# Phoenix Framework

## Proyecto de Red Social en 7 días

MANUEL ÁNGEL RUBIO JIMÉNEZ

# Phoenix Framework

Proyecto de Red Social en 7 días

**Manuel Angel Rubio Jiménez**

# Phoenix Framework
## Proyecto de Red Social en 7 días

Manuel Angel Rubio Jiménez

**Resumen**

Elixir está siendo una revolución para el desarrollo web de aplicaciones gracias al uso de librerías como Ecto, Plug y sobre todo Phoenix Framework que aúna todas las buenas prácticas recogidas de muchos otros entornos de programación y lenguajes para proporcionar un entorno actual, rápido, tolerante a fallos (***thread-safe***) y escalable. Poco a poco se comienza a oír cada vez más dentro de empresas como Bleacher Report, Remote, Pepsico, Cabify o Financial Times entre otras.

Este libro intenta acercar de una forma práctica cómo desarrollar empleando Phoenix Framework a través del desarrollo de un proyecto de red social. Este proyecto nos permitirá abordar y resolver problemas típicos que podemos encontrar en otros desarrollos así como introducir cada aspecto del framework. Crearemos la lógica de negocio de nuestra red social almacenándola en PostgreSQL, implementaremos controladores para definir el flujo de la aplicación, las vistas y plantillas revisando cómo Webpack puede ayudarnos en esta tarea y la puesta en producción de nuestra solución.

Phoenix Framework ha sido y está siendo noticia dentro del mundo del desarrollo de software tanto en sus inicios soportando 2 millones de conexiones Websocket simultáneas como después con la publicación de LiveView y LiveDashboard que nos permiten desarrollar aplicaciones web con una respuesta mucho más rápida y fluida así como una interfaz donde observer qué sucede dentro de nuestra aplicación web. ¿Te unes a la aventura?

Depósito legal CO-XXX-2021.

ISBN 978-84-945523-9-7





[1] http://books.altenwald.com/elixir-i
[2] https://creativecommons.org/licenses/by-nc-sa/3.0/deed.es

# Sesión 1: Desde el Principio: La Base

*La forma de empezar es dejar de hablar y actuar.*
*—Walt Disney*

Oí una vez a alguien decir que si queríamos hacer una tarta de manzana desde cero, primero tendríamos que crear el Universo. Cuando creamos un sitio web, ¿podemos decir que comenzamos desde cero? Si vamos a crear una red social desde cero, ¿por dónde deberíamos comenzar?

Una de las máximas del desarrollo de software es reutilizar tanto como sea posible. Por lo tanto, no comenzaremos desde cero. Nos basaremos en un conjunto de librerías, frameworks y elementos ya desarrollados. Una arquitectura de soporte para disponer de nuestra red social en poco tiempo.

La elección de las herramientas constituye otro elemento importante en el desarrollo del software. En este caso nuestra elección será ***BEAM*** o Erlang, Elixir y Phoenix ***Framework***. Comenzaremos argumentando en esta sesión el porqué elegimos estas herramientas: sus ventajas o fortalezas y desventajas o debilidades.

También es fundamental que definamos el alcance y los requisitos de nuestra red social, por donde debemos partir y hacia dónde queremos llegar. ¿Qué características debe cumplir? Es importante encontrar nuestros requisitos, analizarlos y así contrastarlos para comprobar si elegimos correctamente nuestra herramienta.

## 1. Requisitos Funcionales

La particularidad de nuestra red social es su orientación al mundo del cine. Vamos a construir una red social donde poder interactuar con otras personas en torno a conversaciones sobre películas. La interacción es muy importante y por este mismo motivo requerimos de un sistema capaz de soportar miles e incluso millones de usuarios conectados, recibiendo y enviando mensajes.

Otras redes como Facebook o Twitter se basan en compartir un texto, una imagen, un vídeo o un enlace. Instagram se centra en compartir fotografías o vídeos. Nosotros lo limitaremos al mundo del cine. Podremos agregar comentarios en fichas de películas o personajes y estas serán vistas por nuestros amigos en su cronología.

Para conseguir usuarios, nuestra red social nos pedirá al darnos de alta un email y una clave. Una vez dentro podremos invitar amigos y buscar

películas a seguir, marcar si queremos verla o si ya la vimos para otorgarle una puntuación. Si escribimos un comentario llegará a todos nuestros amigos a través de su cronología.

Debido al alcance y tiempo planteado para realizar la red social vamos a dejar fuera algunas características como las notificaciones al navegador o algunos elementos más avanzados que puedan surgir. Nos centraremos únicamente en la construcción de los elementos base a la vez que profundizamos en cada aspecto del desarrollo en Phoenix ***Framework*** y Ecto.

Emplearemos Phoenix ***Framework*** 1.5 así como ***LiveView***. En un principio no me había planteado emplear ***LiveView*** pero en últimas reflexiones considero su uso mucho más simple que emplear canales para el desarrollo de interacciones en tiempo real. Crearemos nuestro proyecto en modo umbrella en varias aplicaciones. Esta infraestructura nos ayudará a separar mejor la lógica de negocio de la propia interfaz.

Además, en cada sección habrá notas orientadas a introducir ciertas dependencias que nos ayudarán a mejorar la escritura de nuestro código, agregar cierta funcionalidad o simplificar nuestro código para no alejarnos de nuestra meta: construir una red social en 7 días.

# 2. Organización del Código

Para organizar el código existen muchas soluciones. Bruce Tate y James Gray proponen en un libro titulado Designing Elixir Systems with OTP[1] la organización del código en capas: Data, Functional Core (logic), Tests, Boundaries (processes), Lifecycles (supervisors) y Workers (pools and dependencies). Esta organización de código ha sido estudiada y desarrollada por los autores durante mucho tiempo y merece la pena tenerlo presente para proyectos más ambiciosos. En nuestra red social queda fuera de alcance pero podríamos iterar y agregar esta organización disponiendo de más tiempo.

La organización del código depende de la interpretación y por tanto la forma elaborada por Bruce y James no es la única para organizar nuestro código. En la edición de Code BEAM STO Virtual de 2020, Marcelo Lebre comentó otra forma basada en cuatro patrones para organizar el código[2] diseñada por él y su equipo en Remote. Esta forma se basa en organizar el código en manejadores (handlers), servicios (services), buscadores (finders) y valores (values). A diferencia de las capas propuestas por el método anterior, la fragmentación se lleva a cabo principalmente en la lógica de negocio. Recomiendo echarle un vistazo.

[1] https://pragprog.com/titles/jgotp/designing-elixir-systems-with-otp/
[2] https://codesync.global/speaker/marcelo-lebre/#661four-patterns-to-save-your-codebase-and-your-sanity

Por último y algo más genérico y simple, existe la organización propuesta por Phoenix Framework[3] de organizar el código en esquemas y contextos. Este método es muy simple y sigue la línea de los generadores de código de Phoenix Framework permitiéndonos dejar la lógica de negocio esparcida en los contextos y estos empleando los esquemas para simplificar el acceso a los datos.

Finalmente lo importante es poder acceder a la consola y a través de comandos directamente a los contextos poder realizar cualquier acción que pueda realizarse desde la interfaz. Cualquiera. Si alguna acción no puede realizarse puede deberse a la alta responsabilidad depositada en los controladores y haber movido inconscientemente parte de la lógica de negocio dentro de algún controlador con tal de acelerar en el desarrollo. El problema viene ante cualquier refactorización, actualización o modificación de la lógica de negocio. Si el sistema no está perfectamente desacoplado se hace mucho más complejo cada cambio.

La organización del código de esta forma nos facilita también la construcción de las pruebas. Veremos en cada una de las secciones las pruebas que podemos realizar y las facilidades que nos proporciona Phoenix Framework para cada una de ellas, además de dar consejos e implementar las que considero pueden resultar más útiles.

Para organizar el código que no es parte de la lógica de negocio contaremos con el patrón Modelo, Vista y Controlador (***MVC***). La parte del modelo la implementaremos con los esquemas y contextos como habíamos comentado, mientras los controladores serán implementados a través del código propio de Phoenix Framework tanto para peticiones ***HTTP*** como interacciones de ***LiveView*** por ***WebSocket***. La vista correrá a cargo de las vistas implementadas a través del uso de vistas y plantillas principalmente.

No te preocupes si de momento no has entendido algún aspecto de los elementos mencionados hasta el momento. Iremos en detalle sobre cada uno de ellos en las siguientes sesiones.

# 3. Generando nuestro Proyecto

Para comenzar debemos asegurarnos de tener instalado Erlang/***OTP*** (recomiendo comenzar a trabajar con la versión 23), Elixir (recomiendo 1.11), node.js (recomiendo la versión 14) y el generador de proyectos de Phoenix ***Framework*** 1.5.

[3] https://hexdocs.pm/phoenix/contexts.html

**Nota**

Si tienes dudas específicas sobre Elixir o Erlang/OTP te recomiendo echarle un vistazo a otros libros sobre estos temas concretos en Altenwald Books[4].

La instalación del generador de proyecto de Phoenix Framework se instala con tan solo ejecutar en MacOS o GNU/Linux:

```
$ mix archive.install hex phx_new 1.5.7
```

Este comando proveerá el script ***phx_new*** para el comando **mix** proporcionando los comandos necesarios para la generación del esqueleto de nuestro proyecto.

Teniendo en cuenta los requisitos especificados en la sección anterior de nuestro proyecto necesitamos generar un proyecto umbrella con dos aplicaciones. Una de las aplicaciones será para la lógica de negocio y la otra aplicación contendrá la parte web.

Así mismo, por defecto el generador elige una configuración de base de datos (PostgreSQL) y la instalación del código de interfaz web a través de webpack y node.js. Todos estos elementos pueden cambiarse desde las opciones del comando para evitar instalar los ***assets*** (o elementos del frontal web), o evitar la instalación de Ecto, o permitir Ecto pero con otra base de datos, entre otras opciones.

Ejecutaremos el comando de esta forma:

```
$ mix phx.new --umbrella --live --binary-id vemosla
* creating vemosla_umbrella/.gitignore
* creating vemosla_umbrella/config/config.exs
* creating vemosla_umbrella/config/dev.exs
* creating vemosla_umbrella/config/test.exs
* creating vemosla_umbrella/config/prod.exs
* creating vemosla_umbrella/config/prod.secret.exs
* creating vemosla_umbrella/mix.exs
* creating vemosla_umbrella/README.md
* creating vemosla_umbrella/.formatter.exs
...

Fetch and install dependencies? [Yn]
* running mix deps.get
* running mix deps.compile
* running cd apps/vemosla_web/assets && npm install && ...
```

Durante la instalación y tras generar todo el esqueleto del proyecto nos pregunta si queremos descargar e instalar las dependencias. Damos

[4] https://books.altenwald.com/

al retorno de carro y comienza a descargar las dependencias (**mix deps.get**), después compila las dependencias (**mix deps.get**) y se encarga de los ***assets***.

**Nota**

La opción ***binary-id*** hace posible emplear identificadores basados en UUID (Universal Unique ID o Identificador Único Universal) en lugar de números autoincrementales. Este tipo de dato es una de las recomendaciones que os hago para poder escalar a bases de datos como CockroachDB[5] o FoundationDB[6], o para emplear extensiones como BDR[7] o Citus[8] por mencionar algunas de las opciones disponibles.

El último mensaje nos avisa de que todo ha ido bien y tenemos una serie de comandos a nuestra disposición. Debemos ir dentro del directorio generado para poder emplearlos. Los comandos son:

**mix ecto.create**

Crea la base de datos. Esta acción será necesaria únicamente si queremos que el sistema se encargue de crear la base de datos. Para que pueda hacer esto deberíamos tener un servidor PostgreSQL con un usuario con privilegios para crear bases de datos (ver más en el ApéndiceB, ***Instalación de PostgreSQL***). Veremos en la Sesión 2, ***Alta de Usuarios*** más información sobre la configuración de la base de datos.

**mix phx.server**

Este comando lanzará en primer plano el servidor web con nuestras aplicaciones iniciadas.

**iex -S mix phx.server**

A diferencia del comando anterior, este comando lanza una consola. Dentro de esta consola se lanza nuestra aplicación por lo que podemos interactuar con nuestra aplicación.

Podemos probar a ejecutarlos en la consola para así familiarizarnos con su ejecución y la salida por consola que originan.

[5] https://www.cockroachlabs.com/
[6] https://www.foundationdb.org/
[7] https://www.2ndquadrant.com/es/resources/postgres-bdr-2ndquadrant/
[8] https://www.citusdata.com/

### Sugerencia

Mi recomendación es emplear un editor que permita mantener una consola abierta o poder acceder a esta de vez en cuando para poder comprobar qué va sucediendo. En mi caso suelo dejar en ejecución el último comando con un intérprete de comandos de Elixir activo (***IEx***) y cuando realizo cambios ejecuto la orden **recompile**. Además, lo combino con un navegador donde tener la URL del servidor local cargada. Con esta configuración no solo puedo ver qué sucede en la consola sino también puedo interactuar con la web y ver el resultado de lo hecho hasta el momento.

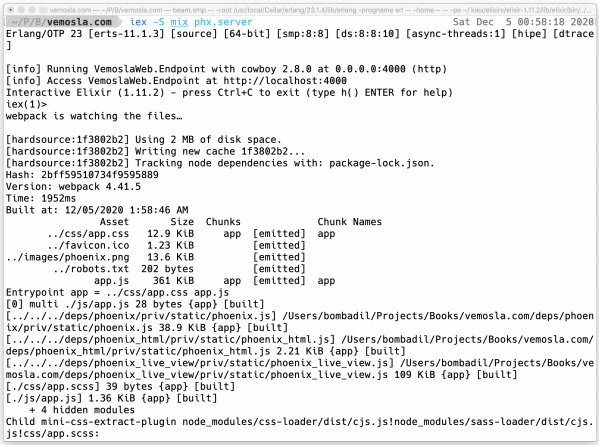
Cuando ejecutemos cualquiera de los comandos veremos la compilación (a menos que se haya realizado anteriormente). Al iniciar el servidor además se realizará la compilación de los assets y finalmente nos muestra la información de ***BEAM***, Elixir y el símbolo del sistema con unas líneas de log sobre la ejecución de Phoenix ***Framework***:

```
Erlang/OTP 23 [erts-11.1.3] [source] [64-bit] [smp:8:8]
 [ds:8:8:10] [async-threads:1] [hipe] [dtrace]

[info] Running VemoslaWeb.Endpoint with cowboy 2.8.0 at
 0.0.0.0:4000 (http)
[info] Access VemoslaWeb.Endpoint at http://localhost:4000
Interactive Elixir (1.11.2) - press Ctrl+C to exit (type h()
 ENTER for help)
iex(1)>
```

Lo más importante es la dirección ***URL*** que nos indica cómo acceder a nuestro endpoint[9] en este caso sirviendo el acceso web en http://localhost:4000.

[9]Endpoint en sistemas informáticos se refiere a un punto que actúa como enlace de comunicación entre dos puntos, normalmente sirviendo de intérprete entre una red de datos y un ser humano.




Accediendo a la web podemos ver la pantalla por defecto proporcionada por el generador de proyectos de Phoenix ***Framework***.

Ahora que ya tenemos control de cómo inciar nuestro proyecto vamos a echar un vistazo a todo el código generado por Phoenix ***Framework*** en las dos aplicaciones: ***vemosla*** y ***vemosla_web***, para la lógica de negocio y para la interfaz web respectivamente.

# 4. Aplicación Web

Accediendo a la aplicación web podemos ver que disponemos de los siguientes directorios:

**assets**

Todos los ficheros estáticos se almacenan en este directorio. Encontramos dentro del directorio un fichero llamado `package.json` donde se indican los paquetes ***JavaScript*** a ser instalados y un fichero `webpack.config.js` para indicar cómo empaquetar los ficheros ***JavaScript***, ***CSS***, imágenes y otros ficheros estáticos que necesitemos para nuestra web para ser empaquetados y llevados a `priv/static`. Veremos en la Sesión 5, ***Plantillas, JavaScript y CSS*** más información sobre los assets.

**lib**

En este directorio encontraremos todo el código a modificar para nuestra interfaz web. Principalmente los controladores, canales,

liveview, plantillas, vistas, la configuración del endpoint y router, internacionalización y telemetry.

**priv**

Los ficheros a incluir dentro de la liberación que no pertenezcan al código en ejecución irán dentro de este directorio. Aquí podemos encontrar los ficheros generados desde assets y los ficheros de las traducciones para la internacionalización.

**test**

Las pruebas desarrolladas sobre los controladores, vistas, liveview o los canales se pueden escribir dentro de este directorio.

Dando una vuelta por el código y las pruebas por defecto vemos ejemplos bastante buenos de cómo comenzar a escribir nuestras pruebas así como nuestro código para la interfaz web. En esta sesión daremos un repaso rápido a cada uno de los elementos encontrados en el código. Más tarde volveremos a ellos con cada elemento que desarrollemos e iremos viendo las posibilidades de cada uno de ellos. De momento los introduciremos de forma general para tener una visión a alto nivel.

## 4.1. Configuración

La configuración se sitúa en el directorio raíz llamado `config` como en todo proyecto umbrella. De cara a la configuración de la interfaz el bloque que más nos interesa es el referente a la configuración del ***endpoint*** tal y como podemos ver a continuación:

```
# Configures the endpoint
config :vemosla_web, VemoslaWeb.Endpoint,
  url: [host: "localhost"],
  secret_key_base: "...",
  render_errors: [view: VemoslaWeb.ErrorView, accepts: ~w(html
 json), layout: false],
  pubsub_server: Vemosla.PubSub,
  live_view: [signing_salt: "x6FI7e/1"]
```

La configuración nos permite definir el dominio con el que se verá nuestra web (en este caso ***localhost***) además de configurar una serie de claves secretas para la firma de cookies y tokens. También configuramos la vista responsable de presentar los errores (***VemoslaWeb.ErrorView***) y el nombre del servidor ***PubSub***.

**Importante**

La configuración de estos datos debe permanecer oculto o debe sobrecargarse mediante el fichero de configuración para producción a modo de no desvelar esta información. Veremos más información sobre aspectos de seguridad en los despliegues en la Sesión 7, ***Lanzamos Nuestra Red Social***.

Dentro del fichero `config/dev.exs` podemos encontrar más configuraciones para nuestro endpoint cuando estamos trabajando en modo desarrollo:

```
config :vemosla_web, VemoslaWeb.Endpoint,
  http: [port: 4000],
  debug_errors: true,
  code_reloader: true,
  check_origin: false,
  watchers: [
    node: [
      "node_modules/webpack/bin/webpack.js",
      "--mode",
      "development",
      "--watch-stdin",
      cd: Path.expand("../apps/vemosla_web/assets", __DIR__)
    ]
  ]
```

Este bloque configura nuestro endpoint para escuchar en el puerto 4000, tal y como vimos en la salida por pantalla de nuestra primera ejecución del entorno. En el bloque también se realizan configuraciones para ayudarnos a desarrollar con más información en caso de fallos. Por ejemplo, ***debug_errors*** nos permite ver con lujo de detalles cada fallo que se produce en la interfaz web, ***code_reloader*** activa la recarga de código ante cambios, desactivar ***check_origin*** nos da mayor flexibilidad al desactivar la configuración de seguridad para ataques ***Cross-Site WebSocket Hijacking***[10] y la configuración de los ***watchers*** permite mantener un proceso de node en ejecución para vigilar los cambios en los assets y realizar el reempaquetado en caso de cambios.

Además podemos encontrar también este otro bloque más abajo en el mismo fichero:

```
# Watch static and templates for browser reloading.
config :vemosla_web, VemoslaWeb.Endpoint,
  live_reload: [
    patterns: [
      ~r"priv/static/.*(js|css|png|jpeg|jpg|gif|svg)$",
      ~r"priv/gettext/.*(po)$",
      ~r"lib/vemosla_web/(live|views)/.*(ex)$",
      ~r"lib/vemosla_web/templates/.*(eex)$"
```

[10] Se verá más sobre seguridad y estos tipos de ataques en la Sesión 7, ***Lanzamos Nuestra Red Social***.

```
    ]
  ]
```

Este fragmento de configuración agrega la funcionalidad de recarga automática en nuestro endpoint. Esto quiere decir que cuando se cambie algo en una plantilla o vista, un archivo de internacionalización, un asset o liveview, se detectará y se recargará la página que estemos viendo en el navegador. Es importante fijarse en los tipos de ficheros que vigila. Si queremos activar la recarga para más tipos de ficheros o desactivarla para ficheros específicos deberemos modificar este bloque.

Esta es la configuración inicial que necesitaremos para el proyecto. A medida que vayamos avanzando hablaremos de otros bloques como la configuración de la base de datos o la configuración de otros elementos necesarios e instalados como dependencias.

Antes de dejar el fichero vamos a agregar un nuevo bloque. Este bloque de configuración nos servirá para los ficheros que subamos a nuestro sitio web. En principio, estos serán únicamente las imágenes del perfil. Vamos a hacer las dos acciones, crear el directorio `files` en el directorio raíz y agregar este bloque al fichero de configuración `config/config.exs`:

```
config :vemosla,
  uploads_files_path: "files",
  uploads_url_path: "/files"
```

Esta configuración nos ayudará a indicar a nuestro endpoint en la siguiente sección donde encontrar estos ficheros.

## 4.2. Nuestro Endpoint

El endpoint es un fichero de código en `lib/vemosla_web/endpoint.ex` de vuelta al directorio de la interfaz web. Usa ***Phoenix.Endpoint*** para definir su comportamiento. A lo largo del cuerpo del módulo podemos ver bloques de configuración. Normalmente bloques de código de ***Plug***[11] como el siguiente:

```
  plug Plug.Static,
    at: "/",
    from: :vemosla_web,
    gzip: false,
    only: ~w(css fonts images js favicon.ico robots.txt)
```

El bloque anterior hace accesible ficheros estáticos disponibles en el directorio `priv/static`. Como veremos en la Sesión 5, ***Plantillas,***

[11] Plug es una librería de Elixir escrita para abstraer y potenciar la funcionalidad de forma agnóstica de servidores web subyacentes. Phoenix ***Framework*** delega en Plug toda la comunicación web incluídos los ***WebSocket***s.

***JavaScript y CSS*** en este directorio es donde se copiarán todos los ficheros depositados en los distintos directorios dentro de `assets`.

A lo largo del fichero podemos ver el por defecto y en orden cada uno de los ***plugs***. Cada uno de ellos tiene una misión concreta. Podemos ver:

**Plug.Static**

Nos permite acceder a los ficheros estáticos, tal y como comentamos anteriormente.

**Plug.RequestId**

Genera un identificador por cada petición entrante o emplea el que llega en la cabecera ***x-request-id***. Este dato es agregado en los ***Logger*** como metadato: ***:request_id***.

**Plug.Telemetry**

Instrumentaliza el flujo de la petición web emitiendo dos eventos, uno al inicio de la petición y otro justo antes de enviar la petición al cliente web.

**Plug.Parsers**

Decodifica el contenido del body (si existe). Phoenix Framework proporciona por defecto tres decodificadores para peticiones ***URL-encoded***, ***MultiPart*** y ***JSON***.

**Plug.MethodOverride**

Sobreescribe el método de entrada por el parámetro de entrada definido en el parámetro ***_method***. Permite desde un formulario enviado como método ***POST*** ser cambiado por el método ***DELETE***, ***PUT*** o ***PATCH***.

**Plug.Head**

Convierte peticiones del método ***HEAD*** en peticiones del método ***GET***.

**Plug.Session**

Gestiona la sesión tomando la cookie de la conexión y poniendo dentro de la conexión la información de la sesión.

**VemoslaWeb.Router**

Nuestro Router es en realidad otro plug hablaremos de él en la siguiente sección.

Podemos modificar nuestro endpoint en caso de necesitar definir nuevos sockets, agregar nuevos plugs o cambiar la configuración de algún elemento ya existente. Los cambios tendrán efecto para todas las peticiones entrantes.

De hecho, tal y como comentamos en la sección anterior vamos a agregar un bloque para los ficheros subidos desde la sección del perfil. Este bloque, al igual que el mostrado más arriba tendrá esta forma:

```
plug Plug.Static,
  at: Application.get_env(:vemosla, :uploads_url_path),
  from: Application.get_env(:vemosla, :uploads_files_path),
  gzip: false
```

Esta sección nos permitirá tener acceso a través de la ***URI*** definida en la configuración a todos los ficheros depositados en el directorio especificado. El directorio podemos especificarlo como ruta relativa o absoluta.

## 4.3. Configurando las Rutas

Una vez le hemos echado un vistazo al endpoint le toca el turno al enrutador. Este fichero se encuentra en `lib/vemosla_web/router.ex` y es donde definimos cada ruta o ***URI*** disponible dentro de nuestra aplicación web. Las definiciones se acompañan de algunas facilidades que nos permiten agregar algunos plugs extra a los ya definidos dentro del endpoint específicamente para grupos de rutas definidas a modo de ***conductos*** o ***pipelines***.

Por defecto encontramos estos dos conductos definidos:

```
  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, {VemoslaWeb.LayoutView, :root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end
```

El primer bloque llamado ***browser*** hace mención a los elementos necesarios para interactuar con un navegador web. El segundo bloque llamado ***api*** indica un uso por parte de otros servidores o clientes automatizados.

Como vemos, cada conducto define una serie de plugs. Estos se agregarán a los definidos anteriormente por el endpoint. Los endpoints

en este caso no son módulos sino átomos. En realidad son funciones definidas dentro del módulo ***Phoenix.Controller***. Nosotros también podemos crear las nuestras propias. Las funciones que encontramos en el controlador para gestionar los conductos son las siguientes:

**accepts**

Indica el contenido aceptado por el servidor. En el caso de ***browser*** el servidor nos aceptará peticiones aceptando contenido ***HTML*** mientras que para ***api*** el contenido aceptado por el cliente debe ser ***JSON***.

**fetch_session**

Retoma la sesión. Veremos este temá en detalle en la Sesión 2, ***Alta de Usuarios***.

**fetch_live_flash**

Procesa los mensajes instantáneos (flash). Estos mensajes pueden generarse en los controladores y serán mostrados o empleados en la siguiente petición al servidor. Veremos su uso en la Sesión 3, ***Agregando Amigos***.

**put_root_layout**

Indica la plantilla ***HTML*** raíz a emplear para renderizar la salida al navegador web. Veremos cómo emplearlo en la Sección4.5, "Vistas y Plantillas".

**protect_from_forgery**

Protección contra los ataques de falsificación de sitios cruzados o ***CSRF***. Esta función activa la funcionalidad. Veremos cómo funciona en la Sesión 2, ***Alta de Usuarios***.

**put_secure_browser_headers**

Agrega cabeceras para mejorar la seguridad en la comunicación con el cliente web. Veremos más en temas de seguridad en la Sesión 7, ***Lanzamos Nuestra Red Social***.

### Sugerencia

En la documentación de los módulos Phoenix.Controller[12] y Phoenix.Conn[13] puedes encontrar más funciones para emplear en los conductos. El único requisito es aceptar como primer parámetro la conexión y obtener como resultado de la ejecución el mismo tipo de dato.

Cada entrada de ruta tiene como objetivo definir una concordancia para la petición entrante de modo que podamos hacer una criba basándonos en el método empleado y la ***URI*** especificada para dirigir la petición al controlador y función correspondiente. Estas rutas suelen incluirse dentro de un bloque de definición de ámbito (***scope***) tal y como vemos a continuación:

```
scope "/", VemoslaWeb do
  pipe_through :browser

  live "/", PageLive, :index
end
```

Vemos que al mismo nivel de la definición de ámbito tenemos la ***URI*** base (/) y la raíz empleada para construir el módulo de cada controlador (***VemoslaWeb***). Vemos dentro del bloque la definición del conducto a emplear (***pipe_through :browser***) y por último una línea indicando qué tipo de petición emplearemos. En este caso vemos por defecto definida ***live*** haciendo mención al uso de ***LiveView*** para la ***URI*** "/".

Si vamos al navegador y accedemos a la ***URL*** base la petición se enrutará al controlador ***VemoslaWeb.PageLive*** y se empleará la función ***index/2***.

Además de ***live*** disponemos de otras funciones para definir los métodos: ***get***, ***post***, ***put***, ***patch*** y ***delete*** principalmente. Disponemos de muchos más elementos que veremos contextualmente cuando los necesitemos.

Vamos a hacer una prueba y adelantar un poco de código agregando en ese mismo ámbito el controlador para la página de información, un ***acerca de (about)***. Este tipo de página será solicitada a través de un método ***get*** y su ***URI*** será ***/about***:

```
scope "/", VemoslaWeb do
  pipe_through :browser

  live "/", PageLive, :index
  get "/about", PageController, :index
end
```

[12] https://hexdocs.pm/phoenix/Phoenix.Controller.html
[13] https://hexdocs.pm/phoenix/Phoenix.Router.html

En esta segunda línea empleamos el método ***HTTP*** de conexión, el nombre del controlador bajo ***VemoslaWeb*** y la función a llamar `index`. Veremos en la siguiente sección cómo definir el controlador que nos ayude a atender esta petición.

## 4.4. Controladores

Los controladores son el siguiente punto en la línea que estamos trazando de una petición entrando en nuestra aplicación. Hemos visto cómo se definen los plugs en el endpoint, la definición de las rutas en el enrutador y cómo cada ruta indica un controlador y función dentro de él.

Por defecto no tenemos creado ningún controlador convencional. La ruta creada por defecto pertenece a un controlador creado para usar ***LiveView***. Estos controladores son especiales porque mezclan la funcionalidad del controlador junto con la funcionalidad de los canales. Estos controladores podemos encontrarlos bajo el directorio `lib/vemosla_web/live`. En ese directorio podemos ver no solo el fichero para controlar la interacción, sino también la plantilla web empleada. Hablaremos de ellos más adelante en la Sesión 6, ***Actualizaciones en Tiempo Real***.

Podemos diferenciar por tanto los controladores en dos grupos: controladores estáticos y controladores dinámicos o ***live***.

En la sección anterior creamos una ruta para un controlador. Ahora procederemos a crear nuestro controlador para atender a esa petición. Solo deberemos crear en el directorio `lib/vemosla_web/controllers` el fichero `page_controller.ex` con el siguiente contenido:

```
defmodule VemoslaWeb.PageController do
  use VemoslaWeb, :controller

  def index(conn, _params) do
    render conn, "index.html"
  end
end
```

Las funciones públicas dentro de un controlador tienen dos parámetros: la conexión (proveniente de ***Plug.Conn***) y los parámetros detectados y obtenidos ya sea desde la ***URL***[14] o desde el cuerpo de la petición decodificándolos previamente.

En este caso la función no está haciendo ningún procesamiento de los datos de entrada. Solo indica que se renderice la página `index.html`

[14]Podemos indicar valores a ser enviados a través de la ***URL***, estos son conocidos como ***query string***, comienzan con un signo de interrogación (?), emplean como separador un signo et o ampersand (&) y entre el nombre de la clave y su valor se separan con un signo de igualdad (=).

a través de la conexión pasada como primer parámetro en la función `render/2`.

Nuestro módulo controlador emplea el módulo ***VemoslaWeb*** pasando la opción ***:controller***. Esta opción nos permite definir código una sola vez y que esté disponible dentro de cada uno de los controladores. Este código podemos encontrarlo en el fichero `lib/vemosla_web.ex` y el bloque interesante para nosotros tiene este código:

```
  def controller do
    quote do
      use Phoenix.Controller, namespace: VemoslaWeb

      import Plug.Conn
      import VemoslaWeb.Gettext
      alias VemoslaWeb.Router.Helpers, as: Routes
    end
  end
```

Gracias a la importación de ***Plug.Conn*** podemos usar funciones de forma directa sin necesidad de escribir el nombre del módulo al que pertenecen, por ejemplo: `render/2`.

Una vez ejecutamos la función `render/2` el código se transfiere a la vista. Veremos en la siguiente sección cómo funcionan las vistas.

## 4.5. Vistas y Plantillas

Las vistas y plantillas son tratadas como un único elemento aunque estan separadas en diferentes ficheros. La vista aglutina un conjunto de páginas o plantillas manteniendo una coherencia con el nombre del controlador y cada plantilla con el nombre de una función dentro de este controlador.

Para que funcione nuestra petición a ***/about*** debemos crear un fichero dentro de `lib/vemosla_web/views` que llamaremos `page_view.ex`. Dentro escribiremos el siguiente contenido:

```
defmodule VemoslaWeb.PageView do
  use VemoslaWeb, :view
end
```

Comprobamos de nuevo el uso de ***VemoslaWeb*** en este módulo también pero esta vez pasando como parámetro ***:view***. El fragmento de código referente a esta opción dentro del módulo ***VemolasWeb*** es el siguiente:

```
  def view do
    quote do
      use Phoenix.View,
```

```
        root: "lib/vemosla_web/templates",
        namespace: VemoslaWeb

      # Import convenience functions from controllers
      import Phoenix.Controller,
        only: [get_flash: 1, get_flash: 2, view_module: 1,
view_template: 1]

      # Include shared imports and aliases for views
      unquote(view_helpers())
    end
  end
```

El código emplea en este caso ***Phoenix.View*** y agrega configuración para localizar las plantillas. También genera los helpers e importa funciones para tenerlas disponibles no solo en la vista sino también en las plantillas.

Como hemos dicho, cada vista creada por nosotros debe tener el nombre del controlador al que pertenece y a su vez dentro del directorio de templates debemos crear un fichero por cada función. El siguiente paso será crear el directorio `lib/vemosla_web/templates/page` y dentro de este directorio el fichero `index.html.eex`. Dentro de este fichero agregamos nuestro código ***HTML*** para mostrar la información de nuestra web. Como prueba podemos escribir únicamente lo siguiente:

```
<h1>Hola mundo!</h1>
```

El fichero será tratado por la librería EEx[15]. Esta librería reconoce unas etiquetas especiales donde podemos insertar código en Elixir permitiendo así una generación dinámica del código ***HTML***. Cada fichero es procesado en tiempo de compilación generando una función estática dentro de la vista. En realidad cada clausula de la función `render/2` disponible tras la compilación dentro de la vista en el módulo ***VemoslaWeb.PageView*** corresponde a un fichero ***HTML***, la diferencia entre uno se realiza por el nombre del fichero que debe generarse. Como vimos en nuestro controlador, el fichero `index.html.eex` se compila y queda disponible como `index.html`.

Llegados a este punto y si hemos creado nuestra plantilla correctamente deberíamos ver el contenido de la misma accediendo a la ruta sin problemas. Pero podemos ver algo curioso. Nosotros no escribimos la cabecera que aparece en esa web, ¿de dónde salen?

Si nos fijamos en el directorio de vistas y plantillas veremos que existe una vista llamada `layout_view.ex` así como un directorio también con el mismo nombre ***layout***. La página de diseño (layout) nos proporciona la posibilidad de incrustar el código de nuestras páginas dentro de una

[15] https://hexdocs.pm/eex/

plantilla general a reutilizar en cada una de las páginas que escribamos. Es un recurso muy útil cuando queremos compartir la estructura básica de meta-información de la página, cabecera, navegación y pie de la página e incluso algunas secciones fijas, mensajes flash o ventanas modales o de carga para la página.

Esta información está por niveles. Debido a ***LiveView*** tenemos la página raíz (root) que aglutina todo lo común y posible de compartir entre todas las páginas y después una página específica para los controladores estáticos (app) y otra para los controladores dinámicos (live).

Veremos más acerca de las páginas de diseño en la Sesión 5, ***Plantillas, JavaScript y CSS***.

# 5. Aplicación para la Lógica

La aplicación a cargo de la lógica de negocio se encarga de la definición del modelo de datos y toda la lógica de qué realizar con esa información y cómo. Por ese motivo tiene como dependencia Ecto[16] y la responsabilidad del acceso a la información a través de una base de datos.

Ecto es una aplicación bastante compleja. En este libro le damos un repaso a través de la creación de usuarios en la Sesión 2, ***Alta de Usuarios***, las relaciones con otros usuarios en la Sesión 3, ***Agregando Amigos*** y mostrando información proporcionada por los usuarios en la Sesión 4, ***Publicaciones en Nuestro Muro*** pero Ecto puede hacer mucho más. Recomiendo revisar la documentación del proyecto y prestar atención a Altenwald Books[17] para futuros lanzamientos hablando específicamente de Ecto.

Volviendo a los requisitos funcionales, en esta aplicación escribiremos cada uno de ellos a lo largo de las siguientes sesiones. Crearemos cada módulo base o contexto donde agregaremos todas las funciones importantes referentes a la funcionalidad requerida y definiremos los esquemas bajo un directorio con el mismo nombre del contexto. Un ejemplo básico y fácil de encontrar en tutoriales y páginas web hablando de Phoenix Framework es la creación de usuarios a través de un contexto con el nombre ***Accounts***. Dentro del directorio con el mismo nombre irían los esquemas ***User*** y ***Contacto***.

Podemos ver una clara diferencia entre los contextos y los esquemas dónde los primeros se definen como nombres en plural y los segundos como nombres en singular.

[16] https://hexdocs.pm/ecto
[17] https://books.altenwald.com

# 6. Resumen

En esta sesión hemos creado nuestro proyecto umbrella. Hemos revisado las aplicaciones para la interfaz web y la lógica de negocio. Vimos cada elemento creado dentro de la aplicación web para atender las peticiones web: endpoint y sus plugs, enrutador, controladores, vistas y plantillas. Además, creamos una nueva ruta, un controlador, una vista y una plantilla. Comprobamos la ejecución de nuestro proyecto y estamos preparados para comenzar a introducir código en nuestra aplicación de código de negocio.

Pero antes de lanzarte a la siguiente sesión te recomiendo revisar estas questiones para asegurarte de haber entendido los conceptos expuestos:

1. Hemos creado una aplicación umbrella, ¿qué beneficios obtenemos?

2. Al crear nuestro proyecto empleamos la opción ***binary-id***, revisando el fichero `config.exs`, ¿sabrías decir qué parámetro indica que está activo o presente?

3. Revisando el archivo de endpoint, ¿qué acciones se llevan a cabo en la conexión de cada petición recibida?

4. ¿Cómo ejecutamos nuestro proyecto para obtener una consola donde poder escribir comandos mientras vemos los logs de cada petición entrante?

5. Si queremos atender una petición de un método ***POST*** en la ***URI /users*** para dirigir esa petición a `VemoslaWeb.UsersController.create/2`, ¿qué debemos agregar en el fichero del enrutador?

6. ¿Qué función empleamos en el controlador para enviar la información ***HTML*** como respuesta?

7. ¿Cuál es el orden de contención entre index, root y app cuando queremos generar la página index?

8. Si queremos provocar una recarga de la web cuando un fichero estático (asset) ha sido agregado o modificado, ¿en qué fichero debemos agregar o cambiar esa configuración?

9. Revisando el cambio hecho en nuestro endpoint agregando la gestión de ficheros dentro de la ruta definida para los ficheros subidos, ¿qué tendríamos que agregar para limitar la subida a 2MB?