The Actor Model, Part Five

CSCI 5828: Foundations of Software Engineering
Lecture 17 — 10/18/2016

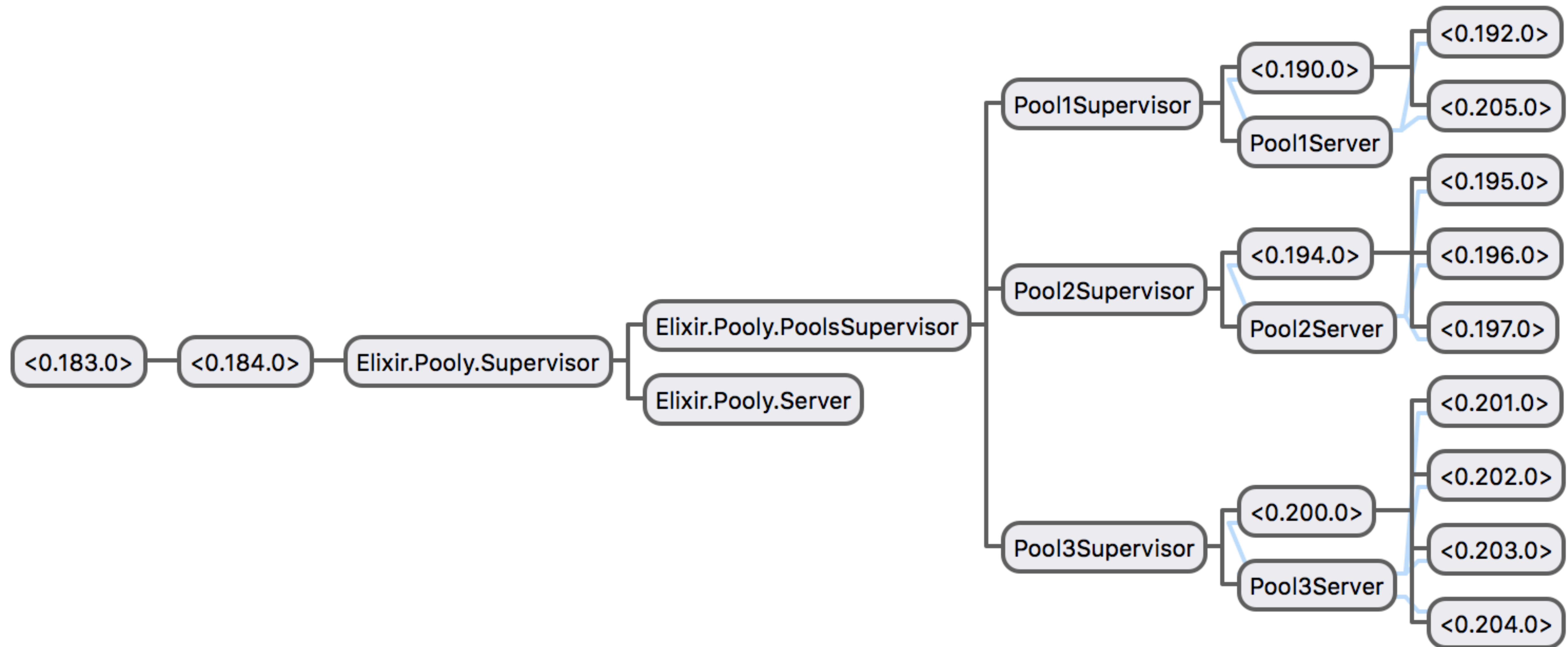
© Kenneth M. Anderson, 2016

Goals

- Cover the material in Chapter 19 of the Elixir Textbook
  - Discuss briefly a more advanced example of using supervisors
  - Introduce the notion of OTP Applications
  - Demonstrate the range of features Elixir provides for applications
    - including configuring, deploying, and upgrading them

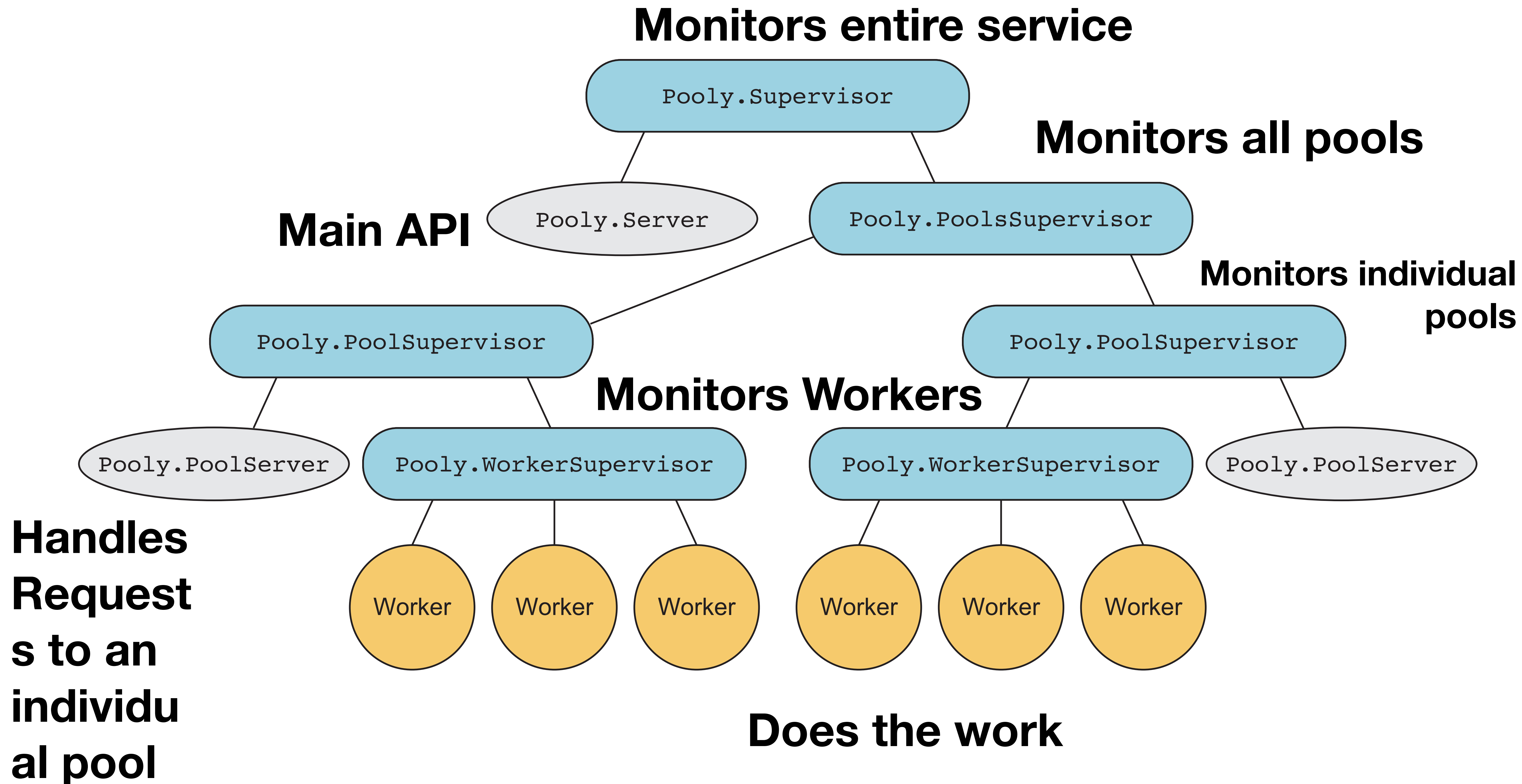
© Kenneth M. Anderson, 2016

The Pooly Application from the Little Elixir Guidebook



© Kenneth M. Anderson, 2016

What does it mean?



OTP Applications (I)

- Elixir had to inherit a bunch of its terminology from Erlang
  - An "application" in Erlang is what we would typically call a "service" or "component" in other settings
- We saw this in action back in Lecture 15
  - The "metex" application that fetched temperatures for cities made use of a 3rd party "application" to make HTTP calls on a web service
    - it was called httpoison
    - behind the scenes, it depended on other "apps"

© Kenneth M. Anderson, 2016

OTP Applications (II)

- We configured metex to depend on httpoison by configuring our mix.exs file

```
  def application do
    [applications: [:logger, :httpoison]]
  end

  defp deps do
    [
      {:httpoison, "~> 0.8.0"},
      {:json, "~> 0.3.0"}
    ]
  end
end
```

- Here, we ensure that :httpoison will be downloaded from hex.pm and that it will be started before "metex" is started

© Kenneth M. Anderson, 2016

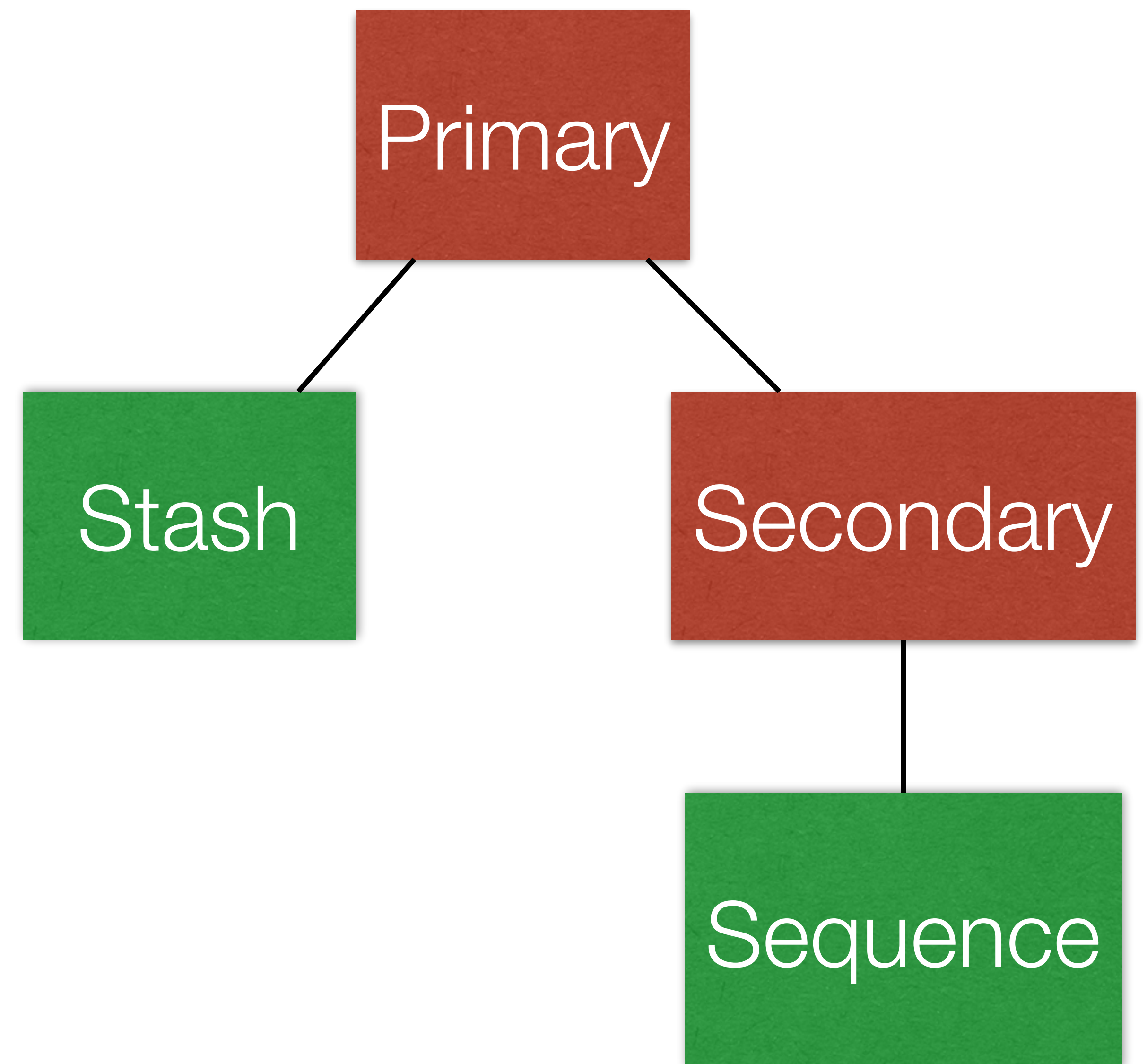
OTP Applications (III)

- When the Erlang VM runs an OTP application, it reads a file called the *application specification* to learn what it needs to do
    - Each time, we modify mix.exs with new information and then trigger a compile of our application, the application specification gets automatically generated
- We're going to learn about this file and the features that OTP provides for applications (remember: "services") by working through a detailed example
- As a result, let's remember where we are at
    - In the last lecture, we developed a supervised version of the "sequence server". Let's review the supervision tree

© Kenneth M. Anderson, 2016

Our Example from Last Time: Sequence Server

- A top-level supervisor manages the "stash" and a second-level supervisor;
- The second-level supervisor will manage the Sequence server;
  - the sequence server generates a unique sequence of integers for its clients
- In this system, the only catastrophic failure would occur if the top-level supervisor were to crash
  - Otherwise, this system can survive a crash in any of the other three nodes



© Kenneth M. Anderson, 2016

Start of an Application

- When we created the Sequence application last lecture, we put into place information in the mix.exs file to make it a true OTP application
- In particular, our mix.exs file contained the following code:

```
def application do
  [
    applications: [:logger],
    mod: {Sequence, []}
  ]
end
```

- The symbol "mod" is short for "module" and this is a variation of the standard MFA convention (module-function-argument). In this case, we only need MA, that is a module name and the arguments because, the ErlangVM assumes that the given module has a `start()` method that accepts the arguments

Updating the configuration

- Last time, we hardcoded the value "123" into the source code as the first integer generated by our sequence server
  - Let's put that value into our mix.exs file and also specify the globally unique names this particular application generates

```
def application do
  [
    applications: [:logger],
    mod: { Sequence, 42 },
    registered: [ Sequence.Server ]
  ]
end
```

  - We are now passing an argument into `start` (we had to modify `start` to compensate) and we've registered the name of our Sequence server.

© Kenneth M. Anderson, 2016

Updating the App

- To update the app, we need to invoke the following command
  - mix compile
- It outputs
  - `$ mix compile`
  - `Compiling 5 files (.ex)`
  - `Generated sequence app`
- That last line means that the application specification has been generated and/or updated
  - But where does that file live?

© Kenneth M. Anderson, 2016

```
engr2-5-50-dhcp:sequence $ tree
.
├── _build
│   └── dev
│       ├── consolidated
│       │   ├── Elixir.Collectable.beam
│       │   ├── Elixir.Enumerable.beam
│       │   ├── Elixir.IEx.Info.beam
│       │   ├── Elixir.Inspect.beam
│       │   ├── Elixir.List.Chars.beam
│       │   └── Elixir.String.Chars.beam
│       └── lib
│           └── sequence
│               └── ebin
│                   ├── Elixir.Sequence.Server.beam
│                   ├── Elixir.Sequence.Stash.beam
│                   ├── Elixir.Sequence.SubSupervisor.beam
│                   ├── Elixir.Sequence.Supervisor.beam
│                   ├── Elixir.Sequence.beam
│                   └── sequence.app
├── lib
│   ├── sequence
│   │   ├── server.ex
│   │   ├── stash.ex
│   │   ├── subsupervisor.ex
│   │   └── supervisor.ex
│   └── sequence.ex
├── mix.exs
└── test
    ├── sequence_test.exs
    └── test_helper.exs

9 directories, 20 files
```

There it is!

This is the location that the ErlangVM will look for the file. It's in `_build/dev/lib/<app>/ebin/`

What's in this file?

In Erlang, everything is a tuple.

```
{application,sequence,
             [{description,"sequence"},
              {vsn,"0.0.1"},
              {modules,['Elixir.Sequence','Elixir.Sequence.Server',
                        'Elixir.Sequence.Stash',
                        'Elixir.Sequence.SubSupervisor',
                        'Elixir.Sequence.Supervisor']},
              {applications,[kernel,stdlib,elixir,logger]},
              {mod,{'Elixir.Sequence',42}},
              {registered,['Elixir.Sequence.Server']}]}.
```

This tuple contains the application name, its version number, all of its modules, all of its dependencies, its registered names, and the "mod" information on how to start this application. Nice!

© Kenneth M. Anderson, 2016

Releasing Your Application (I)

- One of the reasons that Erlang (and by extension, Elixir) has a reputation for producing reliable, fault-tolerant software is…
  - that its community has thought deeply about
    - how applications are released and
    - how applications are upgraded
  - in between an application needs to be deployed (not our focus today)
- When releasing code there are (at least) two independent version numbers that we need to keep track of
  - the version of our system and the version of its data structures
    - in a release, only one or both of these may change

© Kenneth M. Anderson, 2016

Releasing Your Application (II)

- To release a mix application, we make use of exrm which stands for
  - the Elixir Release Manager
- It is a package that you download like any other Elixir package
  - You just add `{:exrm, "~> 1.0.6"}` to your dependencies
- After you do this, you go to any module that has a data structure that you want to version and you add a special attribute to the module
  - `@vsn "0"`
- Once you have done that, you execute the following commands

```
• mix do deps.get, deps.compile # download exrm and compile it
• mix compile                   # recompile sequence app
• mix release                   # generate a release
```

© Kenneth M. Anderson, 2016

Let's try it!

- We add the dependency and we add the @vsn attribute to the module Sequence.Server.
- When we run the `mix release` command, we see:
  - `$ mix release`
  - `Building release with MIX_ENV=dev.`
  - `==> The release for sequence-0.0.1 is ready!`
  - `==> You can boot a console running your release with` `` `$ rel/sequence/bin/sequence console` ``
- This command creates a directory called `rel` which contain a huge number of files including global scripts that can be used to invoke your release locally; the Erlang runtime so you can launch the app anywhere, the beam files of all your application dependencies, metadata for each individual release, and, finally, your packaged release in a `.tar.gz` archive!

© Kenneth M. Anderson, 2016

The packaged release

```
└── releases
    ├── 0.0.1
    │   ├── sequence.bat
    │   ├── sequence.boot
    │   ├── sequence.rel
    │   ├── sequence.script
    │   ├── sequence.sh
    │   ├── sequence.tar.gz
    │   ├── start.boot
    │   ├── start_clean.boot
    │   ├── sys.config
    │   └── vm.args
    ├── RELEASES
    └── start_erl.data
```

This archive contains our packaged release.

As discussed in the book, since we make use of Elixir, we can ignore all files in the release directory and just deploy this file to our servers!

© Kenneth M. Anderson, 2016

Simulating a Deployment Environment

- Like the book, we are going to simulate a deployment environment
  - Unlike the book, I'm not going to use ssh to login to my local machine!
  - I'll just copy things to a different directory!
- The first thing we need is a place for our code to go, we'll make a directory called deploy
  - We will then copy the release to this directory and unpack it
    - `mkdir ../../deploy # location for deployment independent of releases`
    - `cp rel/sequence/releases/0.0.1/sequence.tar.gz ../../deploy`
    - `cd ../../deploy`
    - `tar xvf sequence.tar.gz`
- This produces a directory that contains a bin, lib, releases, and erlang runtime directories. We can invoke our app: `deploy/bin/sequence console`

  We will leave this app running to demonstrate live upgrades.

© Kenneth M. Anderson, 2016

Next version: update the sequence message

- Currently our sequence server just returns an integer
  - Let's change it to return a string
    - "The next number is 42."
- We change the server's client api:

```
def next_number do
  number = GenServer.call(__MODULE__, :next_number)
  "The next number is #{number}."
end
```

- And, we bump the version number in our mix.exs file to 0.0.2.
  - After we run "mix release", a new .tar.gz file is waiting under the 0.0.2 directory

© Kenneth M. Anderson, 2016

Time to deploy…

- We now copy our new .tar.gz file to a 0.0.2 directory on the "server":
    - `$ mkdir ../../deploy/releases/0.0.2`
    - `$ cp rel/sequence/releases/0.0.2/sequence.tar.gz ../../deploy/releases/0.0.2`
- Now, let's upgrade our running application on the server
    - `$ cd ../../deploy`
    - `$ bin/sequence upgrade 0.0.2`

Time to deploy…

- Let's look at the output
  - `$ bin/sequence upgrade 0.0.2`
  - `Release 0.0.2 not found, attempting to unpack releases/0.0.2/sequence.tar.gz`
  - `Unpacked successfully: "0.0.2"`
  - `Generating vm.args/sys.config for upgrade...`
  - `sys.config ready!`
  - `vm.args ready!`
  - `Release 0.0.2 is already unpacked, now installing.`
  - `Installed Release: 0.0.2`
  - `Made release permanent: "0.0.2"`
- The sequence script that was generated for the initial release has functionality built into it to upgrade the deployed system AND update the running application
  - Go back to the console and execute `Sequence.Server.next_number`
    - "The next number is 45." <= That's amazing!

© Kenneth M. Anderson, 2016

Was there a problem?

- If so, not a problem, just:
    - `$ bin/sequence downgrade 0.0.1`
- And run `Sequence.Server.next_number`
    - 46 <= that's amazing too!
- And, back again
    - `$ bin/sequence upgrade 0.0.2`
- `Sequence.Server.next_number`
    - `"The next number is 47."`
- This functionality is quite impressive. (IMHO.)

© Kenneth M. Anderson, 2016

Discussion

- What we just did is upgrade application code
  - We left our data structures the same
- When a new install has occurred, the ErlangVM runs "old" code until a process explicitly references a module by name
  - When our code said "Sequence.Server" in iex, that triggered the loading of the new code for that process
  - In this way, all client processes on the ErlangVM cleaning finishes executing any code that was "in process" and then switches to the new code
- How do we handle the changing of a data structure?

Migrating Server State (I)

- We're going to change the sequence server to keep track of the current delta and use it each time to update a sequence
- Previously, if we had incremented the sequence by 10, we would get this behavior
  - 42, 43, 44, 54, 55, 56, 57
- After the change, we will get this behavior
  - 42, 43, 44, 54, 64, 74, …
- That is, the delta will apply for each next number until we change it again
  - This requires a change in our server state; we need to keep track of the current delta (and the initial delta value is one)

© Kenneth M. Anderson, 2016

Migrating Server State (II)

- The previous state was:
  - `{ current_number, stash_pid }`
- We'll change it to
  - `{ current_number, delta, stash_pid }`
- (This is different that what the book did.)
- Then to handle the migration, we need to bump the @vsn number to "1" and implement a function called code_change
  - `code_change("0", old_state = { current_number, stash_pid }, _ignore)`
  - This function's job is to return a valid value matching the new state
- Let's look at the code

© Kenneth M. Anderson, 2016

Time to release and deploy

- First, bump the version to 0.0.3
- Then, mix release
- Create new release directory: mkdir ../../deploy/releases/0.0.3
- Deploy: cp rel/sequence/releases/0.0.3/sequence.tar.gz ../../deploy/releases/0.0.3
- Upgrade: bin/sequence upgrade 0.0.3
- Over in iex, these lines appear:
- `23:44:47.036 [info]  Changing state from 0 to 1.`
- `23:44:47.036 [info]  {48, #PID<0.536.0>}`
- `23:44:47.036 [info]  {48, 1, #PID<0.536.0>}`

Try out the new release!

- Sequence.Server.next_number
    - "The next number is 48."
- Sequence.Server.increment_number 10
- Sequence.Server.next_number
    - "The next number is 59."
- Sequence.Server.next_number
    - "The next number is 69."
- Sequence.Server.next_number
    - "The next number is 79."

Amazing!

© Kenneth M. Anderson, 2016

Does our fault tolerance still work?

- Sequence.Server.increment_number "ken"
  - <Lots of angry lines>
- But… Sequence.Server.next_number
  - "The next number is 83."
- Sequence.Server.next_number
  - "The next number is 84."
- The only thing we lose is our "delta" and that's because we are not stashing that value
  - That exercise is left to the reader. :-)

© Kenneth M. Anderson, 2016

Wrapping Up

- This concludes our review of the features offered by Elixir and the Erlang Vm
- We have seen how the Actor Model and OTP allow us to create
  - concurrent applications that make use of all cores on a machine
  - that can also be easily distributed across multiple machines
  - that can also be packaged into "applications", i.e. components/services
    - and those applications can then be sensibly released, deployed and upgraded
      - and with "hot swapping," you can upgrade your production system with no downtime
- Next Up: Single-Page Serverless Web Apps!

© Kenneth M. Anderson, 2016