Emacs support for Erlang 🚧

Description	Keystroke	Function	Note
Support for the Erlang Programming Language	📦Emacs provides support for Erlang and Erlang Tools via the erlang.el external package (see erlang.el source) and some other packages. PEL activates Erlang support via the customize user option variable **pel-use-erlang**. It must be set to **t** to activate support for Erlang. Further customization is available via several user options. • PEL customization for Erlang: use the command below: **pel-cfg-pkg-erlang**. • pel-erlang-rootdir: • pel-erlang-exec-path: • **pel-erlang-shell-prevent-echo:** set to **t** to prevent the Erlang shell from echoing every command. • PEL Erlang Source Code Style: **pel-erlang-code-style**: • **pel-erlang-fill-column** : column where line-wrapping occurs : maximum line length (defaults to 100). • **pel-erlang-skel-use-separators**: whether line separators are used in Erlang code templates (see the *Insert Erlang Code Template* section below), • **pel-erlang-skel-use-secondary-separators** : whether secondary separator lines are inserted by some Erlang code templates, • **pel-erlang-skel-insert-file-timestamp**: whether automatically updated time stamps are inserted in Erlang source code file header blocks. • PEL provides the following set of mode-specific key prefixes: **`<f11> SPC e`**, **`<f12>`** and **`<M-f12>`** The first one is always available. The other two prefixes are only available when the current buffer is in erlang-mode. The **`<M-f12>`** prefix helps the typing flow when the next key is a Meta key. For simplification, the **`<f11> SPC e`** prefix is normally omitted in the table.		
Erlang Mode version	`<f12> ?`	(**erlang-version**)	Display the current version of **erlang.el** in the mini-buffer.
Customize PEL Erlang Support (See also: ∑ Customize)	• `<f11> <f1> SPC e` • `<f12> <f1>`	(**pel-cfg-pkg-erlang** &optional OTHER-WINDOW)	Customize PEL Erlang support. • If OTHER-WINDOW is non-nil (use **`C-u`**), display in another window and open Erlang related customization groups as well, each one side its own window. ⚠️ If Emacs available screen size in the current frame is too small, several windows may not be visible, however the buffers will be opened. • The **`<f12> <f1>`** binding is available when point is in a buffer visiting a Erlang file.
Syntax Highlighting	Erlang code syntax highlighting has 4 levels and can be turned off. Use the Erlang menu : **`<f10>`** to access the menu, then select Erlang and then Syntax Highlighting.		
Edit Erlang Code	The following commands help edit Erlang code.		
Create additional clause	`C-c C-j`	(**erlang-generate-new-clause**)	Create additional Erlang clause header. • Parses the source file for the name of the current Erlang function. Create the header containing the name, a pair of parentheses, and an arrow. The space between the function name and the first parenthesis is preserved. The point is placed between the parentheses.
Clone clause arguments	`C-c C-y`	(**erlang-clone-arguments**)	Insert, at the point, the argument list of the previous clause. • Copy the function arguments of the preceding Erlang clause. This command is useful when defining a new clause with almost the same argument as the preceding. • The mark is set at the beginning of the inserted text, the point at the end.

Align arrows inside region	`C-c C-a`	(**erlang-align-arrows** START END)	Align arrows ("->") in function clauses inside marked region or in the current function.
			• With a prefix argument, aligns all arrows in the region (or from beginning of buffer up to point), not just those in function clauses.
			• Example:
			`sum(L) -> sum(L, 0).`
			`sum([H|T], Sum) -> sum(T, Sum + H);`
			`sum([], Sum) -> Sum.`
			becomes:
			`sum(L)           -> sum(L, 0).`
			`sum([H|T], Sum) -> sum(T, Sum + H);`
			`sum([], Sum)     -> Sum.`

Electric Keys	The following keys have "*electric*" behaviour and perform special editing tasks to help edit Erlang source code.		
Electric comma	`,`	(**erlang-electric-comma** &optional ARG)	Insert a comma character and possibly a new indented line. • The variable 'erlang-electric-comma-criteria' states a criterion, when fulfilled a newline is inserted and the next line is indented. • Behaves just like the normal comma when supplied with a numerical arg, point is inside string or comment, or when there are non-whitespace characters following the point on the current line.
Electric semicolon	`;`	(**erlang-electric-semicolon** &optional ARG)	Insert a semicolon character and possibly a prototype for the next line. • The variable 'erlang-electric-semicolon-criteria' states a criterion, when fulfilled a newline is inserted, the next line is indented and a prototype for the next line is inserted. Normally the prototype consists of " ->". Should the semicolon end the clause a new clause • header is generated. • The variable 'erlang-electric-semicolon-insert-blank-lines' controls the number of blank lines inserted between the current line and new function header. • Behaves just like the normal semicolon when supplied with a numerical arg, point is inside string or comment, or when there are non-whitespace characters following the point on the current line.
Electric > (for the end of arrow)	`>`	(**erlang-electric-gt** &optional ARG)	Insert a greater-than sign, and optionally insert a new line and indent.
Erlang Comments	Erlang uses the % character to identify line comments. It uses the following conventions: • **%** - Single percent characters for comments located toward the end of a line of code • **%%** - Two percent characters are used for comments starting at indentation level. • **%%%** - Three percent characters are used to describe modules and are always placed in the first column		
Comment/un-comment Note: **`M-;`** works much better than **`C-c C-c`** and **`C-c C-u`**	`M-;`	(**comment-dwim** ARG)	Comment line or region with **%** or **%%** style comments depending on the location in the buffer. • When no marked region and no comment: • On empty line: insert **%%** comment starter at the proper indentation level. • On line with code: insert **%** comment starter after the code for an end-of-line comment • With marked un-commented region: • Comment region (each line is commented) • With marked commented region: • removes the comment. • To insert **%%%** comment style: type **`M-3 M-;`** • Call the comment command you want (Do What I Mean). • If the region is active and 'transient-mark-mode' is on, call 'comment-region' (unless it only consists of comments, in which case it calls 'uncomment-region'). Else, if the current line is empty, call 'comment-insert-comment-function' if it is defined, otherwise insert a comment and indent it. Else if a prefix ARG is specified, call 'comment-kill'. Else, call 'comment-indent'. ✂️ The erlang.el code binds **`M-1`** to indent-for-comment. However PEL uses **`M-1`** for something else. The **`M-;`** binding to comment-dim works just as indent-for-comment if nothing is marked.

Description	Keystroke	Function	Note
	`C-c C-c`	(**comment-region** BEG END &optional ARG)	Comment or uncomment each line in the region. • With just **`C-u`** prefix arg, uncomment each line in region BEG .. END. • Numeric prefix ARG means use ARG comment characters. • If ARG is negative, delete that many comment characters instead. • The strings used as comment starts are built from '**comment-start**' and '**comment-padding**'; the strings used as comment ends are built from '**comment-end**' and 'comment-padding'. • By default, the '**comment-start**' markers are inserted at the current indentation of the region, and comments are terminated on each line (even for syntaxes in which newline does not end the comment and blank lines do not get comments). This can be changed with '**comment-style**'.
Un-comment region	`C-c C-u`	(**uncomment-region** BEG END &optional ARG)	Uncomment each line in the BEG .. END region. The numeric prefix ARG can specify a number of chars to remove from the comment delimiters.
Fill current paragraph (See also: ∑ Filling/ Justification)	• **`M-q`** • **`<f11> t f p`**	(**fill-paragraph** &optional JUSTIFY REGION)	Fill multi-line comment at or after point. • To justify as well: **`C-u M-q`** • In refill mode this is done automatically. In auto fill mode the filling is done at the end of the line. • See the ∑ Filling/Justification for all filling and justification commands.
Toggle display of comments in buffer or active region (See: ∑ Comments)	**`<f11> ; ;`**	(**hide/show-comments-toggle** &optional START END)	Toggle hiding/showing of comments in the active region or whole buffer. • If the region is active then toggle in the region. Otherwise, in the whole buffer. 📦 This requires the **hide-comnt.el** package (see ∑Comments). ⤴ PEL activates it when the **pel-use-hide-comnt** user option is **t**.
Indentation	All syntactic indentation control for D is controlled by the CC-Mode logic and provided commands listed below. • Rigid indentation commands are also available and listed at the end of this list. They are also listed in the ∑ Indentation table.		
Indent current line or region (See also: ∑ Indentation)	**`<tab>`**	(**c-indent-line-or-region** &optional ARG REGION)	Indent active region, current line, or block starting on this line. • Behaviour depends on syntactic-indentation mode (enabled by default but can be toggled on/off with the **`<f12> M-i`** key): • With syntactic-indentation on (the default): • In Transient Mark mode, when the region is active, reindent the region. • Otherwise, with a prefix argument, rigidly reindent the expression starting on the current line. • Otherwise reindent just the current line. ☝This might seem strange for new Emacs users, but it ends up being very useful. You can type **`<tab>`** anywhere in the line to adjust the indentation of the current line or everything in the marked area if a block is marked. • With syntactic-indentation off: • <tab> always indent current line by one level • C-u - <tab> or M- <tab> always un-indent current line by one level • Indenting marked region is done without syntax knowledge and at the same level as previous line. ☝ If you want to indent rigidly you can use: • (**pel-indent-rigidly &optional N**) (bound to **`C-x <tab>`** and to **`<f11> <tab><tab>`**) to indent the line or region rigidly. • (**tab-to-tab-stop**), bound to **`M-i`** to insert spaces to the next tab stop column.
Indent Erlang function	**`C-c C-q`**	(**erlang-indent-function**)	Indent current Erlang function. ☝This also works with a simple tab (see above).
Indent lines of list after point (See also: ∑ Indentation)	**`C-M-q`**	(**prog-indent-sexp** &optional DEFUN)	Indent the expression after point. When interactively called with prefix, indent the enclosing defun instead.
Indent a region	**`C-M-\`**	(**indent-region** START END &optional COLUMN)	Indent each nonblank line in the region. • A numeric prefix argument specifies a column: indent each line to that column. • With no prefix argument, the command chooses one of these methods and indents all the lines with it: 1. If 'fill-prefix' is non-nil, insert 'fill-prefix' at the beginning of each line in the region that does not already begin with it. 2. If 'indent-region-function' is non-nil, call that function to indent the region. 3. Indent each line via 'indent-according-to-mode'. ☝When a region is marked you can also use the simple **`<tab>`** to do the same when syntactic-indentation is active.
Navigation in Erlang code (See also: ∑ Navigation)	The erlang-mode provides commands to navigate across Erlang source code. PEL complements these. And EDTS also Several commands are specialization of the normal navigation commands which are described in the table ∑ Navigation, but several are specific to Erlang: • Notice the 3 sets of commands: 1. **`<f12> <up>`** and **`<f12> <down>`** move to the beginning of Erlang functions skipping all compiler directives. 2. The standard navigation commands, (mapped to **`<f6>`** prefix) move to beginning/end of Erlang functions but stop at compiler directives. 3. The **`<f12> <M-cursor>`** commands (also accessible via **`<M-f12> <M-cursor>`**, move across Erlang clauses (as opposed to functions). The list below describe the specialized commands only. See the others inside ∑ Navigation, like the navigation by blocks.		
Go to beginning of statement	**`M-a`**	(**backward-sentence** &optional ARG)	Go backward to the beginning of an Erlang clause. • With a numerical argument repeat that many times.
Go to the end of statement	**`M-e`**	(**forward-sentence** &optional ARG)	Go forward to the end of an Erlang clause. • With a numerical argument repeat that many times.
Go to beginning of current function or top-level function	**`C-M-a`**	(**c-beginning-of-defun** &optional ARG)	Move backward to the beginning of an Erlang function. • Every top level declaration that contains a brace paren block is considered to be a defun. • With a positive argument, move backward that many defuns. A negative argument -N means move forward to the Nth following beginning.
Goto end of current function or top-level function	**`C-M-e`**	(**c-end-of-defun** &optional ARG)	Move forward to the end of an Erlang function. • With argument, do it that many times. Negative argument -N means move back to Nth preceding end.
Move backward to beginning of previous function	• **`<f12> <up>`** • **`<f12> f p`**	(**pel-previous-erl-function** &optional N)	Move backward to the beginning of the previous function skipping all compiler directives. • With prefix argument N repeat N times. • Pushes mark; move back to previous position with **`M-``**. ☛Shift marking is available.
Move forward to beginning of next function	• **`<f12> <down>`** • **`<f12> f n`**	(**pel-next-erl-function** &optional N)	Move forward to the beginning of the next function skipping all compiler directives. • With prefix argument N repeat N times. • Pushes mark; move back to previous position with **`M-``**. ☛Shift marking is available.
Backward to beginning of function or compiler directive	• **`C-M-a`** • **`C-M-<home>`** • **`<f6> p`** • **`<f6> <up>`** • **`<f12> f P`**	(**beginning-of-defun** &optional ARG) (**erlang-beginning-of-function** &optional ARG)	Move backward to the beginning of an Erlang function or compiler directive. • With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun. ☛Shift marking is available in graphics mode, not in terminal mode (for **`C-M-a`** and **`C-M-<home>`**). However **`<f6> p`** and **`<f6> <up>`** handle Shift-marking fine in terminal mode. 🐞Erlang.el man page indicates an invalid mapping for this.

Description	Keystroke	Function	Note
Forward to beginning of next function or compiler directive	• `<f6> n` • `<f6> <down>` • `<f12> f N`	(**pel-beginning-of-next-defun** &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next function definition or compiler directive. • Beeps if does not find beginning of next function unless SILENT is non-nil. • If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with `M-``. ☛Shift marking is available for the `<f6>` bindings. ☝This command complements what end-of-defun does. • It moves forward but not to the end of the function definition (like end-of-defun) but to the beginning of the function definition, which is often what users of other editors expect. • It handles nested functions or class methods in languages like Python and others.
Backward to end of previous function or compiler directive	`<f6> <left>`	(**pel-end-of-previous-defun** &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous function definition. • Beeps if does not find end of previous function unless SILENT is non-nil. • If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with `M-``. ☛Shift marking is available for the `<f6>` bindings. ☝This command complements this set of 4 commands.
Forward to end of function or compiler directive	• `C-M-e` • `C-M-<end>` • `<f6> <right>`	(**end-of-defun** &optional ARG) (**erlang-end-of-function** &optional ARG)	Move forward to end of Erlang function. With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun. ☛ Shift marking is available in graphics mode, not in terminal mode (for `C-M-e` and `C-M-<end>`). However `<f6> <right>` handle Shift-marking fine in terminal mode.
Backward to beginning of clause	• `C-c M-a` • `<f12> c a` • `<M-f12> <M-up>`	(**erlang-beginning-of-clause** &optional ARG)	Move backward to previous start of clause. • With argument, do this that many times. 🐞Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314.
Forward to beginning of next clause	• `<f12> c n` • `<M-f12> <M-down>`	(**pel-beginning-of-next-clause**)	Move forward to the beginning of next clause. • Pushes mark; move back to previous position with `M-``. ☛Shift marking is available.
Backward to end of previous clause	• `<f12> c p` • `<M-f12> <M-left>`	(**pel-end-of-previous-clause**)	Move backward to the end of the previous clause. • Pushes mark; move back to previous position with `M-``. ☛Shift marking is available.
Forward to end of current clause	• `C-c M-e` • `<f12> c e` • `<M-f12> <M-right>`	(**erlang-end-of-clause** &optional ARG)	Move to the end of the current clause. • With argument, do this that many times. 🐞Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314.
EDTS/Navigation	EDTS (see below) provides the following commands to move point across Erlang functions. These do not support repetition prefix argument nor they support shift marking. There are other commands and key bindings to move across Erlang functions, and PEL support functions that perform the same and support repetition and shift marking. See the commands listed in the navigation section above.		
Move backward to beginning of previous function	`C-c C-d C-b`	(**ferl-goto-previous-function**)	Move backward to the beginning of the previous function skipping all compiler directives. ☝ PEL provides a more complete command to move across functions (with or without skipping directives) that push mark and support shift marking. See in the navigation section above.
Move forward to beginning of next function	`C-c C-d C-f`	(**ferl-goto-next-function**)	Move forward to the beginning of the next function skipping all compiler directives. ☝ PEL provides a more complete command to move across functions (with or without skipping directives) that push mark and support shift marking. See in the navigation section above.
Search Support	In Erlang mode, the superword mode can be useful since snake_case is often used. Using superword-mode helps searching.		
Toggle superword-mode (See also: ∑ Text Modes, ∑ Search/Replace)	• `<f11> t m p` • `<f12> M-p`	(**superword-mode** &optional ARG)	Toggle superword-mode: a minor mode that treats snake_case as one word. In Erlang, '_' are treated as part of words. • With a prefix argument ARG, enable superword mode if ARG is positive, and disable it otherwise. • PEL provides the `<f12> M-p` key for the programming language modes where snake_case is popular (Emacs Lisp, C, C++, Erlang, Python, etc…)
Marking	The following Erlang-mode specific marking functions are available. They complement what is already available and described in the ∑ Marking table. For those 2 commands the 🐞Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314.		
Mark Erlang function	• `C-M-h` • `<f12> f m`	(**mark-defun** &optional ARG) (**erlang-mark-function** &optional ARG)	Put mark at end of this function, point at beginning. • The function marked is the one that contains point or follows point. • With positive ARG, mark this and that many next functions; with negative ARG, change the direction of marking. • If the mark is active, it marks the next or previous function(s) after the one(s) already marked.
Mark Erlang Clause	• `C-c M-h` • `<f12> c m`	(**erlang-mark-clause**)	Put mark at end of clause, point at beginning.
Highlighting blocks	The following commands can be used to activate or toggle useful modes to highlight blocks of (), {}, and []. • show-paren-mode, which highlights the parens that matches the one before or after point. • rainbow delimiters mode, where matching nested parens are highlighted with the same colour.		
Toggle show-paren mode on/off (see also: ∑ Highlight)	• `<f12> M-9` • `<M-f12> M-9` • `<f11> b h (`	(**show-paren-mode** &optional ARG)	Toggle visualization of matching parens (Show Paren mode). • With a prefix argument ARG, enable Show Paren mode if ARG is positive, and disable it otherwise. • Show Paren mode is a global minor mode. When enabled, any matching parenthesis is highlighted in 'show-paren-style' after 'show-paren-delay' seconds of Emacs idle time.
Enable/Disable coloured highlight of nested blocks (),{},[] (see also: ∑ Highlight)	• `<f12> M-r` • `<M-f12> M-r` • `<f11> b h R`	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with different colours according to their depth. • Customize the depth and colours with **M-x customize-group rainbow-delimiters** 📦 **Requires:** rainbow-delimiters.el ☑ PEL activates this when the **pel-use-rainbow-delimiters** customize variable is set to **t**.
Inserting code			
Insert Parentheses	`M-(`	(**insert-parentheses** &optional ARG)	For Erlang: insert a parenthesis pair '()', leaving point after open-paren. • A positive ARG encloses the following ARG sexps in parenthesis if they are balanced. • A negative ARG encloses the preceding ARG sexps instead. • No argument is equivalent to zero: just insert '()' and leave point between. • PEL makes 'parens-require-spaces' buffer local and set it to nil in Erlang mode buffers, allowing the use of this command to insert the argument parentheses following a function (and without placing a space between the function name and the opening parenthesis. • If region is active, insert enclosing characters at region boundaries. • This command assumes point is not in a string or comment.

Description	Keystroke	Function	Note
Insert Erlang Code Templates (See also: ∑ Inserting Text for more info and information about tempo skeleton and yasnippet template-based text insertion).	The **erlang.el** external package defines a set of text skeletons using the standard tempo skeleton package. • The erlang package make these skeletons available on the Erlang/Skeletons menu (via `<f10>`). • PEL provides the following additional functionality: • Quick access keys to insert the templates, all mapped under the **pel:erlang-skel** key prefix: `<f12> <f12>`. • Several additional templates. These are marked with a **+**. These are also added to the menu. • Several aspects of the PEL Erlang Source Code Style is controlled by the user options inside the **pel-erlang-code-style** group. The controlled templates affected are marked with a **C**. The relevant user options are part of the **pel-erlang-code-style** group accessible with <f12> <f1> from an erlang mode buffer and include the following options: • **pel-erlang-skel-insert-file-timestamp** : set whether an automatically updated timestamp is inserted in the file header block. • **pel-erlang-skel-prompt-for-purpose** : set whether file and function skeletons blocks prompt for purpose and insert it. • **pel-erlang-skel-prompt-for-function-name** : set whether function skeletons prompt for function name and then inserts that name. • **pel-erlang-skel-prompt-for-function-arguments** : set whether function skeletons prompt for function arguments and then insert them. • **pel-erlang-use-separators** : set whether blocks use horizontal separator lines (these are the first of potentially 2 separators). • **pel-erlang-use-secondary-separators** : set whether blocks use a second block horizontal separator line. • **pel-erlang-skel-with-edoc** : set whether generated code comments use EDoc markup. • **pel-erlang-skel-with-license** : set whether file header blocks use open source software license text controlled by **lice**. ☝Emacs user options by default take effect globally. But by using file and directory variables (see ∑ File/Directory Variables) they can also be used to take effect on a single file or all files inside a directory tree. So by default, the user options that control the PEL tempo template take effect globally. If you want to change the behaviour for only one file, write the user option control block at the end of that file. If you want to control the behaviour of the PEL tempo templates for all files inside a directory tree create a .dir-locals file and store the values of the relevant options variables inside that file. This allows you to control the user options affecting the format of the tempo templates precisely and does not affect what you actually type. • Once a skeleton was just entered (or later by activating the pel-tempo-mode) you can move to the next or previous point of interest (so called *tempo-marks)* with the standard tempo-mode keys `C-c M-f` and `C-c M-b` or some other keys like `C-c .` and `C-c ,`. • Instead of using the `<f12><f12>` bindings, you can also type the template name and then hit `C-c C-M-i` or `<f12><f12><f12>`. This supports listing all completions into a separate temporary buffer. This is mainly useful for templates which short names such as “if”, “case”, etc… ☝ Some of the template names in the title column are also links to the relevant Erlang language construct reference page.		
if	`<f12> <f12> i`	(pel-erl-if)	Insert an if statement.
case	`<f12> <f12> c`	(pel-erl-case)	Insert a case expression.
export +	`<f12> <f12> x`	(pel-erl-export	Insert an export module attribute expression.
import +	`<f12> <f12> I`	(pel-erl-import)	Insert an import module attribute expression.
try +	`<f12> <f12> t`	(pel-erl-try)	Insert a try expression.
try-of +	`<f12> <f12> T`	(pel-erl-try-of)	Insert a try expression with of clauses.
receive	`<f12> <f12> r`	(pel-erl-receive)	Insert a receive expression.
after	`<f12> <f12> a`	(pel-erl-after)	Insert a receive expression with an after (timeout) clause.
loop	`<f12> <f12> l`	(pel-erl-loop)	Insert a simple receive loop.
module	`<f12> <f12> m`	(pel-erl-module)	Insert the module attribute.
function C	`<f12> <f12> f`	(pel-erl-function)	Insert a function definition. • This may prompt for function name, argument and purpose according to the user options described above. All prompts maintain independent histories.
author	``<f12> <f12> ` ``	(pel-erl-author)	Insert the author attribute. Uses the **user-mail-address** user option to insert your mail address.
spec	`<f12> <f12> s`	(pel-erl-spec)	Insert a **-spec** for the function following point.
small-header C	`<f12> <f12> M-h`	(pel-erl-small-header)	Insert a small file header without any comment.
normal-header C	`<f12> <f12> M-H`	(pel-erl-normal-header)	Insert a normal file header: includes author name, copyright notice, doc section, file created date.
large-header C	`<f12> <f12> C-h`	(pel-erl-large-header)	Insert a large header block that includes all normal header fields plus separators. • The formatting, use of Edoc, use of separator lines, insertion of automatic timestamp and license of this header and the large header used inside all OTP behaviour skeletons below are controlled by the user options described above. • This command distinguish Erlang module files (files with the .erl extension,) from the Erlang header files (.hrl files) and inserts the appropriate file header block.
small-server C	`<f12> <f12> M-s`	(pel-erl-small-server)	Insert a large file header and template logic for a small server.
application C	`<f12> <f12> M-a`	(pel-erl-application)	Insert a large file header and template logic for an **application behaviour**.
supervisor C	`<f12> <f12> M-u`	(pel-erl-supervisor)	Insert a large file header and template logic for a **supervisor behaviour**.
supervisor-bridge C	`<f12> <f12> M-b`	(pel-erl-supervisor-bridge)	Insert a large file header and template logic for a **supervisor bridge behaviour**.
generic-server C	`<f12> <f12> M-g`	(pel-erl-generic-server)	Insert a large file header and template logic for a **gen-server behaviour**.
gen-event C	`<f12> <f12> M-e`	(pel-erl-gen-event)	Insert a large file header and template logic for a **gen-event behaviour**.
gen-fsm C	`<f12> <f12> M-f`	(pel-erl-gen-fsm)	Insert a large file header and template logic for a **gen-fsm behaviour**.
gen-statem-StateName C	`<f12> <f12> M-S`	(pel-erl-gen-statem-StateName)	Insert a large file header and template logic for a **gen-statem behaviour**.
gen-statem-handle-event C	`<f12> <f12> M-E`	(pel-erl-gen-statem-handle-event)	Insert a large file header and template logic for a gen-statem.
wx-object C	`<f12> <f12> M-w`	(pel-erl-wx-object)	Insert a large file header and template logic for a wx-object generic server.
gen-lib C	`<f12> <f12> M-l`	(pel-erl-gen-lib)	Insert a large file header and template logic for a library module.
gen-corba-cb C	`<f12> <f12> M-c`	(pel-erl-gen-corba-cb)	Insert a large file header and template logic for a **CORBA** callback module.
ct-test-suite-s	`<f12> <f12> M-1`	(pel-erl-ct-test-suite-s)	Insert a large file header and template logic for a test suite
ct-test-suite-l	`<f12> <f12> M-2`	(pel-erl-ct-test-suite-l)	Insert a large file header and template logic for a test suite
ts-test-suite	`<f12> <f12> M-3`	(pel-erl-ts-test-suite)	Insert a large file header and template logic for a test suite

Description	Keystroke	Function	Note
Tempo Template Tag Insertion	• `C-c C-M-i` • `<f12> <f12> <f12>`	(**tempo-complete-tag** &optional SILENT)	Look for a tag and expand it. ☝Instead of using the `<f12><f12>` key bindings above, you can type the template name (shown in the title column like "if", "case", etc) completely or partially and then hit `C-c C-M-i`. (or `<f12><f12><f12>`) A completion buffer opens up if the template name is incomplete (or empty in which case the buffer lists **all** available template names). Select the template name and hit RET. Emacs expands the template. • All the tags in the tag lists in 'tempo-local-tags' (this includes 'tempo-tags') are searched for a match for the text before the point. The way the string to match for is determined can be altered with the variable 'tempo-match-finder'. If 'tempo-match-finder' returns nil, then the results are the same as no match at all. • If a single match is found, the corresponding template is expanded in place of the matching string. • If a partial completion or no match at all is found, and SILENT is non-nil, the function will give a signal. • If a partial completion is found and 'tempo-show-completion-buffer' is non-nil, a buffer containing possible completions is displayed.
Toggle pel-tempo-mode	`<f12> <f12> SPC`	(**pel-tempo-mode** &optional ARG)	Toggle PEL tempo mode on/off. PEL tempo mode activates `C-c .` and `C-c ,` as well as to `C-c C-.` and `C-c C-,` key bindings to navigate across tempo mark hot-spots. When pel-tempo-mode is active the pel-tempo-mode lighter (‡) is shown on the status bar. The second set are only available when Emacs runs in graphics mode. ☝When a skeleton is inserted via the execution of one of the pel-erl-… commands, the pel-tempo-mode is automatically activated.
Jump to next tempo mark	• `C-c M-f` • `C-c .` • `C-c C-.`	(**tempo-forward-mark**)	Jump to the next mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key key bindings are only available when pel-tempo-mode is active.
Jump to previous tempo mark	• `C-c M-b` • `C-c ,` • `C-c C-,`	(**tempo-backward-mark**)	Jump to the previous mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key binding are only available when pel-tempo-mode is active.
Using Flymake to perform dynamic syntax checking	Flymake performs these checks while the user is editing. ☑ Flymake is activated for Erlang source code when **pel-use-erlang-flymake** user option is set to **t**. Flymake has several customizable variables, which some listed here: The following customization variables determine the exact circumstances whereupon Flymake decides to initiate a check of the buffer: • **flymake-start-on-flymake-mode** : **t** to start checking when flymake-mode is started. **nil** to prevent check. • **flymake-no-changes-timeout** : time to wait after last change to start checking. Default = 0.5 seconds. • **flymake-start-syntax-check-on-newline** : **t** to check after insertion or removal of newline char from buffer. **nil** to prevent check. The following variable control navigation to next or previous error: • **flymake-wrap-around** : If non-nil, moving to errors wraps around buffer boundaries. • **flymake-diagnostic-types-alist** : Alist ((KEY . PROPS)*) of properties of Flymake diagnostic types. See Emacs documentation for more info. The `M-n` and `M-p` keys are mapped to flymake commands only when flymake-mode is turned on.		
Toggle Flymake mode on/off	`<f12> F`	(**flymake-mode** &optional ARG)	Toggle Flymake mode on or off. • With a prefix argument ARG, enable Flymake mode if ARG is positive, and disable it otherwise. • Flymake is an Emacs minor mode for on-the-fly syntax checking. • Flymake collects diagnostic information from multiple sources, called backends, and visually annotates the buffer with the results.
Go to next flymake diagnostic	`M-n`	(**flymake-goto-next-error** &optional N FILTER INTERACTIVE)	Move point to the next Flymake diagnostic. • With a prefix arg, skip any diagnostics with a severity less than ':warning'. • Display the error message in the echo line.
Go to previous flymake diagnostic	`M-p`	(**flymake-goto-prev-error** &optional N FILTER INTERACTIVE)	Move point to the previous Flymake diagnostic. • With a prefix arg, skip any diagnostics with a severity less than ':warning'. • Display the error message in the echo line.
Compiling Erlang Code	The following commands are used to compile Erlang source code files to .beam files located in the same directory as the source code. Detected errors are listed in the *erlang* shell opened to compile the files. The buffer shows the location of error and the error description. The following commands are used to navigate to the next or previous detected error.		
Compile code	`C-c C-k`	(**erlang-compile**)	Compile Erlang module in current buffer. • If buffer visiting file was modified and not saved, prompts the user to save it first. • Opens and *erlang* shell, in which the Erlang compile is done with a eshell c() command. • The buffer lists the errors. Hitting <RET> on the error file/line move point to that line in the Erlang file buffer. The <RET> key is bound to (**compile-goto-error** &optional EVENT) • It's also possible to use the next-error and previous error.
Display compilation output	`C-c C-l`	(**erlang-compile-display**)	Display compilation output. • Essentially opens the shell buffer where the last compilation occurred. If that shell was closed nothing can be displayed.
Move to next compile error	• `` C-x ` `` • `M-g n` • `M-g M-n`	(**next-error** &optional ARG RESET)	A prefix ARG specifies how many error messages to move; • negative means move back to previous error messages. • Just `C-u` as a prefix means reparse the error message buffer and start at the first error. ⚠This only shows the result of compilations; it does not report Flycheck reported errors. To use it you must compile the file first.
Move to previous compile error	• `M-g p` • `M-g M-p`	(**previous-error** &optional N)	Prefix arg N says how many error messages to move backwards (or forwards, if negative). ⚠This only shows the result of compilations; it does not report Flycheck reported errors. To use it you must compile the file first.
Move to next compilation or Flycheck detected error	`C-c C-n`	(**edts-code-next-issue** &optional WRAPPED)	Moves point to the next error in current buffer and prints the error. ☝When Flymake is active, this command can be used as soon as an error is reported, even if the file was not compiled.
Move to previous compilation or Flycheck detected error	`C-c C-p`	(**edts-code-previous-issue** &optional WRAPPED)	Moves point to the next error in current buffer and prints the error. ☝When Flymake is active, this command can be used as soon as an error is reported, even if the file was not compiled.
Erlang Shell	The following commands are used to explicitly launch an Erlang shell inside Emacs.		
Open Erlang Shell	`C-c C-z`	(**erlang-shell-display**)	Display the existing Erlang shell, or start a new. Available from Erlang mode buffers only.
Start Erlang Shell	`<f11> x r`	(**erlang-shell**)	Start a new Erlang shell. Can be used from any buffer. • The variable 'erlang-shell-function' decides which method to use, default is to start a new Erlang host. It is possible that, in the future, a new shell on an already running host will be started. • `C-c C-z` starts the Erlang Shell from the Erlang Mode. • `<f11> x r` starts it anytime, as long as it was installed. ☑ Under PEL this command is available only when the **pel-use-erlang** customize variable is set to **t**.

Description	Keystroke	Function	Note
Inside the Erlang Shell	When running the **Erlang Shell** inside Emacs, you may run into some issues. They are listed here along with work-arounds. • *Redundant command echo*: On some systems the Erlang shell annoyingly echoes each typed command. If this is the case for your system, PEL provides a fix: Set the **pel-erlang-shell-prevent-echo** user option to **t**. After doing that execute pel-init or restart Emacs. • *Cannot type the Erlang Ctrl-G escape to access* **Erlang JCL Command Menu**: To pass the Ctrl-G to the Erlang shell running inside Erlang, type: `C-q C-g RET` ⚠️ Unfortunately the above workaround does not work for the Erlang shell invoked via url inside a vterm shell (see ∑ Shells) launched inside Emacs.		
Erlang Shell: Command History	The following commands can be used to retrieve previously issued Erlang shell commands at the shell prompt. ☝The Erlang shell history controlled by Emacs is saved inside a file the is restored when opening a new shell: therefore commands from previously opened Erlang shells are also available. You can also use the **Erlang shell commands** to access the local shell history.		
Next shell command	`M-n`	(**comint-next-input** ARG)	Cycle forwards through Erlang shell input history.
Previous shell command	`M-p`	(**comint-previous-input** ARG)	Cycle backwards through Erlang shell input history, saving input.

Using Man inside Emacs and support Erlang Man pages
 	
(See also: ∑ Help/Info)
	Emacs provide 2 main commands to display man pages inside buffers.
	• Both of these are much more powerful than the usual man reader available on the shell allowing navigation across man pages and opening hyperlinks.
	• The man command uses the system man utility, while woman is a complete implementation. It has some formatting limitations compared to man but it's very useful in systems where man is not available.
	
	To see Erlang man pages using the man command:
	On most systems the Man pages for Erlang are not available to the man utility and therefore not available for man inside Emacs.
	There are several ways this can be remedied:
	• One is to set the MANPATH environment variable to include the directory where these files are located. Then man can be used outside and inside Emacs to access Erlang's man pages. For example the following lines can be stored inside a shell script to do this:
	`MANPATH=`manpath`:/usr/local/Cellar/erlang/22.3.4/lib/erlang/man`
	`export MANPATH`
	• Another way is to customize the Emacs **Man-switches** user option variable to something that includes the same directory. This will add the capability of Emacs man to fin the Erlang's man pages without modifying the capabilities of the parent shell. For example, if we want to use the same directory as the above example we need to set the Man-switches which is normally set to nil to the following value:
	"-M`manpath`:/usr/local/Cellar/erlang/22.3.4/lib/erlang/man"
	
	The second alternative can be used to add other directories for the man pages of other programming languages while leaving the ability to have several shells that have their own value of MANPATH. That might be very useful for someone that uses different versions of Erlang in a system and needs access to the man pages of different versions of Erlang. It becomes possible to run different shells inside Emacs with each having its own value of MANPATH and therefore providing the man pages from different locations. It is also possible to place all of these directories inside the Man-switches or MANPATH and buses man's ability to view several pages for the same topic.
	
	To only see Erlang topics in Man completion:
	When learning Erlang it might help to see only Erlang topics when using the man command completion. To do that , set MANPATH to the Erlang man directory only. You must also ensure that a whatis file is located in the Erlang man page root directory, otherwise Emacs man completion will not work. See my description on how to create whatis file for local man directory.
	
	Using EDTS to access the man pages of the version of Erlang used by various projects:
	EDTS (see below) supports the ability to download and access man pages of several Erlang versions, tied to your Erlang projects. EDTS provides it's own help command to access sections inside the mane pages, allowing EDTS driven man page access to co-exist with manual man command execution and the techniques described above.		

Open a man page inside an Emacs buffer (See also: ∑ Help/Info)	• `<f11> ? m` • `⌘-M`	(**man** MAN-ARGS)	Using man pages inside emacs is even better than using it from the shell because: • the links are active and can be followed. When the man page describes a directory or file, emacs will open the file or the directory (in direct mode) when pressing <RET> over the link. • You can navigate easily between sections (n/p will move to the next/previous section) • You can use any of the searches. • You can use any of the options to the man command at the prompt, like the -a option to access all man pages of the same name. Then use **M-n** and **M-p** to move from one to the other page, inside the same buffer. • See all keys available in mode, with **<f1> m** or **<f11> ? k m**. ☝The man command prompts, using the word at point as the default. PEL key sequence to customize man: `<f11> <f1> M-g m`
Open a man page without external man process: woman (See also: ∑ Help/Info)	`<f11> ? w`	(**woman** &optional TOPIC RE-CACHE)	Open a man page file in Emacs using the woman mode, completely implemented in Emacs Lisp (and therefore without using the external 'man' process). That can be very useful under environments where man is not available (such as basic Windows). PEL key sequence to customize man: `<f11> <f1> M-g w` • text width, use word at point, etc…
EDTS	**EDTS - Erlang Development Tool Suite** 📦The commands in the following rows require the EDTS external package. PEL activates it when the **pel-use-edts** user option is set to **t**. EDTS is customizable through it **edts** customization group. With PEL you can open it, with other Erlang specific groups with `C-u <f12> <f1>` . If you want EDTS to start automatically when you open an Erlang file, set **pel-activate-edts-automatically** to **t**. ☛ If EDTS has not been activated yet, the only EDTS specific key available is `<f12> M-SPC` to activate it. Once it's activated the other keys are available.		
Toggle EDTS mode	`<f12> M-SPC`	(**edts-mode** &optional ARG)	Turn EDTS mode on or off. EDTS is an easy to set up Development-environment for Erlang. EDTS also incorporates a couple of other minor-modes, currently auto-highlight-mode and auto-complete-mode. They are configured to work together with EDTS but see their respective documentation for information on how to configure their behaviour further.
EDTS/Cross References	EDTS provides the following cross-reference commands. It supports navigating in Erlang source code running in the current and remote nodes.		
Find definition of identifier at point	`M-.`	(**edts-find-source-under-point**)	Goto the source code that: defines the function being called at point or header file included at point. For remote calls, contacts an Erlang node to determine which file to look in, with the following algorithm: • Find the directory of the module's beam file (loading it if necessary). • Look for the source file in: • Directory where source file was originally compiled. • Todo: Same directory as the beam file • Todo: Again with /ebin/ replaced with /src/ • Todo: Again with /ebin/ replaced with /erl/ Otherwise, report that the file can't be found.
Go back to where M-. was last issued	`M-,`	(**edts-find-source-unwind**)	Unwind back from uses of 'edts-navigate'-commands.
Lists caller of function at point	• `C-c C-d w` • `<f12> w`	(**edts-xref-who-calls**)	Pops-up a menu of all callers of the function at point.
List the callers again	• `C-c C-d W` • `<f12> W`	(**edts-xref-last-who-calls**)	Redo previous call to edts-who-calls.
Find a function in the current module	• `C-c C-d f` • `<M-f12> M-f`	(**edts-find-local-function** SET-MARK)	Find a function in the current module. • List local functions in the mini-buffer. Support completion. Move point to selected one. • With `C-u` prefix, push mark before moving point.
Find a module in the current project	• `C-c C-d F` • `<M-f12> M-g`	(**edts-find-global-function**)	Find a module in the current project. • List project modules in the mini-buffer. Support completion. Open the file of selected one.

Description	Keystroke	Function	Note
EDTS/AHS Editing	EDTS supports the automatic highlight symbol mode (AHS). and provides commands to modify the name of the highlighted name in the current function or in all of the buffer. The automatic symbol highlighting mode starts when the cursors stays on a symbol for a period longer than the value identified by the **ahs-idle-interval** which defaults to 1.0 second. ☝To turn off the AHS editing mode, use a command to move point away from the highlighted area.		
Edit all highlighted symbols in current function	• `C-c C-d e` • `<f12> e`	(**edts-ahs-edit-current-function**)	Once a symbol is highlighted, use this command to start editing all instances of this symbol in the current function. • Activates ahs-edit-mode with edts-current-function range-plugin.
Edit all highlighted symbols in buffer	• `C-c C-d E` • `<f12> E`	(**edts-ahs-edit-buffer**)	Once a symbol is highlighted, use this command to start editing all instances of this symbol in the current buffer. • Activates ahs-edit-mode with ahs-range-whole-buffer range-plugin.
Move to the next highlighted symbol	`<f12> n`	(**ahs-forward**)	Once a symbol is highlighted, move forward to the next highlighted symbol.
Move to the previous highlighted symbol	`<f12> p`	(**ahs-backward**)	Once a symbol is highlighted, move forward to the previous highlighted symbol.
Move to the originally highlighted symbol	`<f12> .`	(**ahs-back-to-start**)	Once a symbol is highlighted, move back to the symbol that was highlighted at the start of that highlight session.
Refactor: replace region by call to function and add a new function	• `C-c C-d r` • `<f12> r`	(**edts-refactor-extract-function** NAME START END)	Refactor the expression(s) in the region as a function. • The expressions are replaced with a call to the new function, and the function itself is placed on the kill ring for manual placement. The new function's argument list includes all variables that become free during refactoring - that is, the local variables needed from the original function. • New bindings created by the refactored expressions are *not* exported back to the original function. Thus this is not a "pure" refactoring. • This command requires **Erlang syntax_tools** package to be available in the node, version 1.2 (or perhaps later.)
EDTS/Man	EDTS supports opening documentation for a specific function using the information extracted from Erlang Man pages. EDTS maintains a set of Erlang man pages per project, so it is possible to have several Erlang projects each one with a different version of Erlang and their corresponding man pages. These EDTS commands complement the Emacs standard man commands described above in this table.		
Download, install, select Erlang Man pages	`<f12> ``	(**edts-man-setup**)	Download and install OTP man-pages that will be used by the following 2 EDTS commands.
Display help for function at point	• `C-c C-d h` • `<f12> h`	(**edts-show-doc-under-point**)	Find and display the man-page documentation for function under point in a tooltip.
Find and show man-page info for an Erlang module:function	• `C-c C-d H` • `<f12> H`	(**edts-find-doc**)	Prompts for a module, then a function. Find and show the man-page documentation for the Erlang module:function.
EDTS Code Analysis			
Compile current buffer 0	`<f12> a c`	(**edts-code-compile-and-display**)	Compiles current buffer on node related to that buffer's project.
Run eunit tests	• `C-c C-d t` • `<f12> a t`	(**edts-code-eunit** &optional COMPILATION-RESULT)	Runs eunit tests for current buffer on node related to that buffer's project.
Run dialyzer	`<f12> a a`	(**edts-dialyzer-analyze**)	Runs dialyzer for all live buffers related to current buffer either by belonging to the same project or, if current buffer does not belong to any project, being in the same directory as the current buffer's file.
EDTS/Debug			
Toggle breakpoint	• `C-c C-d b` • `<f12> d b`	(**edts-debug-toggle-breakpoint**)	Toggle breakpoint on current line.
List breakpoints	`C-c C-d M-b` • `<f12> d B`	(**edts-debug-list-breakpoints** &optional SHOW)	Show a listing of all breakpoint on all nodes registered with EDTS. If optional argument SHOW is nil or omitted, don't display process list buffer. If it is pop call 'pop-to-buffer', if it is switch call 'switch-to-buffer'.
List Erlang processes	• `C-c C-d M-p` • `<f12> d p`	(**edts-debug-list-processes** &optional SHOW)	Show a listing of all processes on all nodes registered with EDTS. If optional argument SHOW is nil or omitted, don't display process list buffer. If it is pop call 'pop-to-buffer', if it is switch call 'switch-to-buffer'.
Toggle interpretation state of module	• `C-c C-d i` • `<f12> d i`	(**edts-debug-toggle-interpreted**)	Toggle the interpretation state for module in current buffer.
List interpreted modules	• `C-c C-d M-i` • `<f12> d I`	(**edts-debug-list-interpreted** &optional SHOW)	Show a listing of all interpreted modules on all nodes registered with EDTS. If optional argument SHOW is nil or omitted, don't display interpreted list buffer. If it is pop call 'pop-to-buffer', if it is switch call 'switch-to-buffer'.
EDTS/Erlang Node			
Display EDTS Erlang Node Name	`<f12> N`	(**edts-buffer-node-name**)	Print the node sname of the erlang node connected to current buffer. • The node is either: • The module's project node, if current buffer is an erlang module, or • The buffer's erlang node if buffer is an edts-shell buffer. • The project-node of the buffer that was current buffer before jumping to the current buffer if the file of the current buffer is located outside any project (eg. an "externally" loaded module such as an otp-module or a module loaded by ~/.erlang).
Start an EDTS controlled Erlang Shell	`<f12> x`	(**edts-shell** &optional PWD SWITCH-TO)	Start an interactive erlang shell.
Start EDTS server	`<f12> X`	(**edts-api-start-server**)	Starts an edts server-node in a comint-buffer (if not already running).
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside Erlang source code comments. This can be useful when using these markup languages to describe UML diagrams or finite-state machines for example.		
Preview UML diagram from plantUML source in current plantUML region of commented source code (See also: ℳPlantUML)	`<f12> u`	(**pel-render-commented-plantuml** PREFIX &optional POS)	Render the PlantUML markup embedded in current mode comment. • Use region if identified otherwise use PlantUML block at point. • Uses prefix (as PREFIX) to choose where to display it: • 4 (when prefixing the command with **C-u**) -> new window • 16 (when prefixing the command with **C-u C-u**) -> new frame. • else -> new buffer • This can be used inside buffer using **any** major mode, when PlantUML markup is embedded inside source code comment. ☝ Use this in source code to describe your code architecture with PlantUML markup, then generate the UML rendering by moving point inside the PlantUML block and issuing this command. 📦 Requires the **plantuml-mode** external package, ☑ activated by **pel-use-plantuml** user option being non-nil.

Description	Keystroke	Function	Note
Preview diagram created from Graphviz DOT markup embedded in comments (See also: ℳGraphviz Dot)	`<f12> G`	(**pel-render-commented-graphviz-dot** &optional POS)	Render the Graphviz-Dot markup embedded in current mode comment. Search at POS if specified, otherwise search around point. Use region if identified otherwise use Graphviz-Dot block. ☝️The graphviz DOT code must be located within a block delimited by the following special keywords (that are also in comments): • **@start-gdot** • **@end-gdot** ⚠️ The current implementation leaves the created image file in a temporary directory. You will probably want to move that file or delete it, otherwise the size of this directory will increase with each of these created files. The file names use the pel-gdot- prefix. 📦 Requires the **graphviz-dot-mode package** external package, ☑ activated by **pel-use-graphviz-dot** user option set to **t**.
Development Tool	The following commands are used when adding Emacs Lisp support for Erlang.		
Show syntactic information	`C-c C-s`	(**erlang-show-syntactic-information**)	Show syntactic information for current line. • Display semantic Lisp data structure in the echo line. Not useful for writing Erlang.
🚧 **TODO**			Create a PEL command to create/update the TAGS file for the current Erlang project
			Inside the Emacs erlang shell, MFA expansion with the Meta key does not work the way it works in a pure Erlang shell. Why?

Emacs & Erlang— References

Document	Notes
Erlang/OTP	Erlang/OTP home page. This is Erlang's official site.
Erlang versions	• **Erlang Versions - Version Scheme** • **Erlang Support, Compatibility, Deprecations, and Removal**
Erlang/OTP @ Github	Erlang source code
Erlang Community	Links to various topics including how to develop Erlang, learning Erlang, Community mailing lists and chats, contribution, Erlang Issue Tracker, events.
Erlang Mailing Lists	The mailing lists still exist but unfortunately seem to be used less and less.
Erlang References	
Erlang Reference Manual User's Guide	The official Erlang language reference. Lists the BIFs (Built-in functions), reserved words, and all language reference info.
Erlang Code Guidelines	
Erlang Programming Rules and Conventions	Official Ericsson AB Erlang guidelines.
Inaka's Erlang Coding Standards & Guidelines	Guideline used at Inaka, published on Github.
EDoc User's Guide	Describes how to document code.
Erlang Books	There are several printed and online Erlang books. Erlang's FAQ lists several of them. The following lists some extra ones.
Adopting Erlang	A great and recent (2019 and later) online books on Erlang Development that provides information not available in the Erlang introduction books. Describes how to install Erlang, and how to setup editing tools. A must read to setup Erlang development. This is still work in progress as of May 2020. Each page has a date time stamp.
Erlang Information Sites	
How to setup a local Erlang & Elixir dev environment on Mac from source	LambdaCat post on August 2015. Describes how to use Kerl to install Erlang. Also describes tools to install Elixir. However to get kerl on a macOS machine, using Homebrew is simpler.
• **about-erlang** • **trying-erlang**	These are 2 projects of mine, that I am currently building to centralize some information on Erlang.
Emacs and Erlang Man files	
How to create a local whatis file	Show how to create aa missing whatis file for a set of man pages.
• **The Erlang mode for Emacs (user guide)** • **Erlang mode for Emacs (man page)**	On the erlang.org site. Start here. Describes the 2 files (erlang.el and erlang-start.el) provided by the Erlang mode support, how to set them up for various operating systems. Note, however, that PEL provides the setting for you. It also provides an overview of the various features the package provides. • 🐞 I found bugs in the erlang man page in the **Edit- Moving the marker** section. 1) it's the point that is moved, not the marker, 2) C-a is not an Emacs key prefix, so their key binding descriptions like C-a M-a and C-a M-e are invalid. Reported as ERL-1314. • There's missing information in this. I will identify later as I find out how to get the system going. One aspect to learn more is related to the various erlang-electric functions and variables. • The variable erlang-electric-commands was set to (erlang-electric-comma erlang-electric-semicolon erlang-electric-gt) at first, which does not include the erlang-electric-newline function. I tried adding erlang-electric-newline and activated it, but that made things worse: the newline was no longer automatic after a -> on a function definition line. • Another issue: inside the OS-level erlang shell, we can tab-completion a module:function string, but that does not work inside the emacs erlang shell.
Emacs tools for Erlang	

EDTS	EDTS: stands for: The Erlang Development Tool Suite. See also:
	• EDTS Tool Suite - Making Your Life Easier - Thomas Järvstrand presentation @ Youtube
	  • EDTS:
	    • configure your project
	    • One Primary EDTS node
	    • 1 node per open project
	  • To setup an Erlang project: a **.edts** file in the project:
	`:name "my-project"`
	`:otp-path "path/to/otp"`
	`:node-name "project-node-name"`
	`:lib-dirs '("lib" "deps")`

How to install EDTS	Describes some aspects of EDTS and links that may be useful. Lists the requirements. ⚠️🚧After installing EDTS, I got several compile errors, and had to install the following other modules: - auto-complete (v1.5.1) - have to read doc and configure. And perhaps disable company mode?

Document	Notes
company-mode ; Modular in-buffer completion framework for Emacs	
Using Tags with Erlang	
Etags with Erlang @ erlang.org	Describes how to use tags with Erlang source code and how to create the TAGS file.
Troubleshooting	This section describes how to solve some of the problems you may encounter with Erlang on Emacs.
How to prevent Erlang shell echo	On some systems the Erlang shell annoyingly echoes every command typed at the shell. The Emacs manual describes a method to prevent shells inside Emacs from echoing and it describes it as affecting Windows systems. None of the Emacs shells on my system that runs on macOS echo commands, but the Erlang shell does. And the described fix works. PEL activates the fix if the **pel-erlang-shell-prevent-echo** is set to **t**. To activate after setting it: execute pel-init or restart Emacs.