
Machinery

license Apache-2.0 license Apache-2.0 hex docs license Apache-2.0 license Apache-2.0



Machinery is a lightweight State Machine library for Elixir with built-in Phoenix integration. It provides a simple DSL for declaring states and includes support for guard clauses and callbacks.

Table of Contents

- Installing
- Declaring States
- Changing States
- Persist State
- Logging Transitions
- Guard Functions
- Before and After Callbacks

Installing

Add `:machinery` to your list of dependencies in `mix.exs`:

```
1 def deps do
2   [
3     {:machinery, "~> 1.1.0"}
4   ]
5 end
```

Create a `state` field (or a custom name) for the module you want to apply a state machine to, and ensure it's declared as part of your `defstruct`.

If using a Phoenix model, add it to the schema as a `string` and include it in the `changeset/2` function:

```
1 defmodule YourProject.User do
2   schema "users" do
3     # ...
4     field :state, :string
5     # ...
6   end
7
8   def changeset(%User{} = user, attrs) do
9     #...
10    |> cast(attrs, [:state])
11    #...
12  end
13 end
```

Declaring States

Create a separate module for your State Machine logic. For example, if you want to add a state machine to your `User` model, create a `UserStateMachine` module.

Then import `Machinery` in this new module and declare states as arguments.

`Machinery` expects a `Keyword` as an argument with the keys `field`, `states` and `transitions`.

- `field`: An atom representing your state field name (defaults to `state`)
- `states`: A `List` of strings representing each state.
- `transitions`: A `Map` for each state and its allowed next state(s).

Example

```
1 defmodule YourProject.UserStateMachine do
2   use Machinery,
3     field: :custom_state_name, # Optional, default value is `:field`
4     states: ["created", "partial", "completed", "canceled"],
5     transitions: %{
6       "created" => ["partial", "completed"],
7       "partial" => "completed",
8       "*" => "canceled"
9     }
10 end
```

You can use wildcards `"*"` to declare a transition that can happen from any state to a specific one.

Changing States

To transition a struct to another state, call `Machinery.transition_to/3` or `Machinery.transition_to/4`.

`Machinery.transition_to/3` or `Machinery.transition_to/4`

It takes the following arguments:

- `struct`: The `struct` you want to transition to another state.
- `state_machine_module`: The module that holds the state machine logic, where `Machinery` is imported.
- `next_event`: `string` of the next state you want the struct to transition to.
- (optional) `extra_metadata`: `map` with any extra data you might want to access on any of the state machine functions triggered by the state change

```
1 Machinery.transition_to(your_struct, YourStateMachine, "next_state")
2 # {:ok, updated_struct}
3
4 # OR
5
6 Machinery.transition_to(your_struct, YourStateMachine, "next_state", %{
7   extra: "metadata"})
7 # {:ok, updated_struct}
```

Example

```
1 user = Accounts.get_user!(1)
2 {:ok, updated_user} = Machinery.transition_to(user, UserStateMachine, "
  completed")
```

Persist State

To persist the struct and state transition, you declare a `persist/2` or `/3` (in case you wanna access metadata passed on `transition_to/4`) function in the state machine module.

This function will receive the unchanged `struct` as the first argument and a `string` of the next state as the second one.

your `persist/2` or `persist/3` should always return the updated struct.

Example

```
1 defmodule YourProject.UserStateMachine do
2   alias YourProject.Accounts
3
4   use Machinery,
5     states: ["created", "completed"],
6     transitions: %{"created" => "completed"}
7
8   # You can add an optional third argument for the extra metadata.
9   def persist(struct, next_state) do
10    # Updating a user on the database with the new state.
11    {:ok, user} = Accounts.update_user(struct, %{state: next_state})
12    # `persist` should always return the updated struct
13    user
14  end
15 end
```

Logging Transitions

To log transitions, Machinery provides a `log_transition/2` or `/3` (*in case you wanna access meta-data passed on `transition_to/4`*) callback that is called on every transition, after the `persist` function is executed.

This function receives the unchanged `struct` as the first argument and a `string` of the next state as the second one.

`log_transition/2` or `log_transition/3` should always return the struct.

Example

```
1 defmodule YourProject.UserStateMachine do
2   alias YourProject.Accounts
3
4   use Machinery,
5     states: ["created", "completed"],
6     transitions: %{"created" => "completed"}
7
8   # You can add an optional third argument for the extra metadata.
9   def log_transition(struct, _next_state) do
10    # Log transition here.
11    # ...
12    # `log_transition` should always return the struct
13    struct
14  end
15 end
```

Guard functions

Create guard conditions by adding `guard_transition/2` or `/3` (in case you wanna access meta-data passed on `transition_to/4`) function signatures to the state machine module. This function receives two arguments: the `struct` and a `string` of the state it will transition to.

Use the second argument for pattern matching the desired state you want to guard.

```
1 # The second argument is used to pattern match into the state
2 # and guard the transition to it.
3 #
4 # You can add an optional third argument for the extra metadata.
5 def guard_transition(struct, "guarded_state") do
6   # Your guard logic here
7 end
```

Guard conditions will allow the transition if it returns anything other than a tuple with `{:error, "cause"}: - {:error, "cause"}: Transition won't be allowed. - _ (anything else): Guard clause will allow the transition.`

Example

```
1 defmodule YourProject.UserStateMachine do
2   use Machinery,
3     states: ["created", "completed"],
4     transitions: %{"created" => "completed"}
5
6   # Guard the transition to the "completed" state.
7   def guard_transition(struct, "completed") do
8     if Map.get(struct, :missing_fields) == true do
9       {:error, "There are missing fields"}
10    end
11  end
12 end
```

When trying to transition a struct that is blocked by its guard clause, you will have the following return:

```
1 blocked_struct = %TestStruct{state: "created", missing_fields: true}
2 Machinery.transition_to(blocked_struct, TestStateMachineWithGuard, "
3   completed")
4 # {:error, "There are missing fields"}
```

Before and After callbacks

You can also use before and after callbacks to handle desired side effects and reactions to a specific state transition.

You can declare `before_transition/2` or `/3` (in case you wanna access metadata passed on `transition_to/4`) and `after_transition/2` or `/3` (in case you wanna access metadata passed on `transition_to/4`), pattern matching the desired state you want to.

Before and After callbacks should return the struct.

```
1 # Before and After callbacks should return the struct.
2 # You can add an optional third argument for the extra metadata.
3 def before_transition(struct, "state"), do: struct
4 def after_transition(struct, "state"), do: struct
```

Example

```
1 defmodule YourProject.UserStateMachine do
2   use Machinery,
3     states: ["created", "partial", "completed"],
4     transitions: %{
5       "created" => ["partial", "completed"],
6       "partial" => "completed"
7     }
8
9   def before_transition(struct, "partial") do
10     # ... overall desired side effects
11     struct
12   end
13
14   def after_transition(struct, "completed") do
15     # ... overall desired side effects
16     struct
17   end
18 end
```

Copyright and License

Copyright (c) 2016 João M. D. Moura

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.