MTAT.03.295
Agile Software Development
Lecture 4: Elixir GenServer processes

Luciano García-Bañuelos
Elixir processes

› Process is the cornerstone for Elixir’s concurrency model
  – Provides also the means for building distributed and fault-tolerant software

› In Elixir, all code runs inside processes
  – Run in isolation from each other
  – Run concurrent to one another, without sharing any state
  – Communicate with each other via message passing
Not to confuse with
› Operating system processes
› Not even with threads

› Elixir processes are extremely lightweight
› HOW?
  – Each process maintains its own state => No need for mutual exclusion mechanisms
  – For the same reason other mechanisms such as garbage collection and instantiation is extremely fast.
First example: A stack

```elixir
defmodule Stack do
  def create(), do: loop([])

  defp loop(state) do
    receive do
      {:push, value} -> loop([value | state])
      {:pop} -> IO.puts hd(state), loop(tl(state))
    end
  end
end
```

> iex.bat -S mix
iex(1)> pid = spawn(Stack, :create, [])
iex(2)> send pid, {:push, 10}
iex(3)> send pid, {:push, 15}
iex(4)> send pid, {:push, 20}
iex(5)> send pid, {:pop}
GenServer

› Provides an high level abstraction for implementing the behavior of processes

› Relies in the same design principle as the previous example
  – Internally, it maintains a loop with tail recursion
  – Uses also the message exchange mechanism (e.g. receive and send)
  – Provides a error handling and other housekeeping operations (e.g. creation, stopping, hibernating, etc.)
A stack implementation using GenServer

defmodule Stack do
  use GenServer

  def handle_cast({:push, value}, state) do
    {:noreply, [value | state]}
  end

  def handle_call({:pop}, _from, state) do
    {:reply, hd(state), tl(state)}
  end
end
Let us improve the latter code

defmodule Stack do
  use GenServer

  def start_link, do: GenServer.start_link(Stack, [])
  def push(pid, value), do: GenServer.cast(pid, {:push, value})
  def pop(pid), do: GenServer.call(pid, {:pop})

  def handle_cast({:push, value}, state) do
    {:noreply, [value | state]}
  end
  def handle_call({:pop}, _from, state) do
    {:reply, hd(state), tl(state)}
  end
end

> iex.bat -S mix
iex(1)> {:ok, pid} = Stack.start_link()
iex(2)> Stack.push(pid, 10)
iex(3)> Stack.push(pid, 15)
iex(4)> Stack.push(pid, 20)
iex(5)> Stack.pop(pid)
Hands on time

As a way of example, we are going to implement a GenServer that holds the state of a classic paper-and-pencil game: The Hangman
A short introduction to REST web services

› REST stands for Representational State Transfer
  – First described in 2000 by Roy Fielding in his Doctoral Dissertation

› REST is an architecture style for designing networked applications
  – The World Wide Web itself uses a REST-based architecture

› REST is not an standard nor a product
Building blocks

**Hypermedia**
- **HTTP**
- **URI**

**Interaction model**
Hyperlinks as a way for triggering processing

**Interaction protocol**
Methods: GET, POST, PUT, DELETE, etc.
Status codes: 200 OK, 404 Not found, etc.
Metadata: Location, Content format, etc.

**Resource identifier**
e.g. https://courses.cs.ut.ee/2017/asd
## Example of a REST API (Hangman)

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Comments</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/games</td>
<td>Creates a new game</td>
<td></td>
<td>(Header) Location /games/:id (Status code) 201</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Game state as JSON{&quot;feedback&quot;: &quot;h-ll-&quot;, &quot;state&quot;: &quot;playing&quot;}</td>
</tr>
<tr>
<td>GET</td>
<td>/games/:id</td>
<td>Retrieve the current state of a game</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POST</td>
<td>/games/:id/guesses</td>
<td>Submits a new guess (a letter)</td>
<td>As JSON{&quot;guess&quot;: &quot;a&quot;}</td>
<td>(Status code) 201</td>
</tr>
</tbody>
</table>
Elixir’s Plug and Plug.Router

```elixir
defmodule GameRouter do
  use Plug.Router

  plug :match
  plug Plug.Parsers, parsers: [:json], pass: ["application/json"], json_decoder: Poison
  plug :dispatch

  def start_link, do: Plug.Adapters.Cowboy.http(GameRouter, [])

  post "/games" do
    id = UUID.uuid1()
    {:ok, _} = Game.start_link(String.to_atom(id))
    conn |> put_resp_header("Location", "/games/#{id}")
    |> send_resp(201, "Your game has been created")
  end

end
```

Agile Software Development - Elixir processes