
1. `<wtpl:include name=Name />` - tag which specifies the place, where the content of corresponding *wtpl:content* tag (with the same attribute **Name**) will be inserted.
2. `<wtpl:parent path=Path />` - tag which specifies which template will be filled with *wtpl:content* contents. It must be the root element of each child template (template, which contains at least one *wtpl:content*).
3. `<wtpl:content name=Name />` - tag which content will replace corresponding (with the same **Name** attribute) *wtpl:include* tag in *wtpl:parent* file.

To display the page properly, all the *wtpl:include* tags must be substituted with corresponding *wtpl:content* tags.

4.9.3 Control flow

When template expander starts his work and encounters the *wtpl:parent* tag - the template engine starts his work. The control flow is as follows:

1. collecting all the insides of *wtpl:content* tags
2. loading the file pointed by **Path** attribute in *wtpl:parent*
3. fill the possible *wtpl:include* slots with corresponding *wtpl:content* values
4. if the root element of loaded file is *wtpl:parent* expander goes to first step



4.9.4 Example

This is an example of nested template html files:

```

_____ base.html _____
1 <html>
2   ...
3   <body>
4     <wtpl:include name="header"/>
5     <wtpl:include name="content"/>
6     <wtpl:include name="footer"/>
7   </body>
8 </html>

```

```

1 <wtpl:parent path="base.html">
2   <wtpl:content name="header">
3     This is the header of the service
4   </wtpl:content>
5
6   <wtpl:include name="menu"/>
7
8   <wtpl:content name="footer">
9     This is the footer!
10    
11  </wtpl:content>
12 </wtpl:parent>

```

```

1 <wtpl:parent path="half_filled.html">
2   <wtpl:content name="menu">
3     Here we will place our menu
4   </wtpl:content>
5
6   <wtpl:content name="content">
7     And here will be the content of the page
8   </wtpl:content>
9 </wtpl:parent>

```

After expanding, final page ("*filled.html*") will look like that:

```

1 <html>
2   ...
3   <body>
4     This is the header of the service
5     Here we will place our menu
6     And here will be the content of the page
7     This is the footer!
8     
9   </body>
10 </html>

```

4.9.5 wpart_gen

wpart_gen module has been created because of the need for building HTML code efficiently inside of the Erlang modules. The idea was to remove all the HTML tags to the separate files, with extension *.tpl*, and to load and fill them in controller.

The usage of the *wpart_gen* module is very simple. At the beginning we should load our HTML snippets into the memory in order to access them as fast as we only can. It could be done by running:

```
wpart_gen:load_tpl(Namespace, Name, Path).
```

This call will load the file from *Path* and place it in the memory under the key $\{Namespace, Name\}$. Namespaces has been implemented because some names are very common, like *li* - so we want to recognize them in our controller. Loading should be performed only once - during the start of our system.

The templates snippets are the normal HTML files, but have several *marked* places, where we want to inject our content. There are two ways for specifying the templates:

named slots - they are used when we do not want to remember the exact order and number of the parameters. The slots are specified by their names. In case when developer did not pass all names during the template filling, the unfilled slots will be replaced with empty strings. The syntax is as follows:

```
...
<!-- HTMLCode -->
<% name_of_the_slot %>
<!-- HTMLCode -->
...
```

anonymous slots - has been implemented in order to achieve better efficiency. We must remember the exact number of the slots and their order:

```
...
<!-- HTMLCode -->
<% slot %>
<!-- HTMLCode -->
...
```

When we have the snippets prepared and loaded, they are ready to use. In order to get the snippet we should call

```
wpart_gen:tpl_get(Namespace, Name)
```

Then, to fill the template with our values, we should call

```
wpart_gen:build_hml(Tpl, Values)
```

where *Tpl* is a template loaded in previous call and *Values* is a list of our specific values. *Values* list format depends on the type of the slots we used:

named slots - we should pass the list of tagged tuples:

```
[{"name_of_the_slot", Value} | ...]
```

The slot with the specified name will be substituted with the corresponding *Value* which is a string. We do not have to pass all the values - the unfilled slots will remain empty.

anonymous slots - we should pass a list of values:

```
[Value | ...]
```

Length of that list must be the same as the number of the slots defined in the snippet. If the lengths differ, the *erlang:error* function will be called. Moreover, the order is important here - we have to build the list of values in the correct sequence.

The properly prepared HTML code could be returned as the value of the `#xmlText` record from the `handle_call` function.

5 Controller. One to rule them all.

5.1 Overview

This and next two chapters will guide us through building application. Bits and pieces of examples with previous configuration code should cover mayor part of application. In opposite of real web service, to make it simpler to read and learn, tutorial goes layer by layer of MVC pattern. Scope of example is around one controller and it is the best practice to map each model to corresponding controller. In this subject ErlangWeb is very RESTful. URL in REST style still waits for stable *re* module. It is possible to write them now, but with bigger effort.

5.2 The simplest case

The simplest scenario for running any dynamic content with ErlangWeb would be

- defining URL we want to access as static in the dispatcher (even if it is dynamic, it means that request go to template instead of going to controller at first)
- implement functionality in `wpart_new_name`
- create template which uses `wpart` mentioned earlier

That is all. Although that is not real controller. It is possible to have any functionality in new `wpart` but in complex projects it is not used often to display main content. Main reason is that we want to parametrize template to behave different in many cases and it can be done in controller before executing template. On the other hand, `wpart` in template takes arguments - it is all right when they are simple, short and not dynamic (e.g. customizable options in some select field or autocomplete field with 10000 choices). So when it is best to use static `wparts`? Lets take a look at *top menu* on some website.

```
1 -module(wpart_menu).
2 -export([handle_call/1]).
3
4 -include_lib("xmerl/include/xmerl.hrl").
5
6 handle_call(_E) ->
7     Options1 = [{"/", "HOME"},
8                 {"/news/", "NEWS"},
9                 {"/article/", "ARTICLES"},
10                {"/blog/", "BLOGS"},
11                {"/link/", "LINKS"},
12                {"/doc.html", "DOCUMENTATION"}],
13
14     LoggedRoot = filter:is_auth([root]),
15     LoggedOther = filter:is_auth([blogger]),
```

```

16     Options = if
17         LoggedRoot == true ->
18             Options1 ++ ["/users/", "ADMIN "];
19         LoggedOther == true ->
20             Options1 ++ ["/logout/", "LOGOUT "];
21         true ->
22             Options1
23     end,
24
25     List = lists:map(fun({Link, Name}) ->
26         whelper:prepare("list_element", [Option, Name]));
27         end, Options),
28     Ready = whelper:prepare("list_cover", [List]);
29
30     #xmlText{value=Ready, type=cdata}.

```

The example above is a simple version of menu. List of links is created. HTML code is prepared by whelper [25]. Real menu also passes e.g. id = 'active' but for simplicity it is cut off. Now, to use it, just type <wpart:menu /> in proper place of base template.

5.3 Second step: data flow and new home made controller

There is several rules each controller has to implement

1. Every dynamic call from dispatcher (exactly `e_mod_some_http_server`) must return tuple, e.g. `{redirect, "/save"}` or `{template, "/save_form"}`. Full list and spec available in Reference Manual.
2. Controller must export either `validate/1` or pair: `dataflow/1` and `error/2`.
3. To follow MVC rules keep operation on DB in model - so call `wtype`

What we need now is controller which handles saving information from form. What to do?

- build form in template
- set proper URL as dynamic in dispatcher and make it call our new function `add/1`
- implement controller
- implement model to save data

dispatcher.conf

```
{dynamic, "~/save", {new_controller, save}}.  
{static, "~/save_form", "save_form.html"}.
```

save_form.html

```
<form action="/save" method="post" accept-charset="utf-8">  
<table>  
  <tr><td>Example</td></tr>  
  <tr><td>Your name</td>  
    <td><input type="text" name="example_name" value=""/>  
    <br/></td>  
</tr>  
<tr><td>City</td>  
  <td><input type="text" name="example_city"/><br/></td>  
</tr>  
</table>  
<input type="submit"/>  
<br/>  
</form>
```

```
dataflow(save) -> [validate];

%% ...

validate(save,_) ->
  V1 = eptic:fget("post","example_name"),
  V2 = eptic:fget("post","example_city"),
  Bool1 = is_string(V1),
  Bool2 = is_string(V2),
  if Bool1 and Bool2 -> {ok, [V1,V2]};
  true -> {error, bad_values}
end.

%% ..
error(save, bad_values) ->
  {template, "templates/error/error_1.html"}.

%% ..
create(X) ->
  wtype_new_controller:save(X),
  {redirect, "/save/form"}.
```

5.4 Pure dynamic

Let's go step further. We don't want to play with HTML to build form and we need some default values in displayed view. Moreover, it would be very helpful to get error messages on bad values. Validation should be generic. To achieve that we will extend our *new_controller*. Example will present part of CRUD functionality (which stands for **C**reate **R**ead **U**ppdate **D**elete). Several prototypes of new projects in ErlangWeb crafted optimal way of implementing it. Full code is available in the *Examples* directory.

dispatcher.conf

```
{dynamic, "^/before_create", {new_controller, before_create}}.  
{dynamic, "^/create", {new_controller, create}}.
```

example_records.hrl

```
-record(example,  
    {name,  
     city}  
    ).  
  
-record(example_types,  
    {name = {string, [{description, "Name"},  
                    {min_length, 3}]},  
     city = {string, [{description, "City"},  
                    {min_length, 5}]}}  
    ).
```

save_form.html

```
...  
<wpart:lookup key="error_message" />  
<wpart:form type="example" action="\create"/>  
...
```

new_controller.erl

```
dataflow(before_create) -> [];  
dataflow(create)        -> [validate];  
  
% ..  
  
validate(create,_) ->  
    validate_tool:validate_cu(example,create);  
% .. some code
```

```

%%
%% error handlers
%%
error(create, not_valid) ->
    Err = wpart:fget("__error"),
    Message = "ERROR: Incomplete input or wrong
              type in form!"+ " Reason: " ++ Err,
    wpart:fset("error_message",Message),

    Not_validated = wtype_users:prepare_initial(),
    wpart:fset("__edit", Not_validated),
    {template, "templates/users/create_users.html"};

% ..

%%
%% controller functions
%%
before_create(_) ->
    wpart:fset("error_message",""),
    {template, "templates/save_form.html"}.

create(X) ->
    wtype_example:create(X),
    {redirect, "/save_form.html"}.

```

Ready! *New_controller* calls *wtype* but that's the subject of next chapter. For now just imagine that it is there and works perfect. Function *before_create* is responsible for displaying form. It uses *example_records*. No HTML. Just define action under button in template. Validation is on framework side. It returns proper values or prepares error messages. In case of fail error/2 is called for function which failed. In this case *create*. On Error wrong fields are surrounded by frame (table border) - style in css (class *form_error*). Furthermore, field in form has those wrong values, so it is possible to make some changes. The same mechanism works for default values. Just call from model *prepare_default* instead of *prepare_initial*. To make it super simple function, *create* goes back to *save form*. Normally it would be some menu template.

6 Model. What happens in model it stays in model.

6.1 Overview

Model is a layer where all calls to data store are executed. Model file can be easily identified - it should be in application folder and its name should have prefix *wtype_*. It is consequence of mirroring types in records. In that way *wtype_* file has additional functionality.

6.2 Specification

Wtype_ has to include type records and exports number of functions.

<i>validate/1</i>	handling validation of complex type
<i>get_record_info/1</i>	information about type for framework side modules; must have 2 clauses
<i>get_parent_info/1</i>	only in case of sharing type definition. Returns name of type which records are used.

All the other functions are called by controller, helpers, wparts so name convention is not important (usually some of them are *create*, *read*, *update*, *delete* - to correlate with CRUD type of controller).

6.3 Example

6.3.1 Basics

Lets take a look at example of model for users controller, which is responsible for managing users accounts.

```
----- wtype_users.ert -----
1  -module(wtype_users).
2
3  -export([validate/1, get_record_info/1, get_parent_type/0]).
4  -export([create/1, delete/1, update/1, read/1]).
5  -export([prepare_initial/0]).
6
7  -include("../include/users_records.hrl").
8  -include_lib("stdlib/include/qlc.hrl").
9
10 get_record_info(users_types) -> #users_types{};
11 get_record_info(users) -> record_info(fields, users).
12
13 validate(From) ->
14     SuperField = get_record_info(users),
15     SuperType = get_record_info(users_types),
```

```

16
17     wpart_valid:validate(SuperField, SuperType,
18                         From ++ ["users"]).
19
20 create(User = #users{password = Password}) ->
21     Bool = is_available(User),
22     %% ..

```

Be careful with line [18]. Name we are appending should be the same as type.

6.3.2 Dynamic type changes

There are two ways of changing type dynamically. For example, if we want to send feed for autocomplete type instead of defining tuple {complete, "something1 | something2"} (see Ref. Manual). in type records, we should use request dictionary (and place it in controller).

```

----- controller.erl -----
wpart:fset("__types", [{account_id, {complete, wtype_users:get_accounts()}}]),

```

Wpart_derived recognises key *types* and extends field `account_id` with autocomplete proposals taken from database.

The second way is more important in chapter about model. Let's change type by modification of function `get_record_info/1`.

```

----- wtype_link.erl -----
1
2 %% ..all the necessary includes and exports
3
4 get_record_info(link_types) ->
5     C = #link_types{},
6     {_Name, Raw} = C#link_types.category,
7     Where = wpart:fget("__controller"),
8     #group{categories = Cat_ids} = wtype_group:read(atom_to_list(Where)),
9     Cat_str = lists:map(fun(X) ->
10                         #category{title = XX} = wtype_category:read(X),
11                         XX
12                     end, Cat_ids),
13     Zipped = lists:zip(Cat_ids, Cat_str),
14     ToGo = string:join(
15         lists:map(fun({X,Y}) ->
16                     integer_to_list(X) ++ ":" ++ Y
17                 end,
18                 Zipped),
19     "|"),

```

```

20     #link_types{category = {enum, Raw ++ [{choices, ToGo}]}];
21
22     get_record_info(link) -> record_info(fields, link).
23
24
25     validate(From) ->
26         SuperField = get_record_info(link),
27         SuperType = get_record_info(link_types),
28
29         wpart_valid:validate(SuperField, SuperType, From ++ ["link"]).
30
31     %% ..

```

What is done above - it is basically checking who called type Link (line [7]) and basing on this information enum type (HTML radio buttons) is given some options.

Even more important is that we can put there any code and change type as we like, e.g. authorization (session check).

7 View. Let's see it.

7.1 Overview

Final result of interaction with system is rendered on base of html files in *templates* directory.

Usually each controller has folder to keep its own files but this is not requirement. Paths for .html files are directly set in controller in every function.

7.2 Example

7.2.1 Base and dispatcher

Our task now is to build view part of application. Assume we want to have some menu on the horizontal bar and dynamic input in main part of page. In case of some pictures or JS scripts in docroot we need to change dispatcher. Let's extend previous example.

```
dispatcher.conf
{dynamic, "~/before_create", {new_controller, before_create}}.
{dynamic, "~/create", {new_controller, create}}.

{static, "~/images/*.*", enoent}.
{static, "~/style.css", enoent}.
```

Next part is to create base html file which will include some external files.

```
base.html
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml">
5
6 <head>
7   <title>erlangweb</title>
8   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
9   <link rel="stylesheet" href="/style.css" type="text/css" />
10 </head>
11
12 <body>
13 <div class="class1">
14   <div class="header">
15     <p id="ex_class1"><span>ErlangWeb</span></p>
16     <p id="ex_class2"><br/>ErlangWeb is cool<br/>
17     <wpart:menu />
18     <wpart:submenu />
19   </div>
```

```

20     <div class="class2">
21         <div class="content">
22             <wtpl:include name="content"/>
23         </div>
24     </div>
25 </div>
26 </body>
27 </html>

```

Now we can clearly see what is going on in base.html.

- Line 1 - necessary for xmerl.
- Line 8 - just example issue.. IE ignores line 1 so we need to place info in meta information.
- Line 9 - everything like in regular HTML, here goes links to JS libraries, rss, css.
- Lines 17,18 - static call to wparts. This case Menu.
- Line 22 - here is defined name of include slot.

7.2.2 Content

Now is time to construct file which is called in controller.

```

_____ create.html _____
1 <wtpl:parent path="templates/base.html">
2   <wtpl:content name="content">
3     <wpart:lookup key="error_message" />
4     <wpart:form type="link" action="/link/create" />
5   </wtpl:content>
6 </wtpl:parent>

```

- Line 1 - says where to look for parent template
- Line 2 - specifies which slot should be filled
- Line 3 - picking up some error messages
- Line 4 - expanding form

7.2.3 Listing

Templates mentioned in example have just functionality of displaying form. To finish application we need some listing of stored values. Usually we need to display values as a list of items. Even better, sometimes, there should be several lists representing groups of specific values.

Choice of implementation listing view is based on functionality of a model. The simplest case is when groups of items are known and cannot be changed.

Lets change controller and write some view. Remember to add proper URLs to *dispatcher*.

```
link.erl
1 read(_) ->
2     prepare_list(),
3     {template, "templates/link/list.html"}.
```

```
page.html
1 <wtpl:parent path="templates/base.html">
2   <wtpl:content name="content">
3
4   <wpart:list select="map" list="Sites" as="link">
5     <wpart:lookup key="link:title" />
6     <wpart:lookup key="link:description" />
7   </wpart:list>
8
9   <wpart:list select="map" list="Repositories" as="repo">
10    <wpart:lookup key="repo:title" />
11    <wpart:lookup key="repo:description" />
12  </wpart:list>
13
14  </wtpl:content>
15 </wtpl:parent>
```

To make things working just implement function *prepare_list* in controller. It should set values read from database into *request dictionary*.

Situation can get complicated if number, order, names and many other features is expected to be set dynamically. This exaple shows adding and listing links. We can choose group or set new one and change order of links. So there is necessary additional layer to handle all those features. Next example show how to treat this kind of problem.

One of simplest way of doing it is using *wpart_page*.

```
page.html
1 <wtpl:parent path="templates/base.html">
2   <wtpl:content name="content">
3     <wpart:page />
4   </wtpl:content>
5 </wtpl:parent>
```

wpart_page needs some information from controller.

```
link.erl
1 read(_) ->
2     Auth = filter:is_auth([editor]),
3     wpart:fset("__type", {link, whelp_link, Auth, "/link/before_create"}),
4     {template, "templates/crud.html"}.
```

In line 3 *whelp_link* is a module to call by *wpart_page*. Specification would be:

```
wpart:fset("__type", {Name, Module, Authorization, Path})
```

Name = Module = atom()

Authorization = Bool()

Path - valid URL of requested function

whelp_link module calls *wtype* and prepares data. It must return a list of lists to group data in separate list elements.

```
whelp_link.erl
1 %% ..
2
3 submenu(_) ->
4     ["/blog/year/", "BY YEAR"},
5     ["/blog/user/", "BY USER"},
6     ["/blog/tag/", "BY TAG"}].
7
8 list(Auth) ->
9     Blogs = wtype_blog:read(all),
10    {"Blogs",
11     [[whelper:prepare("templates/wparts/list_li.tpl",
12                      blog_lead(Auth, Art))
13      ||
14      Art <- Blogs]}.
15
16 main(Auth) ->
17     All = sort(wtype_blog:read(all)),
18     Blog = case catch hd(All) of
19         {'EXIT', _} ->
20             "No blog posts in database";
21         E1 ->
22             blog_lead(Auth, E1)
23     end,
24     {"Main Page", [whelper:prepare("templates/wparts/list_li.tpl", Blog)]}.
25
26 latest(Auth) ->
27     case sort(wtype_blog:read(all)) of
28         [First | _] ->
```

```
29         blog(Auth, First#article.id);
30     [] ->
31         {"Blog", "No blog posts in database"}
32     end.
33
34 %%..
```

Fuction used by *wpart page* is `list`. Each element returned is `` which is part of list constructed by *wpart page*. Easily every one can write own version of this kind of *wpart* – just more customized.

This is just one way of implementing *whelp* file. To understand it completely we should explain data relation – which is not important here. It is just to show case where general way of implementing view is not enough.

What is significant, we can see how flexible is framework. It is possible to change approche to solve many problems.

8 Reference Manual

8.1 Basic Types

8.1.1 Definition options

Type	Argument	Meaning
all types	{private, Bool()}	if Bool() is <i>true</i> , field is private - it will neither be checked during validation (always accepted) nor displayed in HTML form (even as hidden) Example - field will be private: <code>{private, true}</code>
	{optional, Default-Value}	field will be optional - it will be set to its default value in case of not entering anything by user in HTML form Example - field will be optional with default value set to integer 10: <code>{optional, 10}</code>
	{description, Desc}	<i>Desc</i> will be the description displayed in the generated form. <i>Desc</i> could be either a string (and it will be a description itself) or tuple {key, Key} - in this situation description will be translated with the key <i>Key</i> Example - field will be described as "Your name:": <code>{description, "Your name:"}</code> Example - field will be described with translation of the key <i>user:login</i> : <code>{description, {key, "user:login"}}</code>

integer	<p>{min, Int()}</p> <p>{max, Int()}</p>	<p>checks if integer is greater or equal to the specified value. If not, {error, {smaller_than_min, Int()}} is returned Example - integer must be greater than 2:</p> <p><code>{min, 3}</code></p> <p>checks if integer is smaller or equal to the specified value. If not, {error, {greater_than_max, Int()}} is returned Example - integer must be smaller than 1000:</p> <p><code>{max, 999}</code></p>
float	<p>{min, Float()}</p> <p>{max, Float()}</p>	<p>checks if float is greater or equal to the specified value. If not, {error, {smaller_than_min, Float()}} is returned Example - float must be greater then 3.14:</p> <p><code>{min, 3.14}</code></p> <p>checks if float is smaller or equal to the specified value. If not, {error, {greater_than_max, Float()}} is returned Example - float must be smaller than 256.23:</p> <p><code>{max, 256.23}</code></p>
string, text	<p>{min_length, Int()}</p>	<p>checks if string is longer than or equal length as specified value. If not, {error, {too_short, Str()}} is returned. Length of the string is measured by utf8_api:ulelength/1 function call Example - string must be at least 6 characters long:</p> <p><code>{min_length, 6}</code></p>

	{max_length, Int() }	<p>checks if string is shorter than or equal length as specified value. If not, {error, {too_long, Str()}} is returned. Length of the string is measured by utf8_api:ulength/1 function call</p> <p>Example - string must be at most 20 characters long:</p> <pre>{max_length, 20}</pre>
	{html, Whitelist }	<p>check string for presence of valid XHTML tags. Only those tags, which are specified in whitelist are accepted.</p> <p>The following errors ({error, Reason} where Reason is one of the element below) are returned:</p> <p>{tags_not_closed, List} - user did not close all tags properly. Tags which are not closed are returned.</p> <p>{tag_not_in_whitelist, Tag} - user entered a tag, which is not in whitelist. Blacklisted tag is returned.</p> <p>{closing_bad_tag, ClosingTag, OpenedTag} - user closed tag, which has not been opened most recently. Tag closed by user and the one, which should be closed are returned.</p> <p>{no_closing_tag, Tag} - string has ended but tag remains opened (like "this is a link: ja").</p> <p>open_tag_inside_tag - user open tag inside another tag.</p> <p>{no_open_quote, Tag} - user did not enquoted all attribute values properly.</p> <p>open_tag_inside_attr - user opened next tag inside of the attribute value.</p>

		<p>Example - user is allowed to enter only <i>br</i>, <i>u</i>, <i>i</i> tags:</p> <pre>{html, ["br", "u", "i"]}</pre>
date	<pre>{format, Format}</pre> <pre>{min, MinDate}</pre> <pre>{max, MaxDate}</pre>	<p>specifies the format of entered date. By default format is "YYYY-MM-DD". The accepted separators are "-", "/", " ", "." and "_". If there is a bad separator entered, <code>{error, {bad_separator_in_date_form, Date}}</code> is returned. In case of entering bad input, <code>{error, {bad_date_format, Date}}</code> is returned.</p> <p>Example - date must be in format "MM/DD/YYYY":</p> <pre>{format, "MM/DD/YYYY"}</pre> <p>checks if entered date is later than specified. If not <code>{error, {bad_range, Date}}</code> is returned.</p> <p>Example - date must be later than 01/01/1970:</p> <pre>{min, "01/01/1970"}</pre> <p>checks if entered date is earlier than specified. If not <code>{error, {bad_range, Date}}</code> is returned.</p> <p>Example - date must be earlier than 06-12-2010:</p> <pre>{max, "06-12-2010"}</pre>
password	<pre>{min_length, Min},</pre> <pre>{max_length, Max}</pre>	the same as in <i>string</i> , <i>text</i>
atom	<i>none</i>	entered value atom representation must exist in the system
enum	<pre>{choices, Choices}</pre>	specifies the possible user can choose. <i>Choices</i> is string with elements separated with " " character
bool	<pre>{always, Bool()}</pre>	checks if user's input is set to the given value. If not <code>{error, {bad_bool_value, Val}}</code> is returned. <p>Example - user have to enter false:</p> <pre>{always, false}</pre>

multilist	{option, Options}	specifies the options which could be selected from list. <i>Options</i> is a string with elements separated with " " character. Each element is of format value:description, e.g. "opt1:desc1 opt2:desc2".
upload	<i>none</i>	no additional options
time	{format, Format} {min, MinTime}, {max, MaxTime}	specifies the format of entered time. By default format is "HH:MM:SS". The only accepted separator is ":". If there is a bad separator entered, {error, {bad_separator_in_time_form, Time}} is returned. In case of entering bad input, {error, {bad_time_format, Time}} is returned. Example - time must be in format "SS:MM:HH": {format, "SS:MM:HH"} the same as in <i>date</i>
autocomplete	{min_length, Min}, {max_length, Max}, {html, Whitelist} {complete, List}	the same as in <i>string, text</i> lists the all possible autocompletion list. <i>List</i> should be a string with elements separated with " "
datetime	{format, Format} {max, MaxDateTime} {min, MinDateTime}	Example format "YYYY-MM-DD HH:MM:SS" Options are routed to type date and time validators – so all acceptable date and time formats joined with space are good as well Error codes: <i>bad_date</i> , <i>bad_time</i> – validation of time or date failed; <i>bad_input</i> – returned in case of wrongly formatted input

csv	{type, Type}	<p>sets the type of the elements of entered data - all other options will be related to the pointed type. If there is an error in validation of the basic type element, {<i>error</i>, {<i>wrong_value_in_set</i>, <i>CSV</i>}} is returned.</p> <p>Example - csv of type integer:</p> <pre>{type, integer}</pre>
-----	--------------	---

8.1.2 HTML tags

During HTML form build process, each element of the record (except private ones) is substituted with its corresponding HTML tag:

basic type	HTML tag
integer	<input type="text" />
float	<input type="text" />
string	<input type="text" />
date	<input type="text" />
bool	<input type="checkbox" />
enum	<input type="radio" />
text	<textarea>...</textarea>
upload	<input type="file" />
password	<input type="password" />
atom	<input type="text" />
multilist	<select> <option>...</option> ... </select>
time	<input type="text" />
datetime	<input type="text" />
autocomplete	<input type="text" /> (with JavaScript)
csv	<input type="text" />

8.2 Interesting wparts

8.2.1 wpart_choose

wpart_choose provides *if* functionality in views.

During the tag expanding, wpart looks for *wpart:whens* inside its body (and one optional *wpart:otherwise*. First *wpart:when* body which test evaluates to *true* is inserted as a *wpart:choose* value. Otherwise, *wpart:otherwise* is inserted.

Because XHTML standard does not allow to use `<` character in attribute value, we can use some substitutions of comparison operators:

- eq → `==`
- neq → `!=`
- lt → `<`
- le → `<=`
- gt → `>`
- ge → `>=`

Attributes

test - Erlang code, which will be evaluated and the result will be compared to *true*

Tags

wpart:when - one of the *if*'s branch - the *wpart:when*'s test attributes are evaluated sequentially. If its test has been evaluated to *true* as first, its body is inserted as a *wpart:choose* value

wpart:otherwise - in case of all *whens* failure, its body is inserted as a return of *wpart:choose* expanding

Example

```
choose.html
1  ...
2  <wpart:choose>
3    <wpart:when test="1 + 1 == 3">
4      Impossible!
5    </wpart:when>
6    <wpart:when test="random:uniform(2) == 2">
7      Sometimes...
```

```

8     </wpart:when>
9     <wpart:otherwise>
10        Last resort!
11    </wpart:otherwise>
12 </wpart:choose>
13 ...

```

8.2.2 wpart_include

wpart_include allows to inject content of other file into view.

Attributes

file - specifies the path of the included file

as - if present - inserts the content of the file into the *as* dictionary key

format - if present - a format to apply to the value before printing it out.

Example

We will insert contents of the file *priv/include/licence.txt* into view:

```

_____ licence.html _____
1  ...
2  <h3>Here is out licence: </h3>
3  <pre>
4    <wpart:include file="priv/include/licence.txt" />
5  </pre>
6  <wpart:include file="priv/include/date.txt" format="YYYY-MM-DD" />
7  ...

```

Here is the example, how to use *as* attribute:

```

_____ copyright.html _____
1  ...
2  <wpart:include file="priv/include/copyright.sig" as="copyright" />
3  This file is <wpart:lookup key="copyright" />.
4  ...
5  And here is another place where we are using wpart:lookup:
6  <wpart:lookup key="copyright" />
7  ...

```

8.2.3 wpart_lang

wpart_lang creates the possibility of building multilingual services with the same view structures. More information could be found in *Internationalization* section.

Attributes

key - specifies the key that translation text should be fetched from.

Example

Here is the example explaining how to translate part of the page:

```
main.html
1  ...
2  <wpart:lang key="welcome" />
3  ...
4  <a href="login"><wpart:lang key="login:login" /></a>
5  <a href="register"><wpart:lang key="login:register" /></a>
6  ...
```

8.2.4 wpart_list

wpart_list provides the list search/traversal functionality.

Attributes

select - type of the operation. Can be *map*, *head*, *tail*, *filter*, *find* or *sort*:

map - does a for each on the body, i.e. renders the body for all element in the list

head - renders the body for the first element in the list

tail - renders the body for the last element in the list

filter - filters the list according to the *pred* attribute

find - returns the first element of the list for which the *pred* attribute evaluates to *true*

sort - sorts the list according to the *pred* attribute

as - inserts the content of the evaluation into the *as* dictionary key

list - specifies which list in dictionary will be traversed

pred - Erlang function definition

Example

1. Print out all the elements of the list:

```
_____ map.erl _____
1  ...
2  <ul>
3    <wpart:list select="map" list="names" as="name">
4      <li><wpart:lookup key="name" /></li>
5    </wpart:list>
6  </ul>
7  ...
```

2. Print out first element of the list:

```
_____ head.erl _____
1  ...
2    <wpart:list select="head" list="line" as="first">
3      First in line: <wpart:lookup key="first" />
4    </wpart:list>
5  ...
```

3. Access the last element of the list:

```
_____ tail.erl _____
1  ...
2  <wpart:list select="tail" list="snake" as="piece">
3    <wpart:choose>
4      <wpart:when test="{piece} == rattle">
5        Aargh! It's a rattle snake! Run!
6      </wpart:when>
7      <wpart:otherwise>
8        Ooh! Come here, cute snake!
9      </wpart:otherwise>
10   </wpart:choose>
11 </wpart:list>
12 ...
```

4. Filter some elements:

```
_____ filter.erl _____
1  ...
2  <wpart:list select="filter" list="employees" as="cool_employees"
3    pred="fun(E) -> element(3, E) == cool end">
4    The cool ones are:
5    <wpart:list select="map" list="cool_employees" as="ce">
6      <wpart:lookup key="ce" />,
```

```

7   </wpart:list>
8 </wpart:list>
9   ...

```

5. Find elements of the list:

```

                                find.erl
1   ...
2   <wpart:list select="find" list="everyone" as="the_chosen_one"
3       pred="fun(E) -> E == chosen end">
4       <wpart:choose>
5         <wpart:when test="{the_chosen_one} == []">
6           We are all equal.
7         </wpart:when>
8         <wpart:otherwise>
9           Go cry, emo kid!
10        </wpart:otherwise>
11       </wpart:choose>
12    </wpart:list>
13    ...

```

6. Sort the list:

```

                                find.erl
1   ...
2   <wpart:list select="sort" list="movies"
3       as="top_rated_movies" pred="fun(M1, M2) -> M1 > M2 end.">
4       The movie I liked most ever is
5       <wpart:list select="head" list="top_rated_movies" as="best_movie">
6         <wpart:lookup key="best_movie" />
7       </wpart:list>
8    </wpart:list>
9    ...

```

The strings inside the { } will be used as the keys to the request dictionary: their corresponding values will be inserted in their places.

8.2.5 wpart_lookup

wpart_lookup grants access to previously set variables inside a view.

Variables should be prepared by calling *wpart:fset* function.

Attributes

key - specifies the key which the value should be fetched from.

Example

```
controller.erl
1  ...
2  print_date() ->
3      wpart:fset("date", tuple_to_list(date())),
4      {template, "templates/date.html"}.
5  ...
```

```
date.html
1  ...
2      Today is <wpart:lookup key="date" />!
3  ...
```

8.2.6 wpart_paginate

wpart_paginate makes the possibility of chunking the big collection of items into smaller parts and access them sequentially in the view.

Attributes

action - defines the returning value the expanded tag.

page - chunking the list into pieces and give access to its small part.

next_link - expands to the link to the next part of the list

prev_link - expands to the link to the previous part of the list

list - checked only when *action == page*, paginates the list held under given key in e_dict (it must be manually set in the controller)

as - checked only when *action == page*, saves the part of list we are interested in under the given key, so it is possible to fetch the contents of the list inside the wpart:paginate tag

per_page - checked only when *action == page*, defines the maximum length of the chunk (list) we pass to the inside of the tag

text - checked only when *action == next_page* or *prev_page*, defines the text which should be displayed as the clickable link

Example

```
controller.erl
1  ...
2  search(Keyword) ->
3      ...
4      wpart:fset("search_results", Results),
```

```
5 {template, "templates/search.html"}.
```

```
6 ...
```

search.html

```
1 ...
```

```
2 <wpart:paginate action="page" as="search_results_part"
```

```
3   list="search_results" per_page="10">
```

```
4   <ul>
```

```
5     <wpart:list select="map" as="item" list="search_results_part">
```

```
6     <li><wpart:lookup key="item" /></li>
```

```
7     </wpart:list>
```

```
8   </ul>
```

```
9 </wpart:paginate>
```

```
10 <wpart:paginate action="next_link" text="Previous page" /> ||
```

```
11 <wpart:paginate action="prev_link" text="Next page" />
```

```
12 ...
```

8.3 Tuples for HTTP server

8.3.1 Overview

Each time our controller end processing some request, it must return a tuple, which tells server what to do. We need some kind of protocol to talk to server to serve proper content or handle HTTP codes (200, 404, 501, etc.).

In order to tell server what to do, we need to return a situation-specific tuple from each accessible controller method.

8.3.2 Types of tuples

{redirect, URL} - HTTP code will be set to 302, user will be redirected to *URL*

{content, html, Data} - *Data* is a valid XHTML string, which will be served to the browser. MIME type is set to "text/html"

{content, text, Data} - *Data* is a plain text string, which will be served to the browser. MIME type is set to "text/plain"

{json, Data} - *Data* will be encoded to JSON format and sent to the browser. MIME type is set to "text/plain"

{template, Template} - selected *Template* will be expanded as a response to the request. MIME type is set to "text/html"

template - when skipping dispatcher, the template specified in URL will be expanded

{custom, Custom} - *Custom* is passed to the server. This option is for all the types the framework doesn't handle, but server does

{error, Code} - error with code *Code* is generated, template responsible for this kind of error (defined in *errors.conf*) is expanded

{headers, Headers, RetVal} - sets the given headers in the session. *Headers* is a list of tuples describing which headers should be set. Currently only setting cookies is supported. The valid formats are:

- {cookies, CookieName, CookieValue} - sets the cookie with the name *CookieName* to the value *CookieValue*. The expiration date is set to the end of the session, path is set to /.
- {cookies, CookieName, CookieValue, CookiePath} - does the same as above, but set path to *CookiePath*.
- {cookies, CookieName, CookieValue, CookiePath, CookieExpDate} - does the same as above, but set the expiration date to *CookieExpDate*. The date should be in right format: DAY, DD-MMM-YYYY HH:MM:SS GMT

RetVal should be the actual returning value - one of the tuples mentioned above (like {redirect, URL} or other).

9 Credits

All suggestions and corrections are welcome!

Developed by *Erlang Training & Consulting Team*
Email: erlangweb@erlang-consulting.com

If you have any questions to the tutorial:
Michal Ptaszek: michal.ptaszek@erlang-consulting.com
Michal Zajda: michal.zajda@erlang-consulting.com