# Phoenix Framework

Wannes Fransen & Tom Eversdijk

UC Leuven

2021

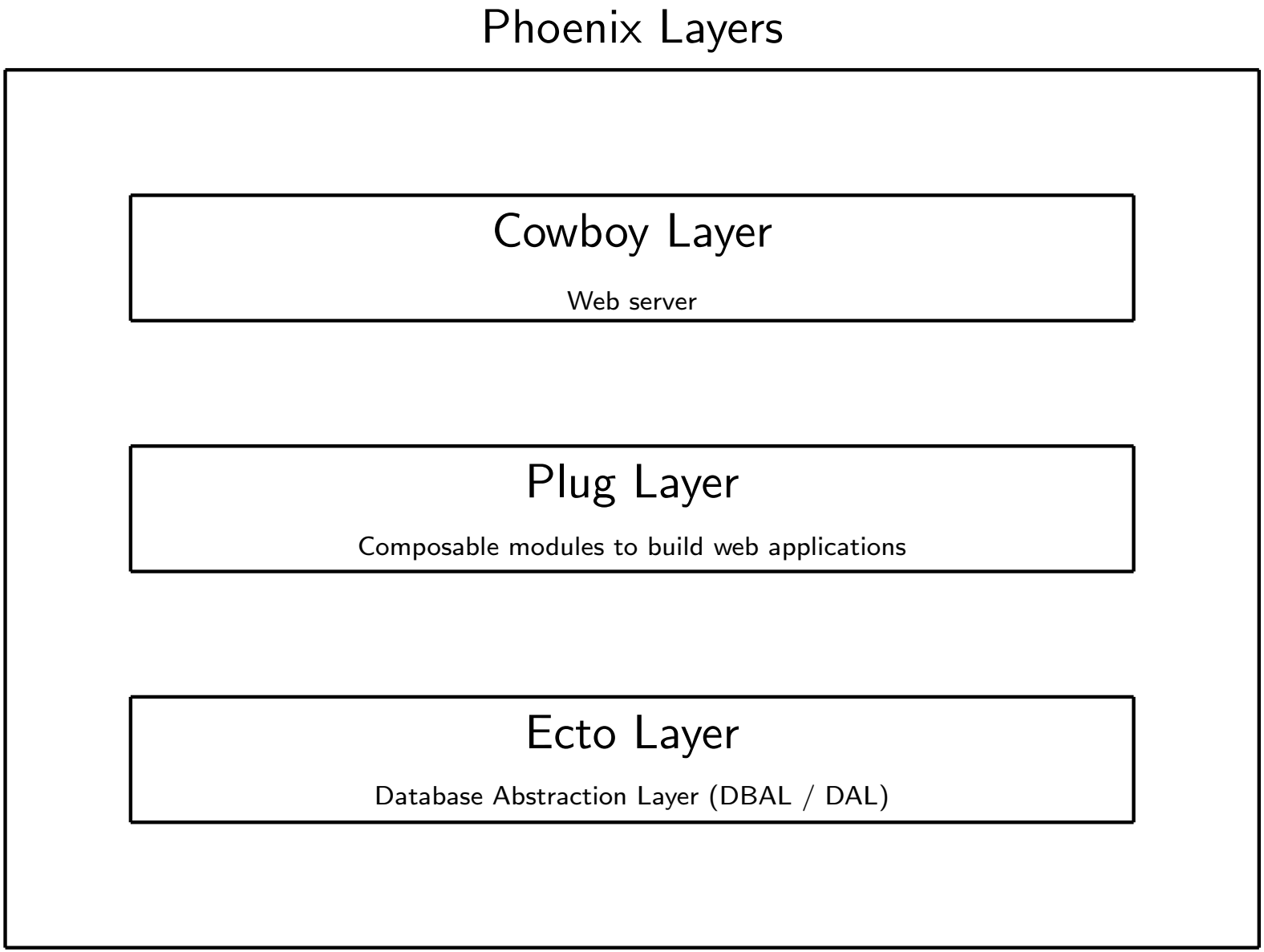
# Why a framework?

- Avoid writing the same code over and over again
- Higher interopability between your code & boilerplate code
- Abstractions e.g. sessions, security, databases, etc...
- Other developers can focus on the important bits

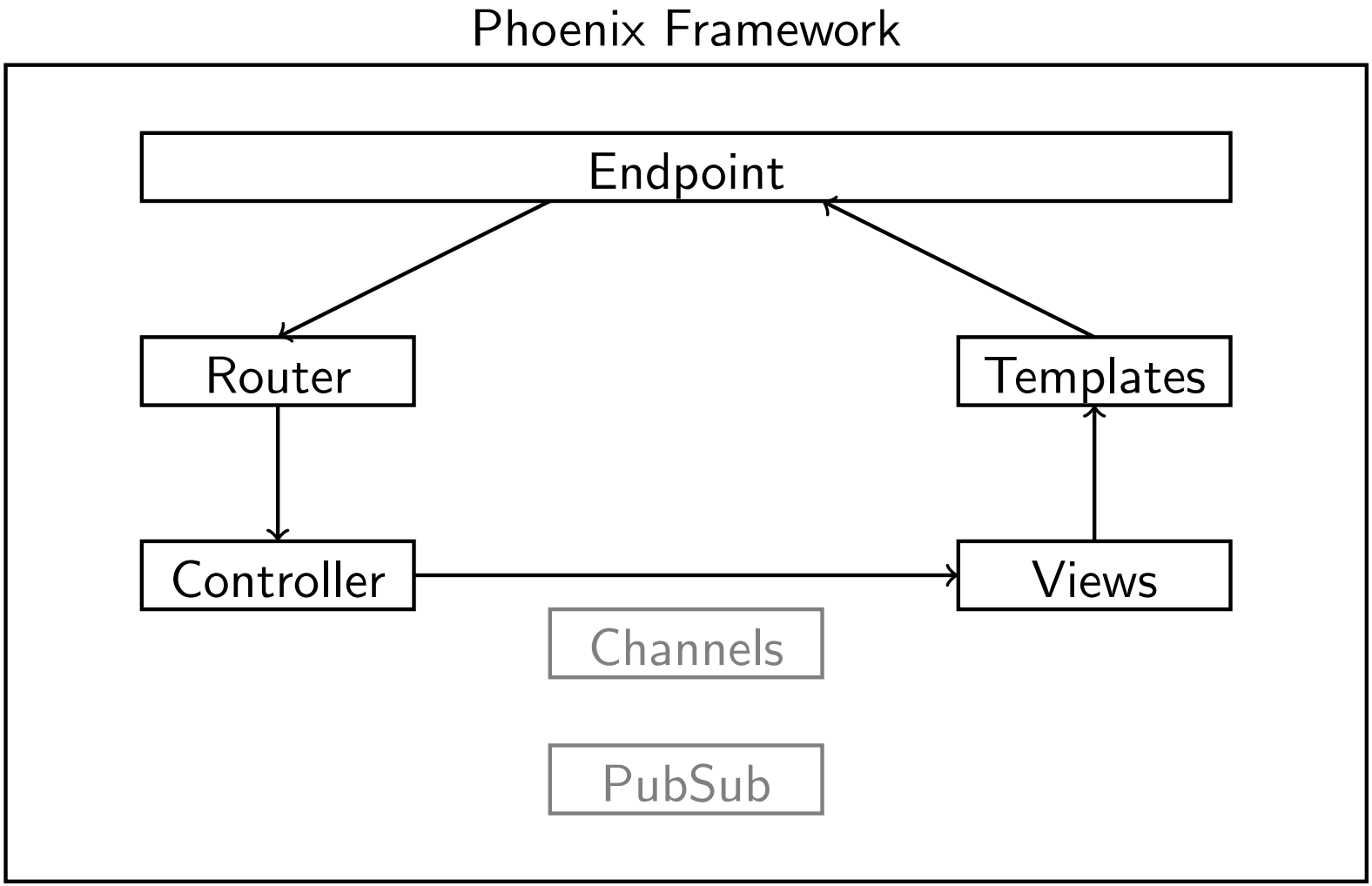
# What is Phoenix?

- Web development framework written in Elixir
- Inspired by concepts of Ruby on Rails
- Server Side MVC framework
- High performance & made for realtime features
- Great language/framework for long connections

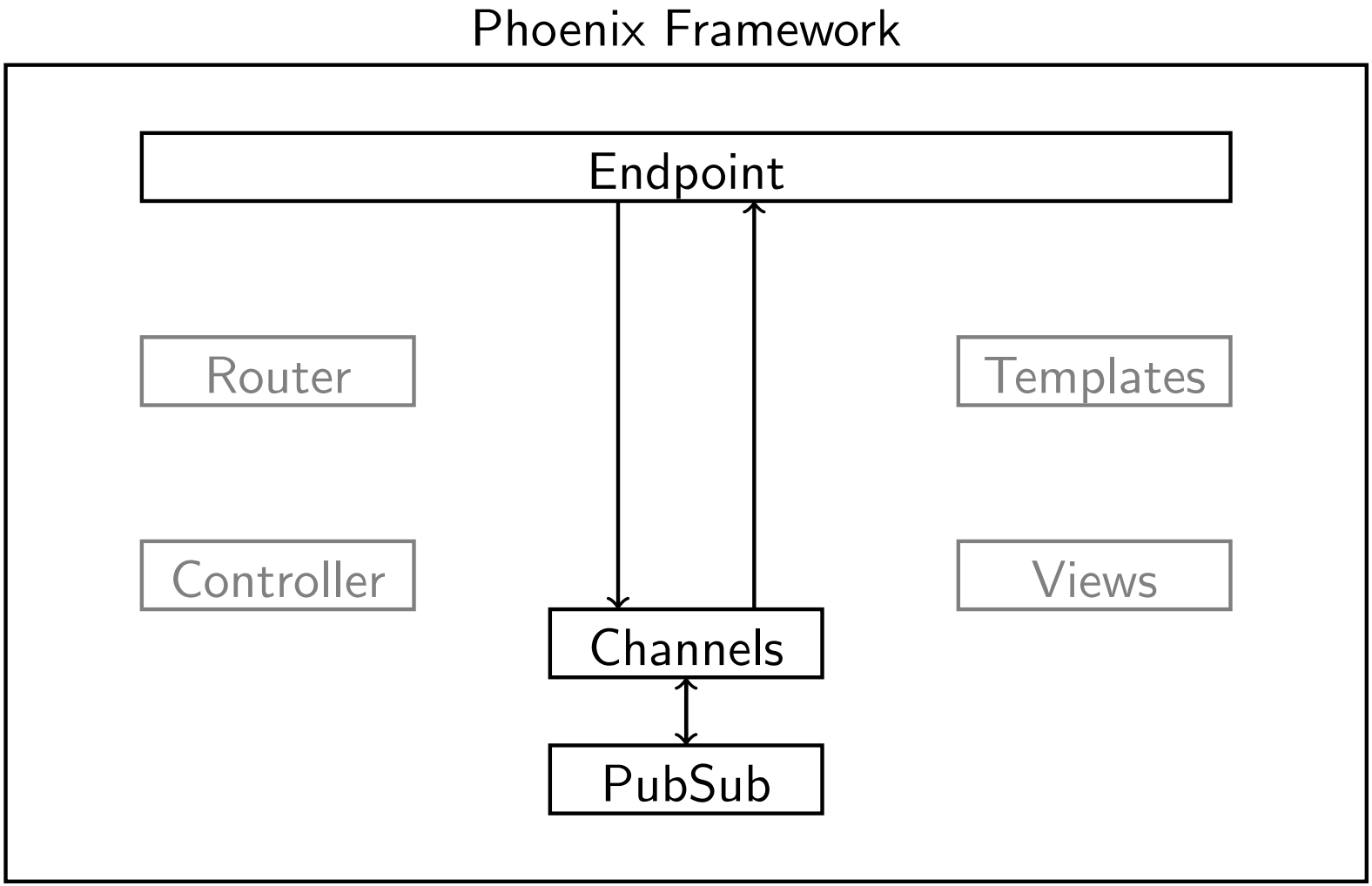
# Phoenix layers



# Request lifecycle



# Websocket communication (soft-realtime)



# Libraries explanation 1/3

## Endpoint

- start and end of the request lifecycle
- all aspects of requests up until the router takes over
- core set of plugs to apply to all requests

## Router

- parses and dispatches requests to the correct controller/action
- helpers to generate route paths or urls to resources
- pipelines - applies groups of plugs to a set of routes

## Controller

- provide functions, called actions, to handle requests
- action: prepare data and pass it into views
- action: invoke rendering via views
- action: perform redirects

# Libraries explanation 2/3

## Views - presentation layer

- render templates
- define template helper functions to decorate data

## Templates

- files containing the contents that will be served in a response
- basic response structure, allow dynamic data to be inserted
- precompiled and fast

# Libraries explanation 3/3

## Channels

- manage sockets for easy realtime communication
- analogous to controllers, but allow bi-directional communication with persistent connections

## PubSub

- underlies the channel layer and allows clients to subscribe to topics

# What is mix?

- Build tool
- Create your application
- compile / test your code
- Dependency management
- Custom commands
- Configuration management
- ...

# Sample usage

- mix new sample_application (- - sup)
- mix phx.new application_name (- - no-html, - - database)

- mix run
- mix test
- mix compile
- mix deps.get
- mix deps.compile

# Basic mix project structure

## Umbrella projects

Instead of building a single large monolith, you can structure your code with multiple isolated contexts.

- poor man's microservices solution
- compiled and run under the same BEAM instance
- Dependencies between applications must be explicitly defined
- Degree of separation, not fully decoupled!

Different configurations in each application for the same dependency or use different dependency versions, then it is likely your codebase has grown beyond what umbrellas can provide.

# Sample generated project - umbrella structure

mix phx.new demo –umbrella –database mysql

```
hello_umbrella
|-- _build
|-- apps
|   |-- hello                  => domain application
|   |   |-- ...
|   |-- hello_web              => web application
|   |   |-- ...
|-- config                     => shared config
|-- deps
```

# Sample generated project - domain structure

```
hello
|-- lib
|   |-- hello
|   |   |-- foo_context      => context folder
|   |   |   |-- foo.ex       => foo-related modules
|   |   |-- foo_context.ex   => context module
|   |   |-- application.ex   => starts app processes
|   |   |-- repo.ex          => module for db operations
|   |   |-- ...
|   |-- hello.ex             => app interface
|-- priv
|   |-- repo
|   |   |-- migrations
|   |   |-- seeds.ex         => default data in your db
|   |   |-- ...
|-- test
|   |-- ...
```

# Sample generated project - web structure

```
hello_web
|-- controllers
|   |-- foo_controller.ex
|   |-- bar_controller.ex
|-- templates
|   |-- foo
|   |   |-- index.html.eex
|   |   |-- ...
|   |-- bar
|   |   |-- index.html.eex
|   |   |-- ...
|-- views
|   |-- foo_view.ex
|   |-- bar_view.ex
```

# General guidelines

- No domain code in controller
- No direct usage of Repo in your web project
- Controllers will use Contexts to communicate with your domain

# Routing

- Match HTTP requests to controller actions
- Nested resources - create nested requests automatically
- Scoped routes - group routes under a common path prefix and scoped set of plugs
- Path helpers - ensure our controllers, views and templates are linking to pages our router can actually handle sure

# Ecto - the concepts

- Repo module - Via the repository, we can create, update, destroy and query existing entries.
- Schemas - used to map data sources to Elixir structs.
- Changesets - way to filter and cast external parameters, as well to validate changes before applying them
- Query - queries written in Elixir syntax with specific DSL. Queries are by default secure, avoiding common problems. These can be created composable / piece by piece

# Ecto SQL - not the same thing!

- This provides functionality for working with SQL databases in Ecto
- Migrations are an example of this

# Setup

- Database is up to you
- Configure your database:
  development: In /config/dev.exs
  production: As environment variables defined in /config/prod.secret.exs
- Use generators such as:

```
mix phx.gen.schema User users \
name:string email:string \
```

- migrate database:

```
mix ecto.migrate
```

# Exercises

- Install NodeJS
- Task 1: Make a new (static) page
- Task 2: Make a new page with a random number on item
- Task 3: Implement CRUD for a single entity [1]
- Task 4: Link 2 entities [1]

Tip: treat .eex files as html

[1] Seen next lesson

# Links

- Phoenix Framework
- Request lifecycle
- Routing
  - Nested resources
  - Scoped routes
  - Path helpers
- Ecto
  - Repo configuration
  - Schemas
  - Changesets and validations