
On Fire
WITH
PHOENIX

Level 1 - Section 1

Sparks of Data

Basics of Phoenix and Working with a Database

What Is Phoenix ?

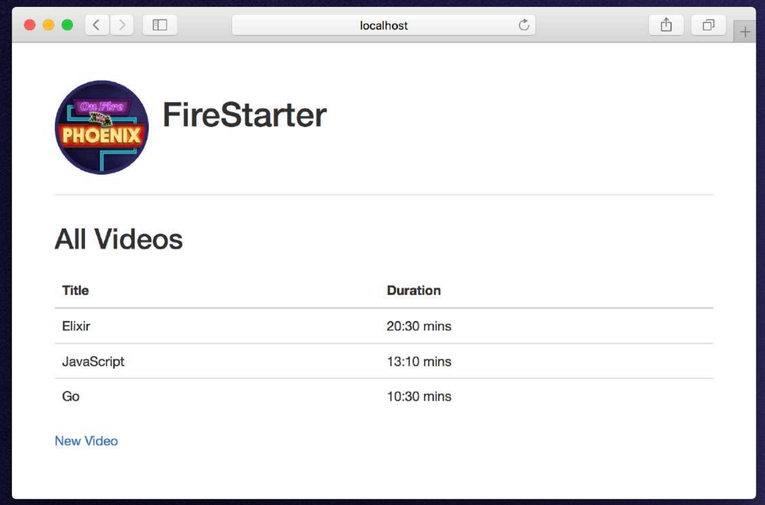
It's a web development framework written in *Elixir*.

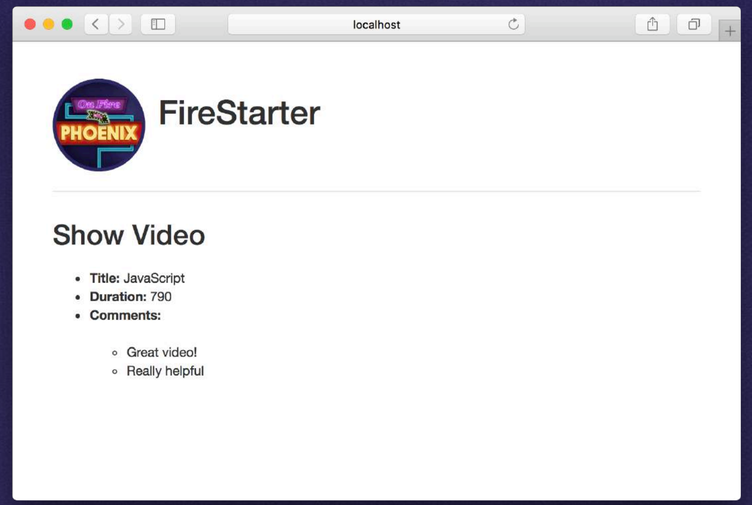
- **Model View Controller (MVC)** - Intuitive structure for code files.
- **Developer Productivity** - Leverages existing *Elixir* and *Erlang* conventions.
- **High Performance** - Runs on the blazing fast Erlang Virtual Machine.
- **Batteries Included** - Full stack, backend code, database access, JavaScript.

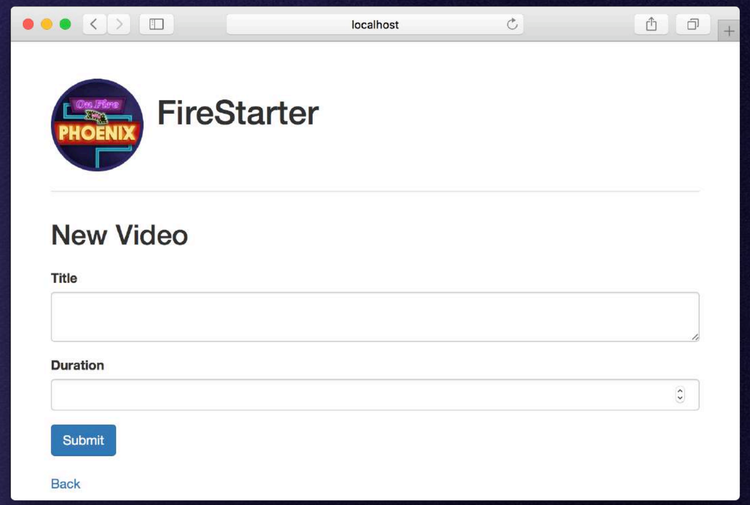
Before proceeding, make sure you know *Elixir* and *SQL*:

What We Will Learn in This Course

In this course, we'll write features for a *Phoenix* web app for viewing videos called ***FireStarter.***





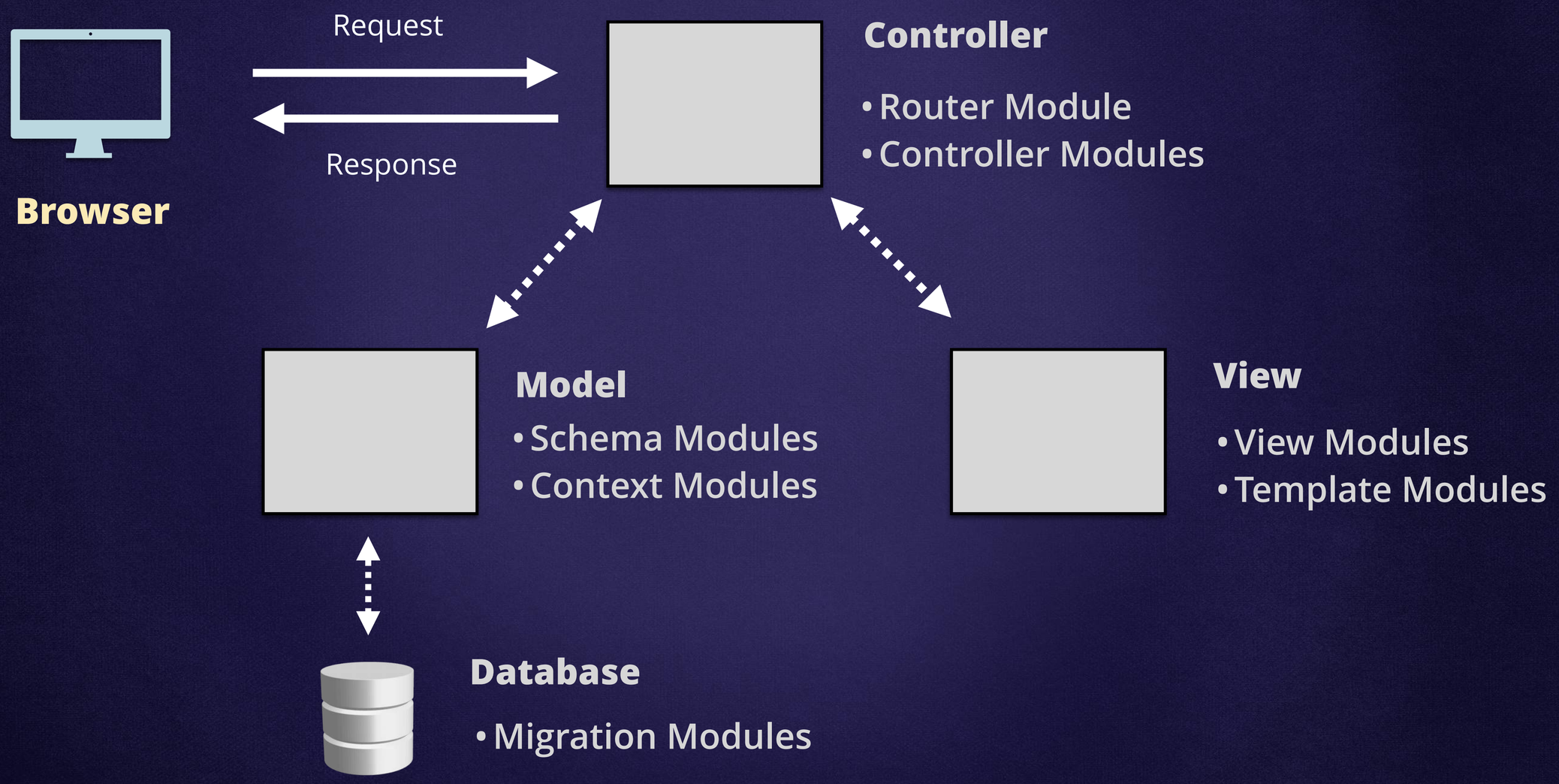


Some of the things we'll learn include:

- Using the **Ecto** library to work with a database.
- Creating new **HTTP routes.**
- Working with **forms.**
- **Validating** user input.

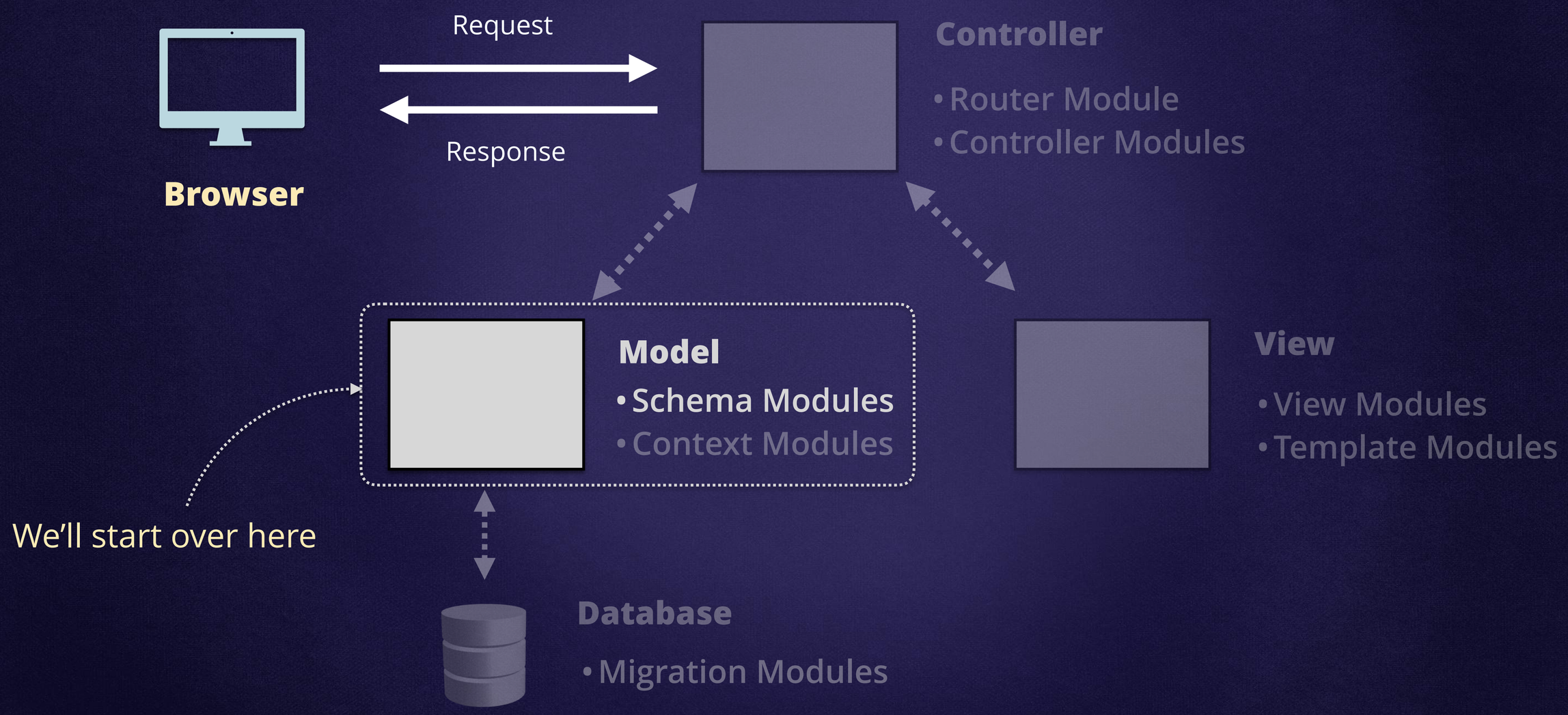
MVC in Phoenix

This is how **Model View Controller** (MVC) is represented in *Phoenix*.



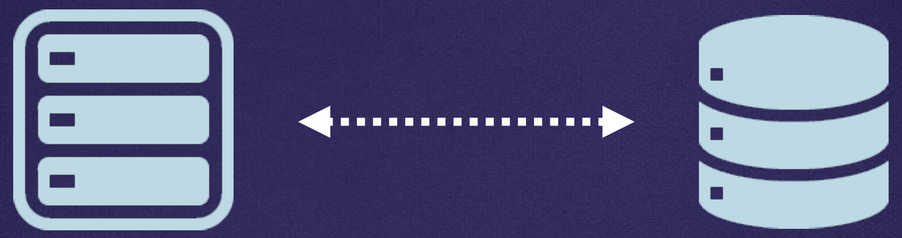
MVC in Phoenix

This is how **Model View Controller** (MVC) is represented in *Phoenix*.



Listing Data

The first thing we'll learn in *Phoenix* is how to read data from a database table.



Phoenix

Database

Defaults to `PostgreSQL` but others are supported.

In this section we'll learn how to:

1. Map *Elixir* code to a database table.
2. List all records from a database table.

The Videos Table

Here's a database table called videos **with 4 columns (**id, title, url and duration**) and two records.**

```
videos

id |       title        |            url              | duration
----+--------------------+-----------------------------+----------
 1 | Elixir             | example.com/elixir          |     1230
 2 | JavaScript         | example.com/javascript      |      790
```

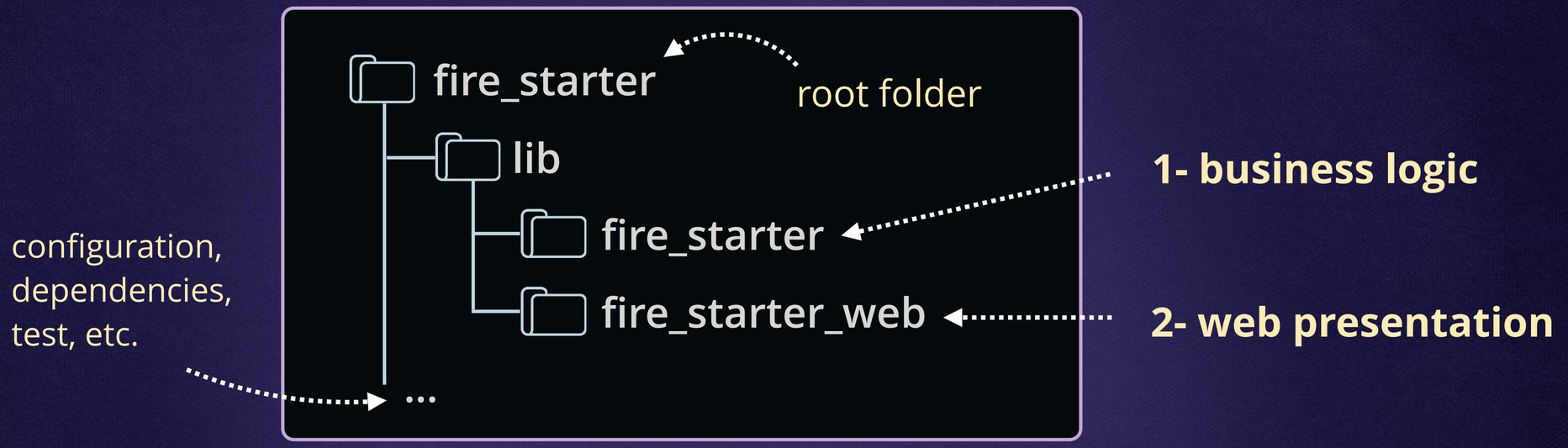
text

integer

On Fire WITH PHOENIX

The Folder Structure

In order to promote **better design**, *Phoenix* organizes our source code in two main folders:



1) The `lib/<name_of_our_app>` folder

The place for **core business logic** of our application, i.e.:

a) Calculating sales tax in a shopping cart
b) Max amount of users in a chat room

2) The `lib/<name_of_our_app>_web`

The place for **web presentation** logic of our application, i.e.:

a) Max number of records per page
b) Error messages on form submissions

An *Ecto* Schema

Schema modules are responsible for **mapping** data sources (usually databases) to Elixir code.

lib/fire_starter/video.ex

```
defmodule FireStarter.Video do



end
```

The top-level module **must** be named after our app name, *FireStarter.*

The module name can be anything, but it's common to use the singular version of the **table name.**

```
videos
...
```

The schema() function

The schema() **function is available from** Ecto.Schema **and maps tables to *Elixir* modules.**

lib/fire_starter/video.ex

```
defmodule FireStarter.Video do
  use Ecto.Schema

  schema "videos" do

  end
end
```

`use` allows us to call functions from `Ecto.Schema` as if they were part of `FireStarter.Video`

The `schema` function takes the name of the database table as its argument.

Elixir

Elixir is mapped to SQL...

...and SQL is mapped to Elixir.

Database

Mapping To Columns

The field() **function takes the name of a database column followed by an *Elixir* data type.**

lib/fire_starter/video.ex

```
defmodule FireStarter.Video do
  use Ecto.Schema

  schema "videos" do
    field :title, :string
    field :url, :string
    field :duration, :integer


  end
end
```

The `field` function takes the name of the column, followed by its data type

The `id` column is automatically inferred as the **primary key** for each table

```
 id |       title      |            url              | duration
----+------------------+-----------------------------+----------
  1 | Elixir           | example.com/elixir          |     1230
  2 | JavaScript       | example.com/javascript      |      790
```

Tracking Creation and Last Update

The timestamps() **function maps to columns that help keep track of when a record was initially inserted and when it was last updated.**

```
defmodule FireStarter.Video do
  use Ecto.Schema

  schema "videos" do
    field :title, :string
    field :url, :string
    field :duration, :integer

    timestamps()
  end
end
```

same thing as...

```
field :inserted_at, :naive_datetime
field :updated_at, :naive_datetime
```

Columns populated by *Ecto* when:

1. a record is **inserted**
2. a record is **updated**

```
 ... |     inserted_at      |    updated_at
-----+----------------------+---------------------
 ... | 2017-05-25 20:27:19  | 2017-05-25 20:27:19
 ... | 2017-05-25 20:27:19  | 2017-05-29 18:13:05
```

automatically populated

Fetching All Records

All communication with the database is done through the FireStarter.Repo module.

The `all()` function takes a *Schema* as argument and returns **all records** in the corresponding table.

```
FireStarter.Repo.all(FireStarter.Video)
```

SQL: `SELECT * FROM "videos"`

Using Alias

The alias **directive lets us refer to** FireStarter.Repo **and** FireStarter.Video **as** Repo **and** Video**.**

```
alias FireStarter.Repo
alias FireStarter.Video
```

creates new alias named after
the last part of the module name

The code that
used to be long...

```
FireStarter.Repo.all(FireStarter.Video)
```

same result

...can now be
written like this!

```
Repo.all(Video)
```

Level 1 - Section 2

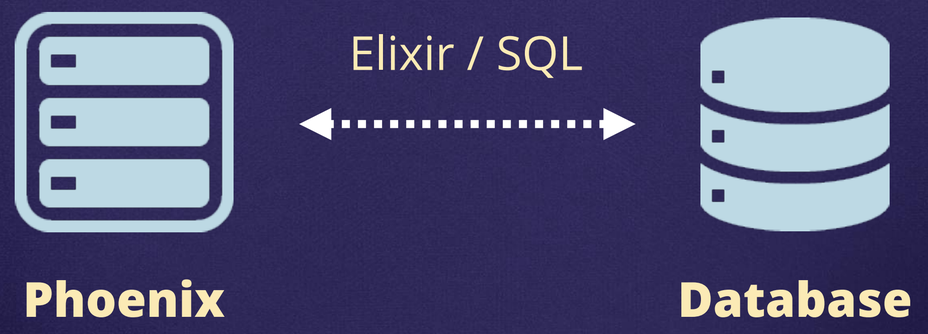
Sparks of Data

Reading Data With Conditions

More Ways to Read Data

We know how to fetch all videos from the database. In this section, we'll learn how to:

- Fetch a single video by its id
- Filter videos based on a condition.



Fetch a Video by id

The get() function takes a **Schema** and an **integer** as argument, and returns a single record.

```
Repo.get(Video, 2)
```

```
%FireStarter.Video{id: 2, title: "JavaScript",
duration: 790, url: "example.com/javascript"}
```

Returns a video *Struct*

SQL:

```
SELECT * FROM "videos"
WHERE id = 2
```

```
 id |        title     |  url
----+------------------+-----
  1 | Elixir           | ...
  2 | JavaScript       | ...
```

A Schema Is a *Struct*

Structs are data types built on top of *Maps* and provide **compile-time checks.**

A *Map*...

...sees `tilte` *(not* `title`*)* and assigns it a value anyway

```
video = %{tilte: "Elixir"}
video.title
```

```
** (KeyError) key :title not found
```

A *Struct*...

...checks that `tilte` was **NOT** defined for `Video` and immediately **raises error.**

```
%Video{tilte: "Elixir"}
```

```
** (KeyError) key :tilte not found in:
%FireStarter.Video{id: nil, duration: nil,
inserted_at: nil, title: nil, updated_at: nil,
url: nil}
```

The keys allowed are the ones defined using the `field()` function in the `Video` **Schema Module.**

get() vs. get!()

The get() **function returns** nil **when no record exists; the** get!() **version raises an error.**

When no record is found...

```
Repo.get(Video, 3)
```

→ `nil` ...no error is raised

```
 id |       title    |  url
----+----------------+-----
  1 | Elixir         | ...
  2 | JavaScript     | ...
```

When no record is found...

```
Repo.get!(Video, 3)
```

→ `** (Ecto.NoResultsError)` ...an error is raised

Filtering Videos based on a Condition

We want all videos where duration is **LESS THAN** 800 seconds.

id	title	url	duration
1	Elixir	...	1230
2	JavaScript	...	790
3	Go	...	630

SQL: `SELECT * FROM "videos" WHERE duration < 800`

Using Ecto.Query

The Ecto.Query **module provides a Domain Specific Language** (DSL) **for querying data.**

```
defmodule FireStarter.Video do
  use Ecto.Schema
  import Ecto.Query

  schema "videos" do
    ...
  end

  def short_duration do
    from v in __MODULE__, where: v.duration < 800
  end
end
```

references enclosing module

Generated SQL

SQL: `SELECT * FROM "videos" WHERE duration < 800`

Running a Query

The all() function can also take an Ecto.Query and it will return a **filtered list of records.**

```
defmodule FireStarter.Video do
  ...
  def short_duration do
    from v in __MODULE__, where: v.duration < 800
  end
end
```

1. **builds** a Query...

```
Video.short_duration |> Repo.all
```

2. ...**executes** the Query.

```
[%FireStarter.Video{id: 2,
  duration: 790, title: "JavaScript", ...},
 %FireStarter.Video{id: 3,
  duration: 630, title: "Go", ...}]
```

Returns a *list* of video *Structs*

alias **vs.** use **vs.** import

alias **helps setup aliases for modules so we can refer to them using shorter names.**

```
alias FireStarter.Repo
alias FireStarter.Video
```

import **allows easy access to functions from other modules without using the fully-qualified name.**

```
import Ecto.Query

def short_duration do
  from v in __MODULE__,
       where: v.duration < 800
end
```

Imported from here

use **is similar to** import**, but gives module authors more control over what is imported and allows for "injecting" code** (metaprogramming)

```
defmodule FireStarter.Video do
  use Ecto.Schema

  schema "videos" do
    ...
  end
end
```

Level 2 - Section 1

Responding with Data

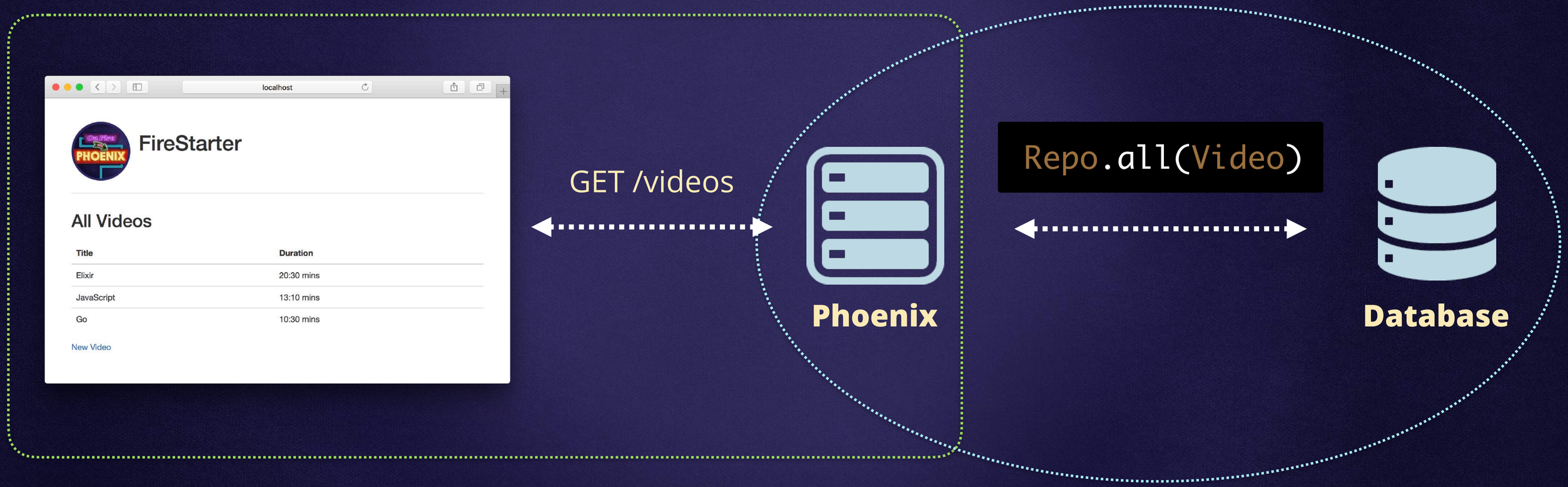
Sending Data Back and Rendering HTML

On Fire WITH PHOENIX

Listing Videos

We can read from the database. Now let's learn how to return this data in a **response.**

Step 1 - Read from database



Step 2 - Return response to client (browser)

The C in MVC

The **Controller** part of **MVC** in *Phoenix* includes **Controller Modules** and the **Router.**

Browser

Request

Response

Controller

- Router Module
- Controller Modules

...now we'll write code over here

We've seen how to write code here...

Model

- Schema Modules
- Context Modules

View

- View Modules
- Templates

Database

- Migration Modules

Routing Requests in the Router

A route is composed of 4 things:

Notice we are at the "_web" folder now

lib/fire_starter_web/router.ex

1. **HTTP method**
2. **URL path**
3. **Controller name**
   (module name)
4. **Action name**
   (a function from the controller module)

```
defmodule FireStarterWeb.Router do
  ...
  scope "/", FireStarterWeb do
    ...

    get "/videos", VideoController, :index
  end
end
```

The *Controller* module name...

...and the function which will be invoked

The index action

Requests to */videos* are routed to the index() **function in the** VideoController**.**

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do


  def index(conn, _) do

  end
end
```

The first argument passed by the router is the **connection.**

The second argument is a *Map* with parameters, but we'll ignore this for now.

Sending text from the Controller

The simplest way to respond with **plain text** from a ***Controller*** is using the text() function.

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  use FireStarterWeb, :controller

  def index(conn, _) do
    text conn, "Hello from VideoController"
  end
end
```

available from here

The **connection** is the first argument...

...and the second argument is a string

From Text to HTML

Now that we know how to send text back to the client, let's see how we can respond with HTML.

Hello From VideoController

What we have now

- Elixir
- JavaScript
- Go

What we want to display

Setting up Data for the HTML Page

To render HTML with video data from the database we can use the render() function. This function takes three arguments: the **connection,** a **template name** and a **Keyword List.**

```
defmodule FireStarterWeb.VideoController do
  use FireStarterWeb, :controller

  alias FireStarter.Video
  alias FireStarter.Repo

  def index(conn, _) do
    videos = Repo.all(Video)
    render conn, "index.html", videos: videos
  end
end
```

available from here

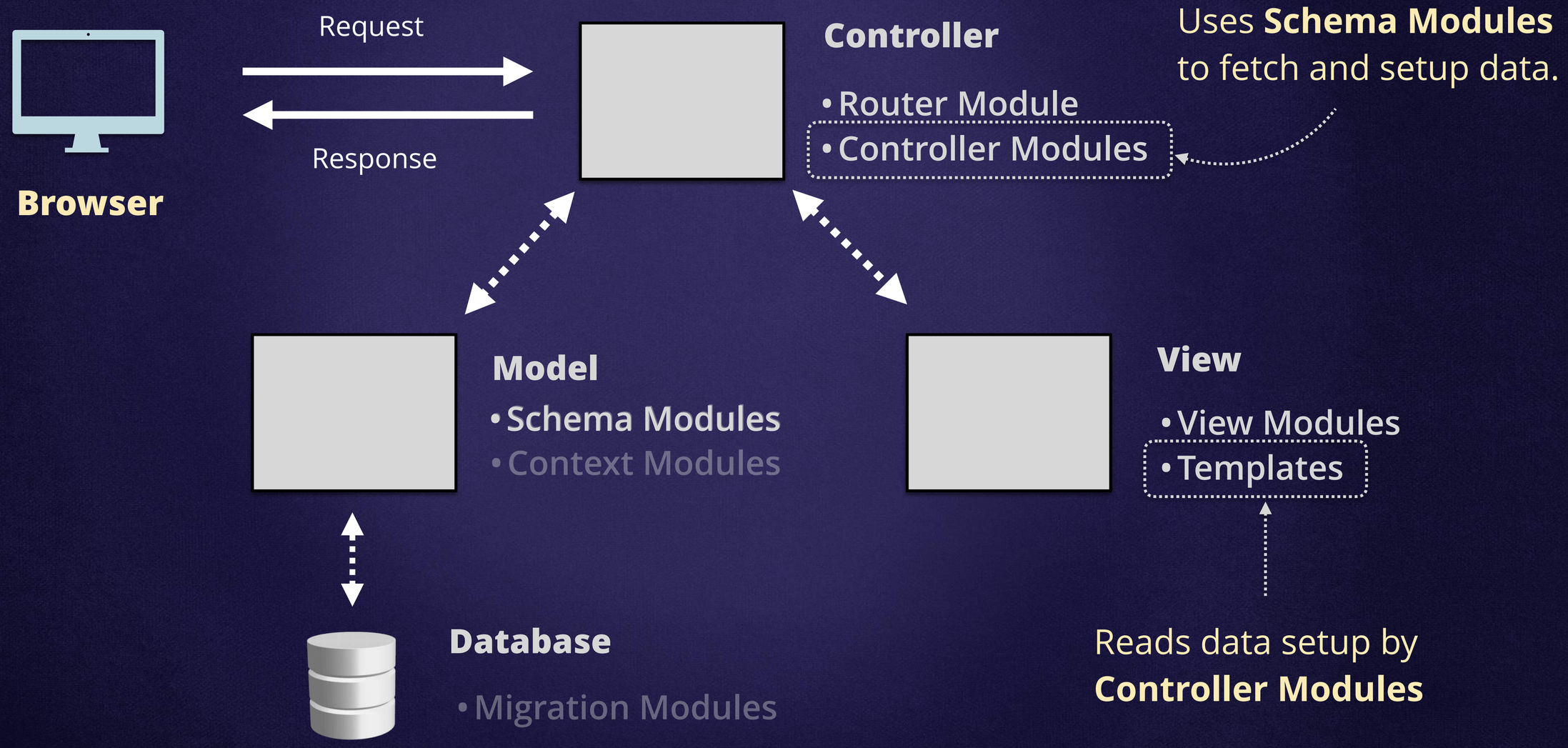
`alias` allows us to use shorter names

template name

made available to the template

Sending Data from *Controller* to *Template*

Controllers use **Schemas** to fetch data from the database which will be read from **Templates.**



The Video *Template*

Templates are files compiled on the server and which output **HTML responses.**

lib/fire_starter_web/templates/video/**index.html.eex**

```
<h2>All Videos</h2>
<ul>
  <%= for video <- @videos do %>
    <li><%= video.title %></li>
  <% end %>
</ul>
```

EEx is the default template system in *Phoenix*

Set from the **Controller**

creates a list of videos using *list comprehension*

List comprehensions are used to loop through enumerables:

```
output = for letter <- ["a","b","c"] do
  "Letter: #{letter} "
end
IO.puts output
```

```
Letter: a Letter: b Letter: c
```

Displaying List of Videos in HTML

Now we are successfully displaying a list of videos in HTML

- Elixir
- JavaScript
- Go duration

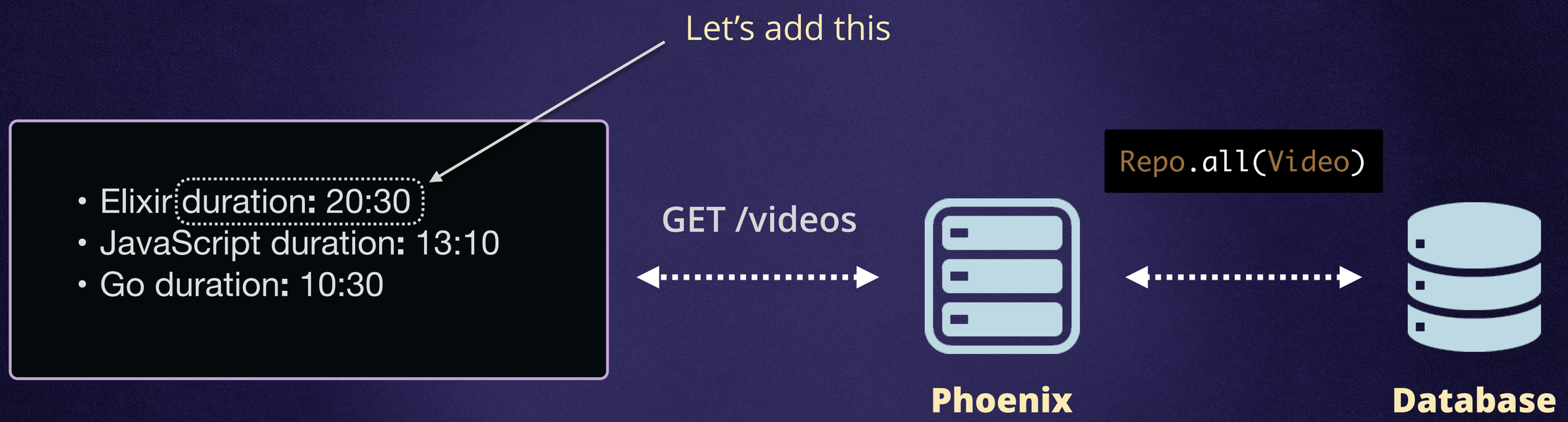
Level 2 - Section 2

Using View Modules

Formatting Data for Templates

Adding *duration* to the list of videos.

We are displaying a list of video titles. Now we want to add the **duration** for each video.



Reading the *duration* property

We can read the *duration* property from each video *Struct* in the template.

lib/fire_starter_web/templates/video/index.html.eex

```
<h2>All Videos</h2>
<ul>
  <%= for video <- @videos do %>
    <li><%= video.title %> duration: <%= video.duration %> </li>
  <% end %>
</ul>
```

Issues with *duration* displayed in seconds

The *duration* is stored in the database as **seconds**. We need to make this easier to read.

- Elixir duration: 1230
- JavaScript duration: 790
- Go duration: 630

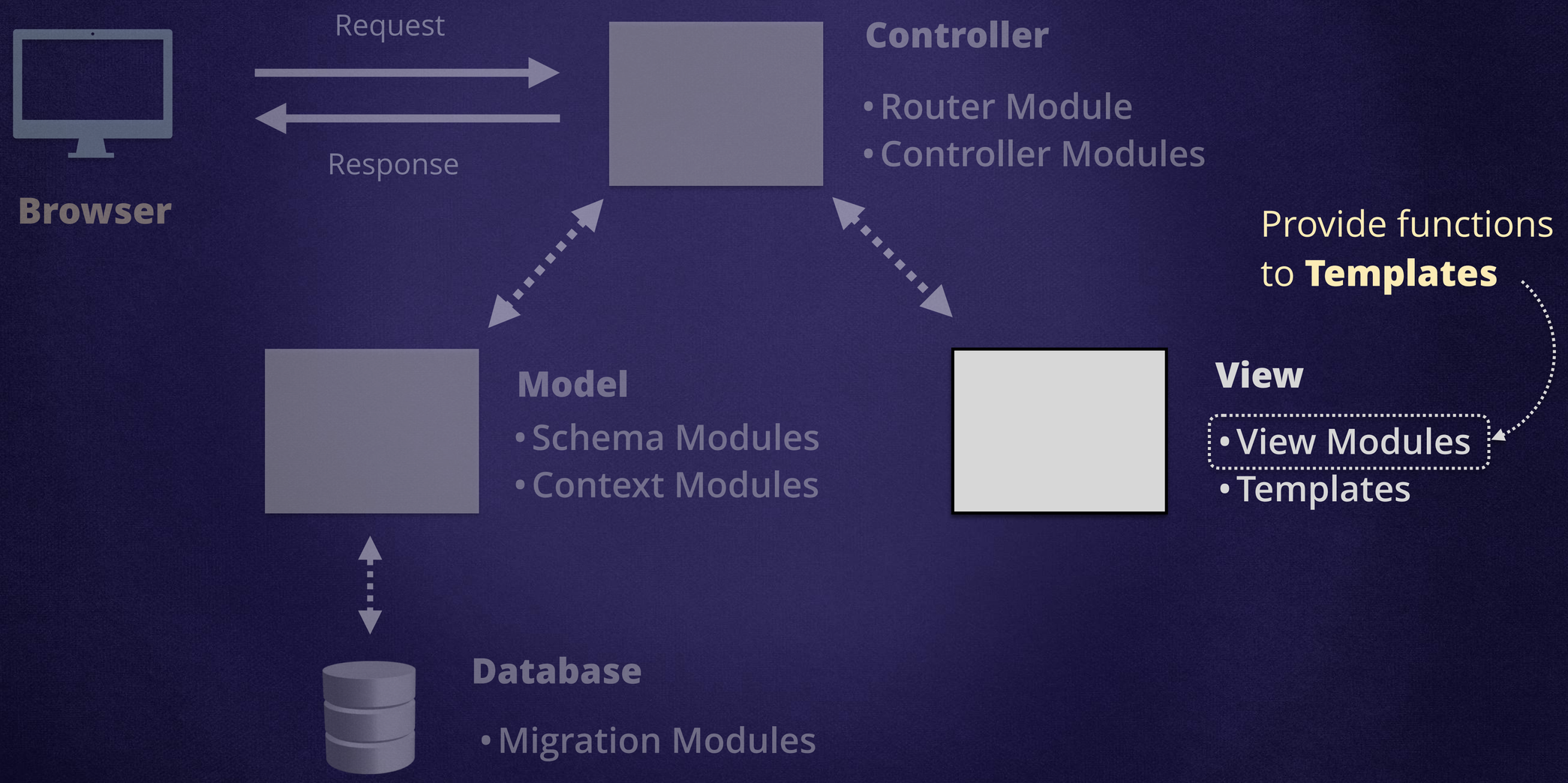
Duration in seconds is hard to understand...

- Elixir duration**: 20:30 mins**
- JavaScript duration**: 13:10 mins**
- Go duration**: 10:30 mins**

It would be much easier to read in this format!

View Modules are the *V* in *MVC*

View Modules are also part of the **V** in **MVC**. They provide **helper functions** for **Templates.**



Functions in the View

Our new helper function expects **one argument** and returns a formatted string.

lib/fire_starter_web/views/video_view.ex

```
defmodule FireStarterWeb.VideoView do
  use FireStarterWeb, :view

  def duration_in_mins(seconds) do
    minutes = div(seconds, 60)
    seconds = rem(seconds, 60)
    "#{minutes}:#{seconds}"
  end
end
```

Performs division and rounds down to the closest integer

Remainder of division by 60

`div` and `rem` are part of the *Kernel* module, automatically imported by Elixir.

Calling a *View* Function

The **Template** can call any function defined in a **View.**

lib/fire_starter_web/templates/video/index.html.eex

```
<h2>All Videos</h2>

<ul>
  <%= for video <- @videos do %>
    <li><%= video.title %>
        duration: <%= duration_in_mins(video.duration) %> mins</li>
  <% end %>
</ul>
```

function is available to the **Template**

The *duration* Is Now Easy to Understand

With the new format, it's easier to understand the duration for each video.

more human readable

- Elixir duration: 8:30 mins
- JavaScript duration: 20:30 mins
- Go duration: 13:10 mins

Level 3 - Section 1

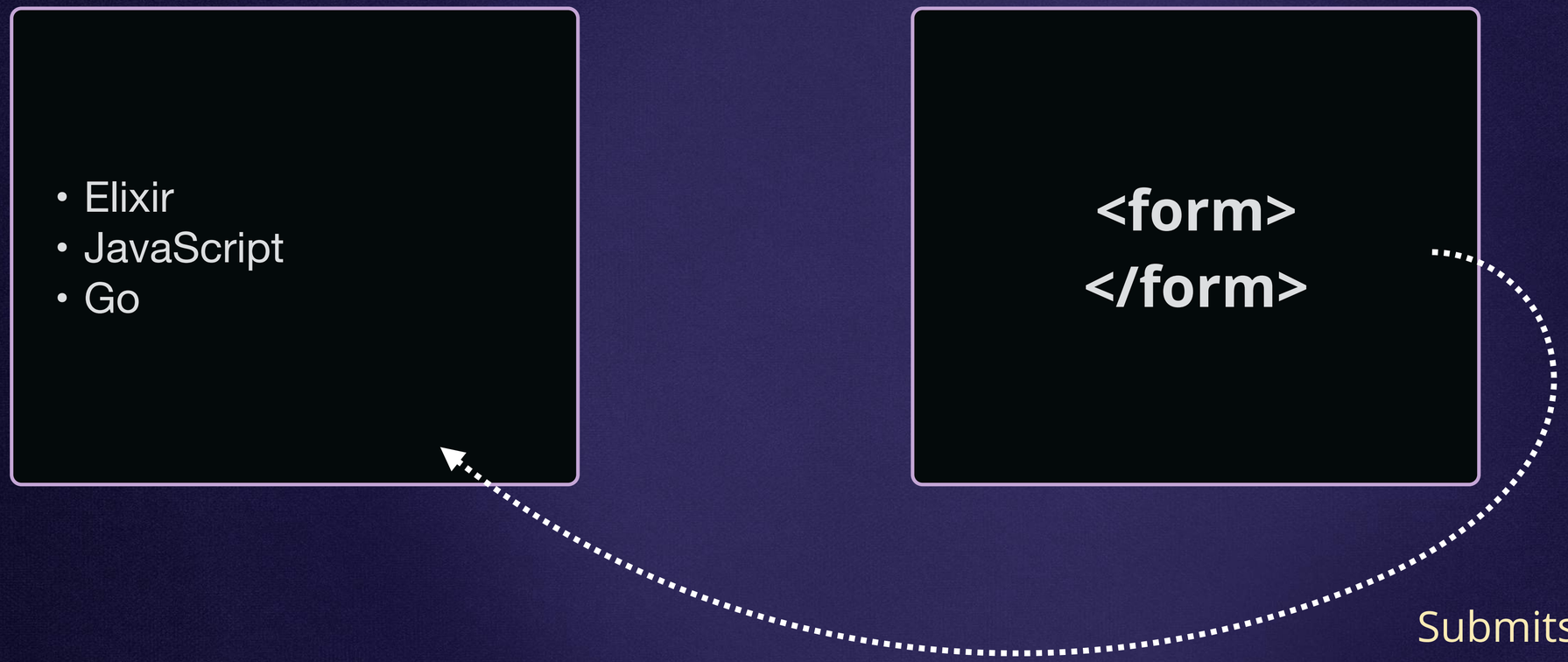
New Records

Using *Elixir* to Output HTML Forms

On Fire WITH PHOENIX

From Listing to Creating

So far we have a page that lists all videos. Now we need a form page so we can **create new videos!**



Steps for Creating New Video Records

Let's write *Phoenix* code that allows users to create new video records. This includes:

1. **Adding new routes to render a form and create a new record**
2. **Using form helpers to generate HTML forms.**
3. **Defining** :create **actions in the** ***Controller***

Requests for Creating a New Video

1 **Render the form**

2 **Handle form submission**

Routes for a New Video

Using the router DSL, we use get() and post(), passing them the **path, controller and actions.**

lib/fire_starter_web/router.ex

```
defmodule FireStarterWeb.Router do
  ...
  scope "/", FireStarterWeb do
    ...
    get "/videos", VideoController, :index
    get "/videos/new", VideoController, :new
    post "/videos", VideoController, :create
  end
end
```

Path Helpers

Path helpers are functions **dynamically generated**, derived from each **controller** in the router.

lib/fire_starter_web/router.ex

```
defmodule FireStarterWeb.Router do
  ...
  scope "/", FireStarterWeb do
    ...
    get "/videos", VideoController, :index
    get "/videos/new", VideoController, :new
    post "/videos", VideoController, :create
  end
end
```

`connection` made available from the *Controller* action

```
video_path(@conn, :index)
video_path(@conn, :new)
video_path(@conn, :create)
```

returns "`/videos`", "`videos/new`" and "`/videos`" respectively

The new action

Requests to "/videos/new**" are routed to the** new() **function in the** VideoController**.**

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...
  def new(conn, _) do


  end
  ...
end
```

ignore second argument

the *connection* is **always** the first argument to controller actions

Creating a *changeset*

A *changeset* is a *Struct* representing changes made to the underlying *Schema.**

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...

  import Ecto.Changeset

  def new(conn, _) do
    changeset = change(%Video{})
    render conn, "new.html", changeset: changeset
  end
  ...
end
```

empty changeset

available from here

will be made available to the template

* Passing an empty changeset to the template helps us build our form fields.

Using form_for **to Create Forms**

The form_for() **function takes a changeset, a url path and an anonymous function.**

template file rendered from the *new* action in the *VideoController*

lib/fire_starter_web/templates/video/**new.html.eex**

```
<%= form_for @changeset, video_path(@conn, :create), fn f ->  %>
  <h3>New Video</h3>


<% end %>
```

helper function returns **"/videos"**

anonymous function

Using text_input to text inputs

The text_input() **function creates an HTM input of type** text **and with the name given as the second argument.**

argument to this function helps generate names for the input fields...

lib/fire_starter_web/templates/video/new.html.eex

```
<%= form_for @changeset, video_path(@conn, :create), fn f ->  %>
  <h3>New Video</h3>
  <p>Title: <%= text_input f, :title %></p>

<% end %>
```

...used as first argument to form input helpers

populates **name** attribute on HTML input with the value "`video[title]`"

The Rest of the Form Fields

The remaining form fields call the same text_input() with their respective *names*.

lib/fire_starter_web/templates/video/new.html.eex

```
<%= form_for @changeset, video_path(@conn, :create), fn f ->  %>
  <h3>New Video</h3>
  <p>Title: <%= text_input f, :title %></p>
  <p>Url:<%= text_input f, :url %></p>
  <p>Duration: <%= text_input f, :duration %></p>
  <p><%= submit "Create" %></p>
<% end %>
```

generates `submit` button

The Form Markup

The form_for() function generates form markup with some added features, like:

- **Protection against *Cross-Site Request Forgery* (CSRF) Attacks**
- **Attributes forcing browsers to use UTF-8 as the charset.**
- **Naming convention for input fields (`module[field]`)**

```
<form accept-charset="UTF-8" action="/videos" method="post">
  <input name="_csrf_token" type="hidden" value="GRgSJipkFiJcKR....">
  <input name="_utf8" type="hidden" value="✓">
  <h3>New Video</h3>
  <p>Title: <input id="video_title" name="video[title]" type="text"></p>
  ...
</form>
```

The First Step Is Done!

1 **Render the form**

2 **Handle form submission**

Level 3 - Section 2

New Records

Reading User Input and Creating New Records

On Fire WITH PHOENIX

Handling Form Submission

1 Render the form

2 Handle form submission

The create Action

The create() function takes two arguments: the **connection** and the **user form data.**

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...
  import Ecto.Changeset

  def create(conn, %{"video" => video_params}) do


  end
end
```

using pattern matching to read user data from a *Map*

always the first argument to *Controller* actions!

Casting Form Data to Expected Types

The cast() **function transforms user input data from String to their corresponding types and filters allowed fields.**

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...
  import Ecto.Changeset

  def create(conn, %{"video" => video_params}) do
    changeset = cast(%Video{},  video_params, [:title, :url, :duration])

  end
end
```

Given this data (empty Video for now)...

...apply this data sent by the user...

...but allow **ONLY** these fields to be populated

Inserting Data

The Repo.insert() function takes a **changeset** as its single argument and translates it to an INSERT SQL statement.

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...
  import Ecto.Changeset

  def create(conn, %{"video" => video_params}) do
    changeset = cast(%Video{},  video_params, [:title, :url, :duration])
    Repo.insert(changeset)


  end
end
```

SQL: `INSERT INTO "videos" ...`

Using case to Pattern Match

We can use the case **statement to check for the return from** Repo.insert() **- responses are either** *{ :ok, record }* **or** *{ :error, changeset }*.

```
defmodule FireStarterWeb.VideoController do
  ...
  import Ecto.Changeset

  def create(conn, %{"video" => video_params}) do
    changeset = cast(%Video{},  video_params, [:title, :url, :duration])
    case Repo.insert(changeset) do
      {:ok, _} ->
      {:error, changeset} ->
    end
  end
end
```

ignoring this value for now

new changeset includes any validation errors

Inserting Data with Success

On successful responses, we use put_flash() to store a flash message for display and redirect() to issue a **301 HTTP response**, redirecting the request to the :index action.

```
...
def create(conn, %{"video" => video_params}) do
  changeset = cast(%Video{},  video_params, [:title, :url, :duration])
  case Repo.insert(changeset) do
    {:ok, _} ->
      conn
      |> put_flash(:info, "Video created successfully")
      |> redirect(to: video_path(conn, :index))
    {:error, changeset} ->
  end
end
```

path helper function returns "`/videos`"

Error When Inserting Data

When inserting a new record is **not** successful, then we render() the form again and pass the **new changeset** including the errors.

```
...
def create(conn, %{"video" => video_params}) do
  changeset = cast(%Video{}, video_params, [:title, :url, :duration])
  case Repo.insert(changeset) do
    {:ok, _} ->
      ...
    {:error, changeset} ->
      conn
      |> put_flash(:error, "Error creating video")
      |> render "new.html", changeset: changeset
  end
end
```

includes **errors** which prevented the insert from being successful

The Complete create Action

This is all the code for the create action in the *VideoController*.

```
...
def create(conn, %{"video" => video_params}) do
  changeset = cast(%Video{}, video_params, [:title, :url, :duration])
  case Repo.insert(changeset) do
    {:ok, _} ->
      conn
      |> put_flash(:info, "Video created successfully")
      |> redirect(to: video_path(conn, :index))
    {:error, changeset} ->
      conn
      |> put_flash(:error, "Error creating video")
      |> render "new.html", changeset: changeset
  end
end
```

It Works for Valid Data

When filling in the form with valid data, it works as expected.

<form>

Title: PHP 101

Url: example.com/php-101

Duration: 100

</form>

Video Created Successfully

- Elixir
- JavaScript
- Go
- **PHP 101**

Submission Errors Are Not Clear

If something goes wrong on the form submission, it's hard to tell where the error is.

<form>

Title: PHP 101

Url: example.com/php-101

Duration: super quick

</form>

Error Creating Video

<form>

Title: PHP 101

Url: example.com/php-101

Duration: super quick

</form>

The form looks **the same.** Could use some help to **spot the error!**

Adding Error Helpers on Template

The error_tag **function generates a tag for input errors, when they exist.**

lib/fire_starter_web/templates/video/new.html.eex

```
...
<p>Title: <%= text_input f, :title %>
  <%= error_tag f, :title %></p>
<p>url: <%= text_input f, :url %>
  <%= error_tag f, :url %></p>
<p>Duration: <%= text_input f, :duration %>
  <%= error_tag f, :duration %></p>
...
```

```
<span class="help-block">is invalid</span>
```

The error tag for a field, when an error exists.

The Rendered Form Displaying Errors

Error Creating Video

<form>

Title: PHP 101

Url: example.com/php-101

Duration: super quick

is invalid

</form>

Now we know where to look!

Two Essential Steps to Create New Videos

1 **Render the form**

2 **Handle form submission**

Level 4 - Section 1

Migrations & Associations

Creating a New Database Table

On Fire WITH PHOENIX

Showing *Video* with *Comments*

Title: Elixir

Comments:

- *Very Helpful!*
- Great video.

We want to show *comments* for each *video*...

...but first we need to create a database **table** for *comments*.

Reference to the **parent** video

```
comments

id |        body              |    author    |  video_id
--+-------------------------+--------------+-------------
 1 | Very helpful!            |    Brooke    |      42
 2 | Great video              |    Sam       |      42
```

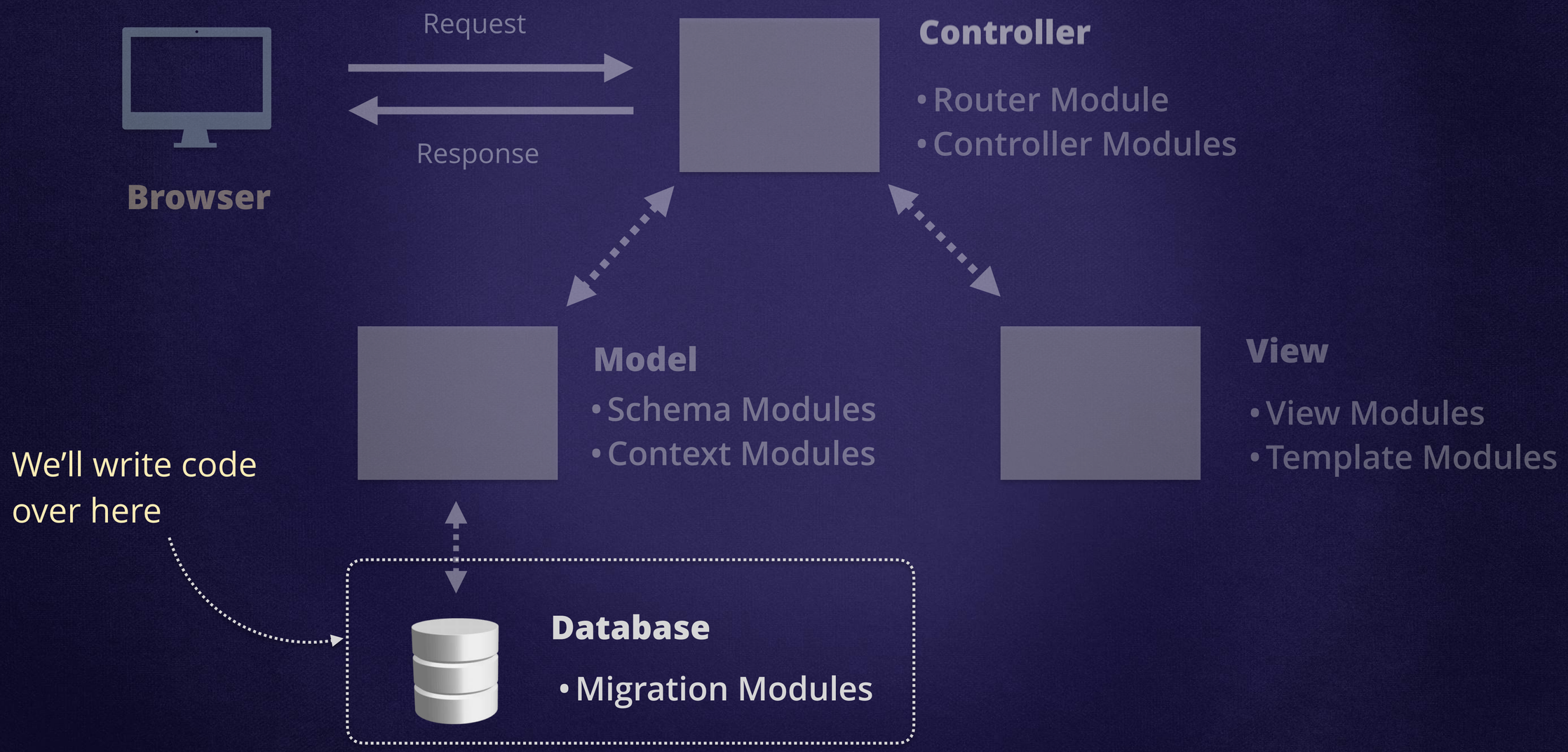
In This Level

We'll learn how to make changes to the database and how to create associations between *Schema Modules*. Things we'll learn include:

- Use **migrations** to create new tables
- Define **foreign key** fields and relationships
- Load **associated** records

Where Migrations Fit

In *Phoenix*, we use **migration files** in order to make changes to the database.



What Are Migrations ?

Migrations are changes to the database structure expressed as *Elixir* code.

Elixir

Database

each and every database change
must be expressed as a migration

A Migration Module

Our migration modules are submodules of the FireStarter.Repo.Migrations **module.**

file creation date helps sort migrations

priv/repo/migrations/**20170523182010**_add_comments_table.exs

```
defmodule FireStarter.Repo.Migrations.AddCommentsTable do


end
```

The change() function

Inside the change() **function we write code that will be translated to SQL statements.**

priv/repo/migrations/20170523182010_add_comments_table.exs

```
defmodule FireStarter.Repo.Migrations.AddCommentsTable do

  def change do

  end
end
```

must be named `change`

code inside this function
will be translated to SQL

Creating a Table

In order to **create** a new table, we can use two functions: create() **and** table().

priv/repo/migrations/20170523182010_add_comments_table.exs

```
defmodule FireStarter.Repo.Migrations.AddCommentsTable do
  use Ecto.Migration

  def change do
    create table(:comments) do



    end
  end
end
```

both functions available
from this module

defaults to a **primary key**
of name `id` and type `serial` 👍

name of the table
to be created

Adding Columns

Also part of Ecto.Migration**, the** add() **function adds a new column to the table.**

column name and type are **required**

```
...
create table(:comments) do
  add :body, :text, null: false
  add :author, :text

  timestamps()
end
```

optional fields

similar to the one used on **Schema Modules**

the same as this

```
add :inserted_at, :naive_datetime
add :updated_at, :naive_datetime
```

Defining a Foreign Key

The references() function is used to define a foreign key to another database table.

```
...
create table(:comments) do
  add :body, :text, null: false
  add :author, :text
  add :video_id, references(:videos, on_delete: :delete_all)

  ...
end
```

deletes all *comment* records when its parent *video* record is deleted

a foreign key to the ***videos*** table is created

Creating a Database Index

The create() **function can be used alongside** index() **to create a database index.**

```
...
create table(:comments) do
  add :body, :text, null: false
  add :author, :text
  add :video_id, references(:videos, on_delete: :delete_all)

  ...
end

create index(:comments, [:video_id])
```

table name

column names **as a list**

Running a Migration

We use the *mix* task ecto.migrate to run migrations and issue changes to the database.

```
>>> mix ecto.migrate
```

```
12:20:24.602 [info]  == Running FireStarter.Repo.Migrations.AddCommentsTable.change/0 forward

12:20:24.602 [info]  create table comments

12:20:24.640 [info] == Migrated in 0.0s
```

Success! 👍

Level 4 - Section 2

Migrations & Associations

Showing *Comments* for a *Video*

On Fire WITH PHOENIX

The *Comments* Table

With the new table in place, we can now start reading *comments.*

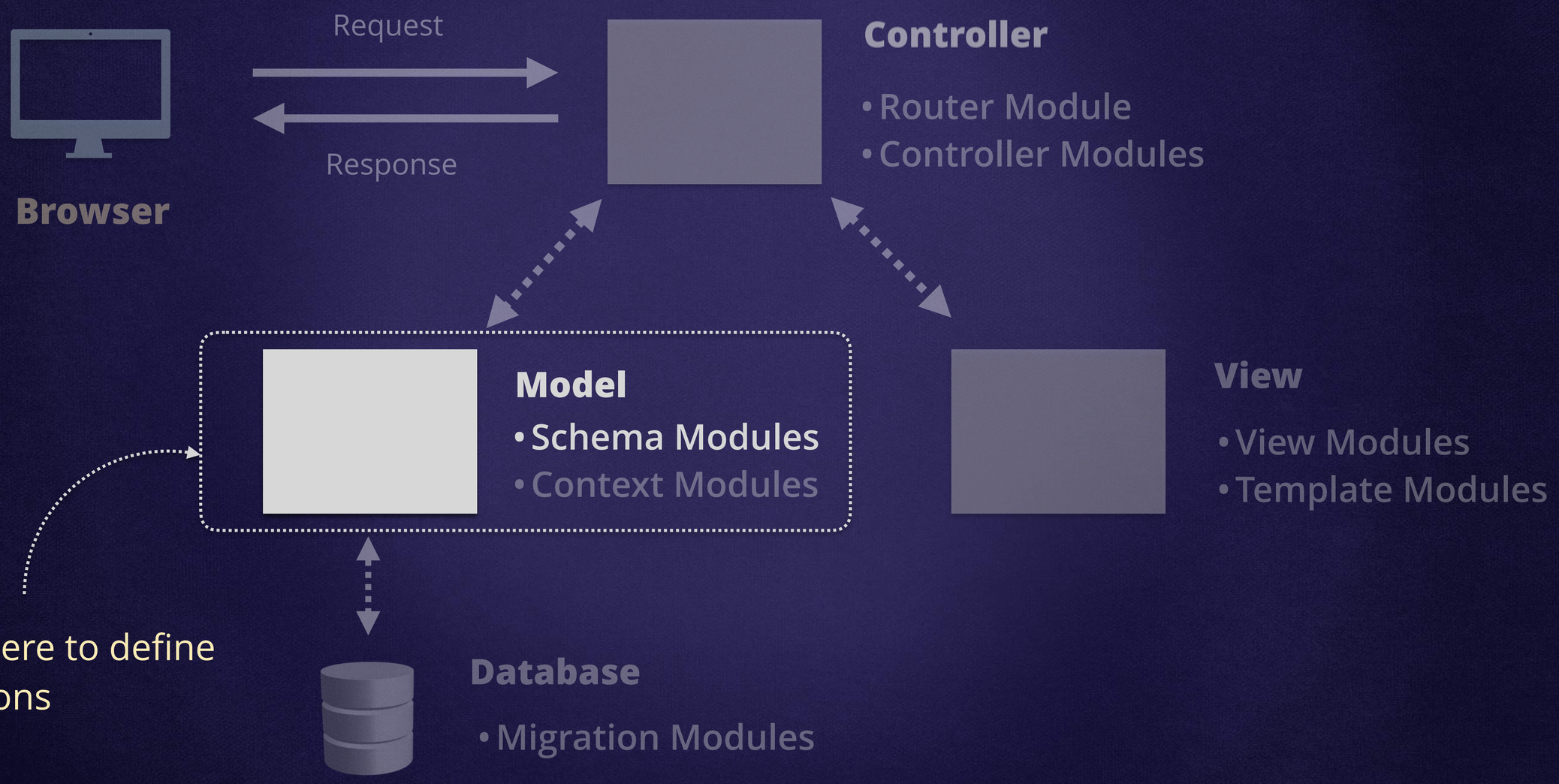
```
comments

 id |        body        |    author    | video_id
--+--------------------+--------------+----------
  1 | Very helpful!      |    Brooke    |    42
  2 | Great video.       |    Sam       |    42
```

```
videos
```

Let's learn how to write *Phoenix* code to read comments that **belong to Videos.**

Schemas Define Associations

We need to tell *Ecto* how our **Schema Modules** are associated with one another.



Back to here to define associations

Adding a has_many association

The has_many function indicates a **one-to-many association** with another schema.

```
defmodule FireStarter.Video do
  use Ecto.Schema

  schema "videos" do
    field :title, :string
    field :url, :string
    field :duration, :integer

    has_many :comments, FireStarter.Comment

    timestamps()
  end
  ...
end
```

the associated module

the property name

"A video has many comments!"

Adding a belongs_to **Association**

The belongs_to **function indicates a one-to-one association between parent and child.**

```
defmodule FireStarter.Comment do
  use Ecto.Schema

  schema "comments" do
    field :body, :string
    field :author, :string

    belongs_to :video, FireStarter.Video

    timestamps()
  end
end
```

the associated module

the property name

"A comment belongs to a video!"

The *Comments* and *Videos* Tables

This is the data that currently resides in our tables.

```
comments

 id |       body        |    author    | video_id
--+-------------------+-------------+----------
  1 | Very helpful!     |    Brooke    |    42
  2 | Great video.      |    Sam       |    42
```

```
videos

id |       title      |          url           | duration
---+-----------------+-----------------------+----------
 42 | Elixir          | example.com/elixir    |     1230
 43 | JavaScript      | example.com/js        |      790
```

Reading *Comments* for a *Video*

Using Repo.get() **does NOT automatically load associations.**

```
video = Repo.get(Video, 42)
video.comments
```

Gets *video*, but **NOT** its *comments.*

```
#Ecto.Association.NotLoaded
<association :comments is not loaded>
```

Preloading Associations

The Repo.preload() **function returns a struct with its associations preloaded.**

```
video = Repo.get(Video, 42) |> Repo.preload(:comments)
video.comments
```

```
[%FireStarter.Comment{..., video_id: 42},
 %FireStarter.Comment{..., video_id: 42}]
```

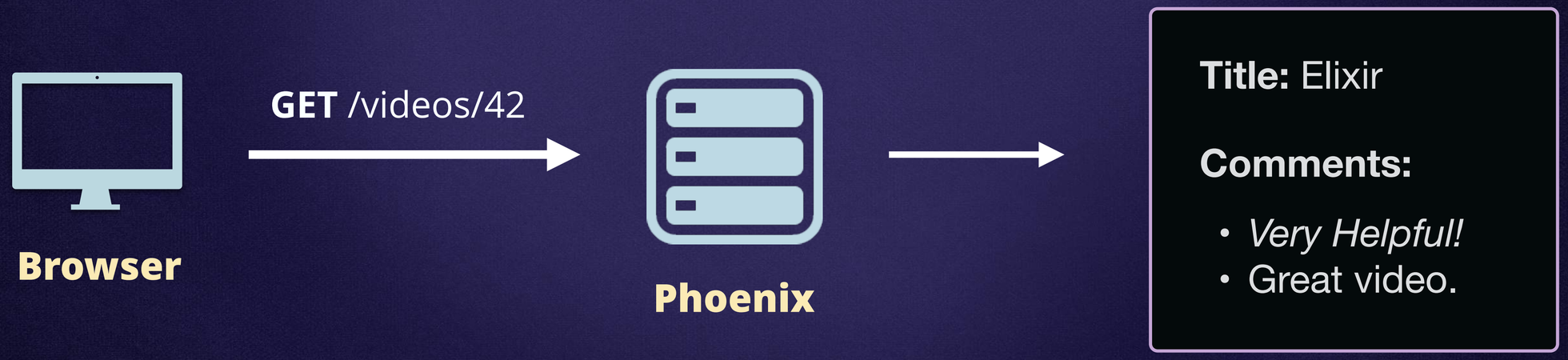
Gets *video* **AND** *comments*

all comments belong to same video

Building the *Video* page

To finish building the *Video* page we need three things:

1. Add a **new route** for the video page.
2. Fetch the video and preload comments.
3. Render the HTML with the video title and list of comments



The Route for a *Video*

The new route will match GET requests to "/videos/" followed by a value.

(spoiler alert: the value is the **id** for the video)

lib/fire_starter_web/router.ex

```
defmodule FireStarterWeb.Router do
  ...
  scope "/", FireStarterWeb do

    get "/videos", VideoController, :index
    get "/videos/new", VideoController, :new
    post "/videos", VideoController, :create
    get "/videos/:id", VideoController, :show
  end
end
```

calls the `show()` function on `VideoController`

The *VideoController* :show **Action**

On the show() function, we use **pattern matching*** to read the **id** from the path.

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...

  def show(conn, %{"id" => id}) do
    video = Repo.get(Video, id) |> Repo.preload(:comments)
    render conn, "show.html", video: video
  end
end
```

the value from `/videos/:id`

* Using pattern matching to read values passed by the router is a widely used practice in *Phoenix*

The *Video* show Template

On the *show* template, we can read the *comments* property from *@video*

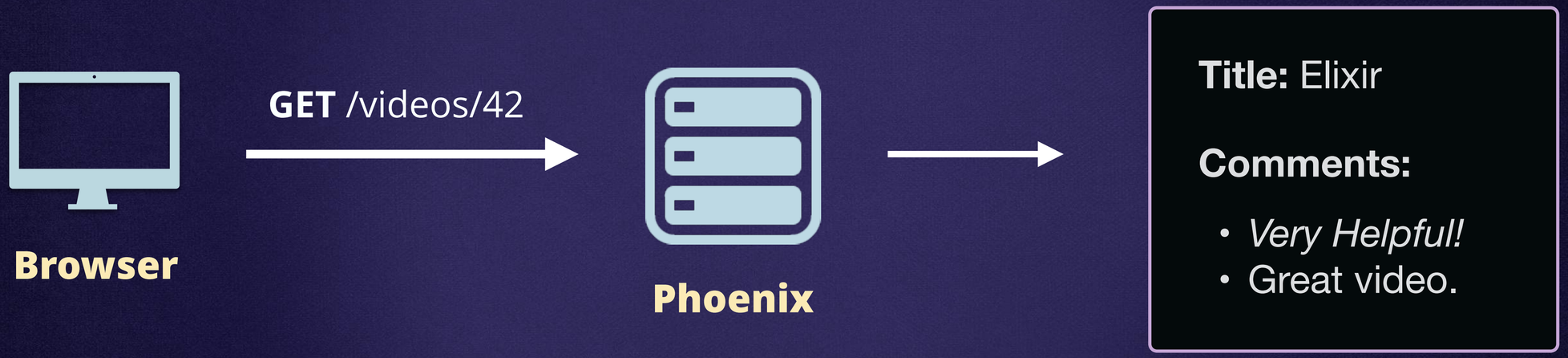
lib/fire_starter_web/templates/video/show.html.eex

```
<h2><%= @video.title %></h2>
<ul>
  <%= for comment <- @video.comments do %>
    <li><%= comment.body %></li>
  <% end %>
</ul>
```

using *list comprehension* to loop through comments

The *Video* page

The video show page is now complete!



Level 5 - Section 1

Using *Contexts*

Reading Videos

On Fire WITH PHOENIX

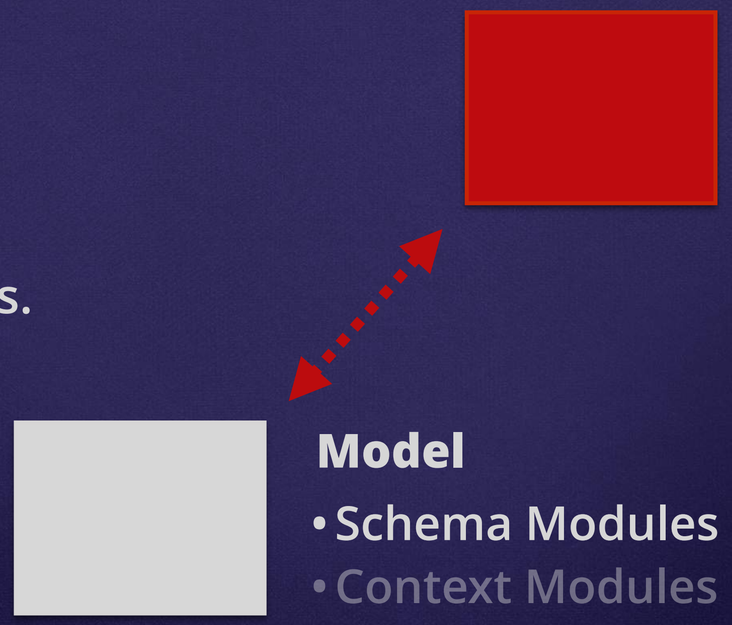
Tight Coupling Leads to Bad Code

When parts of the app **know too much** about other parts, it's called **tight coupling.**

Examples of tight coupling in *Phoenix*:

1. Too much code in *Controllers*, known as "Fat Controllers".

2. References to *Repo* and *Schema* functions from *Controller* actions.



Controller

- Router Module
- **"Fat Controller" Modules**

Lots of code and high dependency on other parts of the app

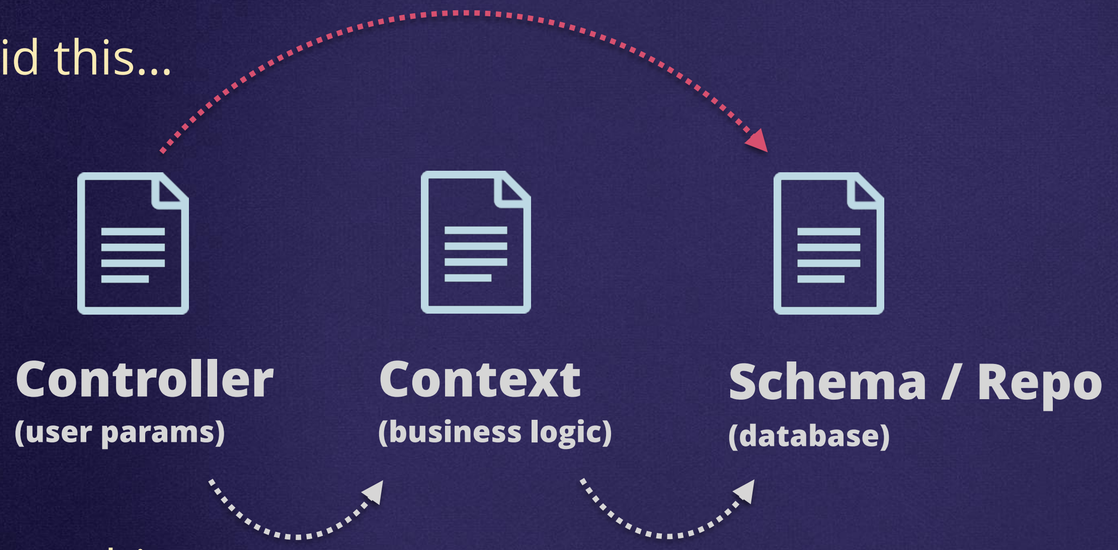
Model

- Schema Modules
- Context Modules

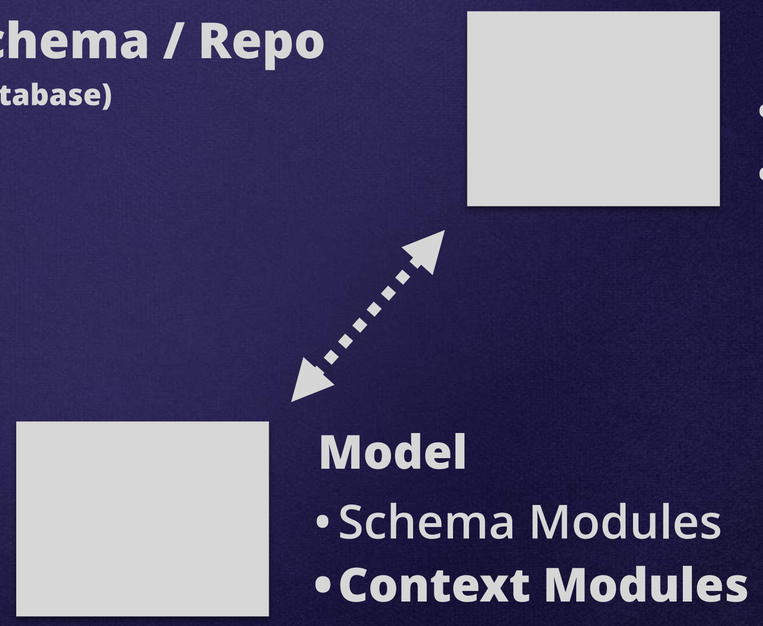
Controllers Should Talk to Contexts

Context modules allows us to **decouple and isolate** our code into manageable and independent parts.

We should avoid this...



...and instead favor this



Controller

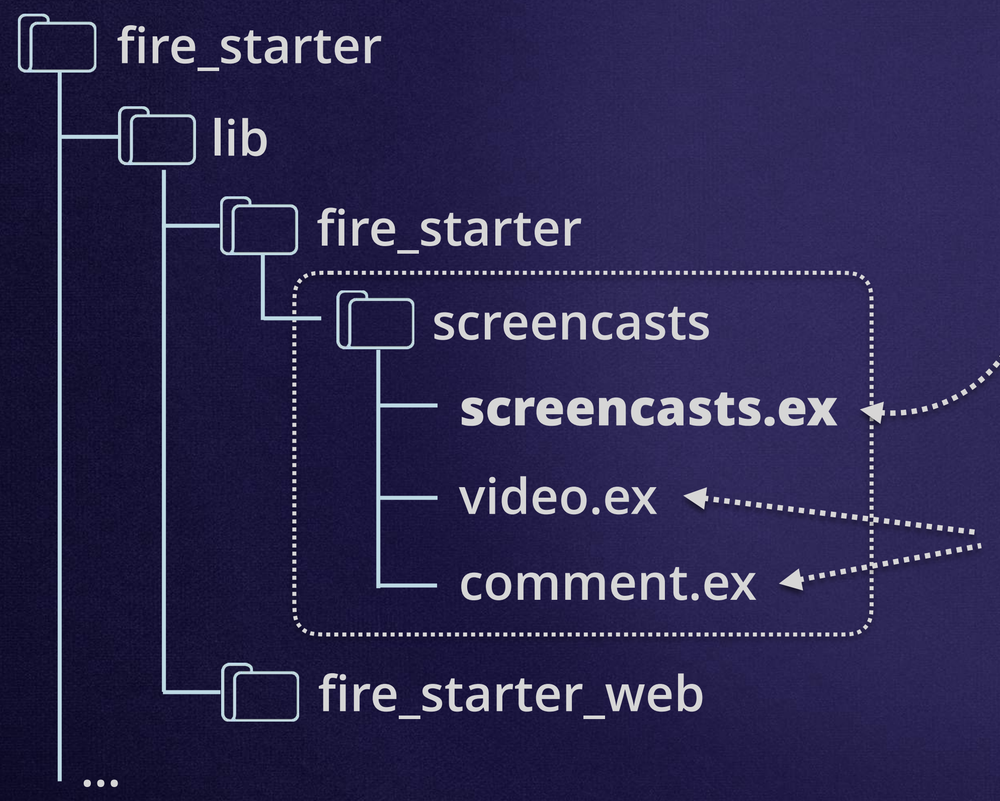
- Router Module
- Controller Modules

Model

- Schema Modules
- **Context Modules**

Moving *Video* **and** *Comment* **Inside** *Screencasts*

The *Screencasts* **module will be the entry point for all video-related operations.**



A *Context* is simply
an **Elixir module**

Now part of the **screencasts** folder
and *Screencasts* namespace

Tight Coupling When Listing Videos

Currently, any changes to reading videos from the database will **directly affect** this code.

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...
  def index(conn, _) do
    videos = Repo.all(Video)
    render conn, "index.html", videos: videos
  end

  def show(conn, %{"id" => id}) do
    video = Repo.get(Video, id) |> Repo.preload(:comments)
    render conn, "show.html", video: video
  end
end
```

Too much knowledge
about other parts of the app...

...and references to *Repo.*

Moving Calls to *Repo* to the *Context*

Reading videos from the database is now **decoupled and isolated** from the *Controller*.

lib/fire_starter/**screencasts/screencasts.ex**

```
defmodule FireStarter.Screencasts do

  def list_videos do
    Repo.all(Video)
  end

  def get_video(id) do
    Repo.get(Video, id) |> Repo.preload(:comments)
  end
end
```

A module part of the *FireStarter* namespace

Move code here from `VideoController`

Moving Aliases from *Controller* to *Context*

The necessary calls to *alias* must also be moved to *Screencasts* module.

lib/fire_starter/**screencasts/screencasts.ex**

```
defmodule FireStarter.Screencasts do
  alias FireStarter.Repo
  alias FireStarter.Screencasts.Video

  def list_videos do
    Repo.all(Video)
  end

  def get_video(id) do
    Repo.get(Video, id) |> Repo.preload(:comments)
  end
end
```

Needed for shorter references to *Repo* and *Video*

Calling a *Context* from the *Controller*

The code for reading videos is now shorter and **decoupled** from *Repo* and *Schema*.

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  use FireStarterWeb, :controller
  alias FireStarter.Screencasts
  def index(conn, _) do
    videos = Screencasts.list_videos()
    render conn, "index.html", videos: videos
  end

  def show(conn, %{"id" => id}) do
    video = Screencasts.get_video(id)
    render conn, "show.html", video: video
  end
end
```

Level 5 - Section 2

Using *Contexts*

Creating Videos

On Fire WITH PHOENIX

Tight Coupling for New Forms

The *Controller* is **tightly coupled** to the *Ecto* library for generating new video forms.

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarterWeb.VideoController do
  ...
  import Ecto.Changeset

  def new(conn, _) do
    changeset = change(%Video{})
    render conn, "new.html", changeset: changeset
  end

end
```

Controller needs to know about *Ecto*...

...in order to create a changeset

Moving changeset code to *Schema*

We'll slightly change the code for a **changeset** and move it inside the *Video* ***Schema.***

lib/fire_starter_web/controllers/video_controller.ex

```
defmodule FireStarter.Screencasts.Video do
  ...

  def changeset(%Video{} = video, attrs) do
    video
    |> cast(attrs, [:title, :duration])
  end
end
```

part of the *Screencasts* namespace

By using *cast* instead of *change,* we can later re-use the `changeset` function when creating a new *Video*

Moving alias **and** import

We need these two lines: one to invoke cast() **and the other one to reference** %Video{}**.**

```
defmodule FireStarter.Screencasts.Video do

  import Ecto.Changeset
  alias FireStarter.Screencasts.Video

  def changeset(%Video{} = video, attrs) do
    video
    |> cast(attrs, [:title, :duration])
  end
end
```

Creating a changeset from the *Context*

The *change_video* function from *Screencasts* is now in charge of **creating a changeset.**

```
defmodule FireStarter.Screencasts do

  ...

  def change_video(%Video{} = video) do
    Video.changeset(video, %{})
  end
end
```

This function will be called from the *VideoController*

Controller calls *Context* for changeset

The *Controller* now calls a function from *Screencasts* in order to create a changeset.

```
defmodule FireStarterWeb.VideoController do
  ...

  def new(conn, _) do
    changeset = Screencasts.change_video(%Video{})
    render conn, "new.html", changeset: changeset
  end

end
```

It's fine to use the *Schema* as argument here

No longer relies on *Ecto* for creating a changeset!

Tight Coupling for Creating New Videos

The *VideoController* is **tightly coupled** with *Ecto* and *Repo* for creating new videos.

```
defmodule FireStarterWeb.VideoController do
  ...
  import Ecto.Changeset
  alias FireStarter.Repo

  def create(conn, %{"video" => video_params}) do
    changeset = cast(%Video{},  video_params, [:title, :url, :duration])
    case Repo.insert(changeset) do
      {:ok, _} -> ...
      {:error, changeset} -> ...
    end
  end
end
```

Controller needs to know about *Ecto* and *Repo*.

Too many details about creating video are exposed.

Moving Creation Code to *Context*

The new Screencasts.create_video **function encapsulates the logic** for creating a new video.

```
defmodule FireStarter.Screencasts do

  ...

  def create_video(attrs \\ %{}) do
    %Video{}
    |> Video.changeset(attrs)
    |> Repo.insert()
  end
end
```

User submitted attributes

Uses the new function we've created in *Video.*

It's ok to call *Repo* from the *Context*

Using *Context* to Create New Videos

The code for creating videos is now shorter and **decoupled** from *Repo* and *Schema*.

```
defmodule FireStarterWeb.VideoController do
  ...

  def create(conn, %{"video" => video_params}) do
    case Screencasts.create_video(video_params) do
      {:ok, _} -> ...
      {:error, changeset} -> ...
    end
  end
end
```