www.erlang-solutions.com

Good Design Practices

Erlang
SOLUTIONS

Good Design Practices

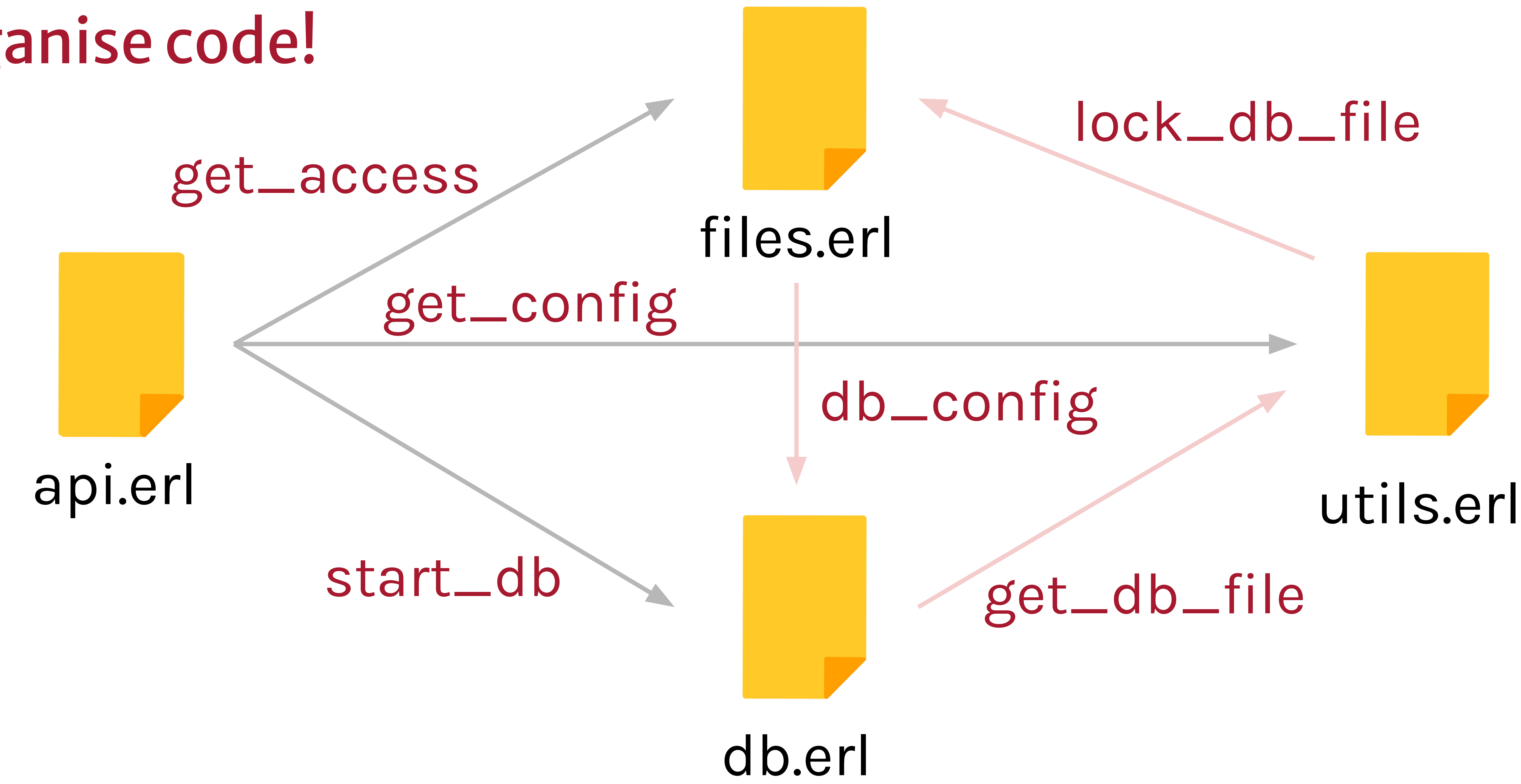
- Style
  - Applications and Modules
  - Libraries
  - Return Values
  - Internal Data Structures
  - Concurrency
  - Conventions
- Efficiency

Applications and Modules

- Prefix module names part of a given application or library with the name of the application/library
- If many submodules exist for the same application, create an interface module acting as a single point of entry
  - Allows more flexibility to refactor internal code.
- For the user, the complexity of a module is proportional to the number of exported functions
- Try to write modules with fewer responsibilities and exported functions

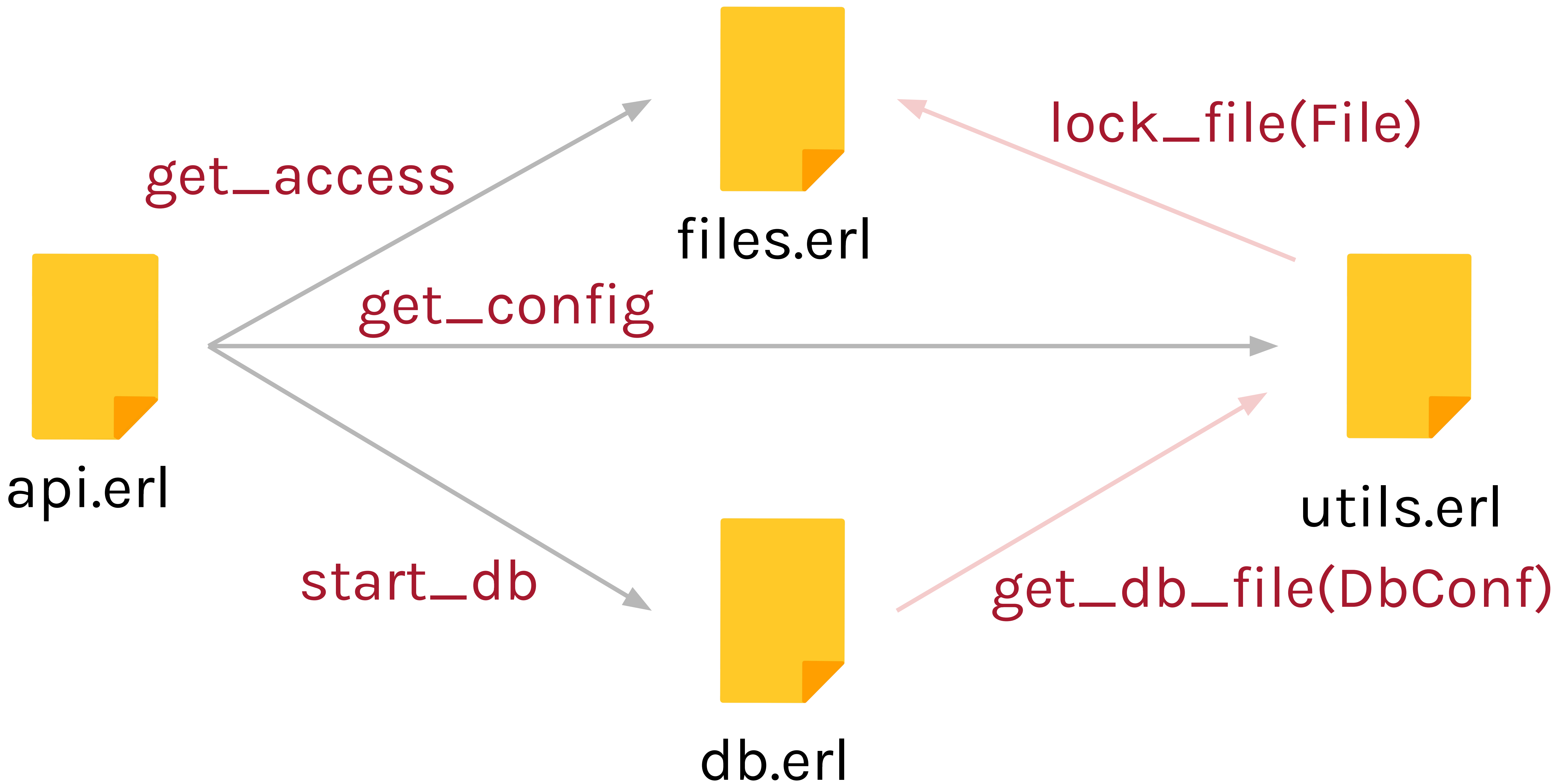
Applications and Modules

Reorganise code!



- ➢ Try to reduce inter-module dependencies
  - ○ Makes it easier to refactor
- ➢ The call graph should be an acyclic graph

Applications and Modules



- ➢ Try to reduce inter-module dependencies
  - Makes it easier to refactor
- ➢ The call graph should be an acyclic graph

Libraries

- Collections of related functions should be regrouped into libraries
- The functions should be free of side effects
- If they have side effects, make sure they are related together
  - Functions manipulating a given ETS table, for example
- Document the exported functions!

Return Values

- The return value of a failing function call should be distinct from a successful one
  - it is sometimes impossible to know if there is an error or a value that is the same as the error case
  - tag tuples: {ok, undefined} vs. {error, undefined}.
  - If the function call always returns a value of a different type when it successful than when it fails, tagging tuples is not necessary
- Pick values that will simplify the caller's task
  - Do not tag values if the call should always succeed, let it crash when it fails

Internal Data Structures

```
-module(q).
-export([add/2, fetch/1]).

add(Item, Q) ->
   lists:append(Q, [Item]).

fetch([])    -> {error, empty}.
fetch([H|T]) -> {ok, H, T};

%% Used as follows

main() ->
    NewQ = [],
    Queue1 = q:add(joe, NewQ),
    Queue2 = q:add(mike, Queue1).
```

- Do not allow private data to leak out
  - All details about the data structure should be abstracted away by the interface
- In the example, NewQ = [] exposes information about the structure of the queue
  - It is impossible to change the representation without changing all the callers

Internal Data Structures

```
-module(q).
-export([add/2, fetch/1,
         empty/0]).

empty() -> [].

...


%% Used as follows

main() ->
    NewQ = q:empty(),
    Queue1 = q:add(joe, NewQ),
    Queue2 = q:add(mike, Queue1).
```

- The new representation is equivalent in functionality
- Allows to change the queue representation without modifying the callers

Concurrency: hiding information

```
resource_server ! {free, Resource}

%% becomes

resource_server:free(Resource)
```

- Hide all message passing in a functional interface for greater flexibility, debugging capabilities and information hiding
- Place the client functions in the same module as the process
  - Makes it easier to follow the message flow without jumping between modules
- Create a mapping between each process and a concurrent activity in the system you are modelling.

Concurrency: tagging messages

```
loop(State) ->
  receive
    {Mod, Fun, Args} ->
      NewState = apply(Mod, Fun,
                       Args),
      loop(NewState)
  end.

loop(State) ->
  receive
    {apply, Mod, Fun, Args} ->
      NewState = apply(Mod, Fun,
                       Args),
      loop(NewState)
  end.
```

- Matching only on unbound variables can be problematic
  - The pattern **{Mod, Fun, Arg}** can also match **{system, From, Req}** and break the server that expects other values
- Tag the messages to be sent and received
  - By tagging all messages and maintaining a functional interface, the matching is generally safer

Concurrency: using references

```
call(Pid, Message) ->
  Ref = make_ref(),
  Pid ! {request, {Ref, self()},
         Message},
  receive
    {reply, Ref, Reply} ->
       Reply
  end.

reply({Ref, Pid}, Reply) ->
  Pid ! {reply, Ref, Reply}.
```

- make_ref/0 returns a unique reference data type.
- Similar requests and responses might come from different processes
- Multiple communications with replies can become confusing
- References help uniquely tag a set of messages to identify them

Concurrency: using references

```
call(Pid, Message) ->
  Ref = erlang:monitor(process, Pid),
  Pid ! {request, {Ref, self()}, Message},
  receive
    {reply, Ref, Reply} ->
       erlang:demonitor(Ref, [flush]),
       Reply;
    {'DOWN', Ref, process, Pid, _} -> error(noproc)
  end.

reply({Ref, Pid}, Reply) -> Pid ! {reply, Ref, Reply}.
```

- Monitors allow the same properties as using **make_ref/0**
- They allow to find failures or missing processes earlier
- They must be taken down after a successful call

Conventions: nesting

- Avoid deeply nested code
- Keep only two levels of indentation for:
  - case
  - if
  - receive
  - funs
- Reduce indentation by
  - pattern matching
  - creating temporary composite data types
  - adding more functions

Conventions: nesting example

```
case Oper of
  enabled ->
      case Admin of ->
          enabled -> restart(Board);
          disabled -> ok
      end;
  disabled ->
      ok
end.

case {Oper, Admin} of
   {enabled, enabled} -> restart(Board);
   {_,_}              -> ok
end.
```

Conventions: code size

- Avoid long lines
  - Line should not be longer than 80 characters by convention
  - People who review the code by printing it have it easier
  - Can read more files with a split screen
- Avoid large modules
  - A module should usually not contain more than 500 lines of code, plus comments.
  - Not a strict metric, can be broken, but usually helps restricting each module to one responsibility

Conventions: code size

- Make your functions short
  - A function should rarely be above 50 lines and it should not span many pages
  - Versions of Erlang prior to R15 do not give line numbers in stack traces, makes it easier to debug
  - debugging trace flags can be enabled on specific functions only

Conventions: strategies

- Minimise the number of functions with side effects
- Abstract out common design patterns
- Avoid defensive programming
  - When you do not know how to handle a specific error, let it crash.
- Avoid copy-and-paste programming
- Don't comment out dead code, just delete it
  - Get it back from your repository if you need it again in the future

Good Design Practices

- Style
- Efficiency
  - Pattern matching
  - Recursion
  - IO lists, strings and binaries
  - Garbage collection
  - Concurrency

Pattern Matching: rearranging clauses

```
day(monday)    -> weekday;
day(tuesday)   -> weekday;
day(wednesday) -> weekday;
day(thursday)  -> weekday;
day(friday)    -> weekday;
day(Name) when Name==saturday; Name==sunday -> weekend.
```

- There is nothing to gain by rearranging most of the clauses
- If the function is called more often with friday, it is not worth it to put day(friday) first before all other clauses.
- The compiler knows how to read clauses in the most efficient way possible without losing the semantics of the code:
  - A single instruction tries the first five clauses with a binary search
  - The last clause is tried if none of the first did match

Pattern Matching: rearranging clauses

```
day(monday)    -> weekday;
day(tuesday)   -> weekday;
day(wednesday) -> weekday;
day(Name) when Name==saturday; Name==sunday -> weekend;
day(thursday)  -> weekday;
day(friday)    -> weekday.
```

- The compiler does a fast search to know if the value matches monday, tuesday, or wednesday.
- If they do not match, the clause with **Name** then always matches
  - The guard test is executed
- If the guard test fails, the two last clauses are searched

Pattern Matching: exceptions

- Functions pattern matching with binaries are never rearranged
  - usually, placing the clause with the empty binary (<<>>) last is faster
  - This is version-specific and could change with newer releases
- Complex patterns and guards can make things harder for the compiler and lead for more manual adjustments and rearranging

Recursion

- Tail-recursive functions are not necessarily faster than non-tail-recursive functions
  - If the function generates a single value (ex.: factorial), tail-recursive functions are generally faster
  - If the function takes a list and returns a list of the same size, non-tail-recursive functions might be faster. ex.: lists:map/2 is implemented in a non-tail-recursive manner
  - If the function is a server loop that never returns, you must use tail-recursive calls
  - Measure and profile your code

IO Lists, Strings and Binaries

- Strings are implemented as lists of integers
  - They take more space
  - Useful if the string needs to be transformed and manipulated
- Binary strings are implemented as binaries
  - Compact with O(1) read access
  - Useful for storage, transport and limiting the size in memory

IO Lists, Strings and Binaries

```
1> IoList = [$", "hello", " ", <<"hello",65>>, [<<$.,$ ,$W>>, [$o]] |
"rld"].
[34,"hello"," ",<<"helloA">>,[<<". W">>,"o"],114,108,100]
2> io:format("~s~n",[IoList]).
"hello helloA. World
ok
```

- IO lists are lists of binaries, strings, or characters used for input/output operations
  - Automatically flattened by any io function, socket or port. No need to flatten it yourself.
  - They allow to append data in constant time by just wrapping the elements in a list: [OldString | NewEndOfString].
  - Allows to mix the use of any string representation for a single element

IO Lists, Strings and Binaries

- For general string manipulation, use the **re** module
  - PCRE engine in C, faster than the obsolete **regexp** module
  - Works with all types of strings and can give the output format of your choice (index, list or binary)
- Use the binary module for binary string manipulation
- Use the string module for lists-based string manipulation

Garbage Collection

- Garbage Collection costs are largely proportional to the amount of live data on the heap
  - Large heaps for selected processes can give significant performance gains
- Try to avoid building large heap structures
  - ets:foldl/3, ets:foldr/3, ets:tab2list/1, etc.
  - Copying and rewriting lists
    - Use iolists rather than flattening strings when possible

Garbage Collection: heap size

- Set each process' heap size with
  - **spawn_opt(Module, Function, Args, OptionList)**
  - where OptionList includes **{min_heap_size, Size}**.
- Erlang heap memory is divided into the old heap and the new heap
  - Data in the new heap that survives a garbage collection sweep is moved to the old heap.
  - The option {fullsweep_after, Number} makes it possible to specify the number of garbage collections which should occur before the old heap is swept

Garbage Collection: heap size

- The **fullsweep_after** and **min_heap_size** options can be set globally for new processes by calling **erlang:system_flag(Flag, Value)**.
  - Use **fullsweep_after** only if you know there are problems with the memory consumption of your processes
  - the **min_heap_size** can be set when starting the VM with the **+h Size** flag
- You can force a garbage collection with **garbage_collect()** for the current calling process or **garbage_collect(Pid)** for a particular one

Concurrency: **hibernation**

```
erlang:hibernate(Module, Function, Arguments)
```

- If a particular process has to wait for a message for a while doing nothing, it can be compacted by calling **erlang:hibernate/3**. The process wakes up when receiving a message and keeps running with **Module:Function(Arg1, ...)**.
    - Discards the call stack for the process
    - The process is garbage collected
    - All of the data is stored in one continuous heap, which is then shrunken to the exact same size as the data it holds.
- Hibernation is a tradeoff between processor use and memory use. Mostly useful for processes that are mostly inactive.

Concurrency: message passing

- Data is fully copied between processes
- Send messages about what happened, not the new state to carry
  - This tends to reduce message size, although this is generally a minor efficiency issue.
- Binaries larger than 64 bytes are passed around as pointers and are not copied between processes on a single node
  - Storing text or data that needs to be forwarded between many processes could be planned as binaries if required

Coding Strategies

First, make it work

Then, make it beautiful

Then, if you really have to, make it fast

Good Design Practices

- Style
- Efficiency
  - Pattern matching
  - Recursion
  - IO lists, strings and binaries
  - Garbage collection
  - Concurrency