Phoenix Framework Cheatsheet

A comprehensive cheat sheet covering essential Phoenix framework concepts, commands, and code snippets for efficient web development in Elixir.

Project Setup & Core Concepts

Project Generation

Creating a new Phoenix project:

```
mix phx.new my_app
```

This command generates a new Phoenix project named `my_app`. Follow the prompts to install dependencies and create the database.

Creating a new Phoenix project with `ecto` (database interaction):

```
mix phx.new my_app --no-ecto
```

Use the `--no-ecto` to skip ecto.

Starting the Phoenix server:

```
cd my_app
mix phx.server
```

Starts the Phoenix application server. By default, it runs on port 4000.

Key Directories

`lib/my_app_web`	Contains the core web application code, including controllers, views, channels, and templates.
`lib/my_app`	Contains the application's business logic and domain models.
`priv/repo`	Contains database migrations and schema definitions (if using Ecto).
`config`	Contains configuration files for different environments (dev, test, prod).

Core Components

Router	Directs incoming HTTP requests to the appropriate controller action.
Controller	Handles user requests, interacts with models, and renders views.
View	Prepares data for presentation in templates.
Template	Generates the HTML output using data provided by the view (typically `.eex` files).
Channel	Handles real-time communication using WebSockets.

Routing and Controllers

Defining Routes

Routes are defined in `lib/my_app_web/router.ex`. Use the `scope` and `pipe_through` macros to group routes.

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

  get "/", PageController, :index
  resources "/users", UserController
end
```

`get`, `post`, `put`, `delete`, `patch` - HTTP methods mapped to controller actions.

`resources` - Generates routes for common CRUD operations.

Controller Actions

A controller action receives `conn` (connection) and `params` (request parameters).

```
def index(conn, _params) do
  render(conn, :index)
end
```

`render(conn, template, assigns)` - Renders a template with the given assigns.

`redirect(conn, opts)` - Redirects the client to a different URL.

`put_status(conn, status_code)` - Sets the HTTP status code.

Working with Parameters

Accessing parameters	Parameters are available in the `params` argument of a controller action.
	`def create(conn, %{"user" => user_params}) do`
	`  # Access user_params`
	`end`

Strong parameters	Use `permit` in your controller to filter parameters.
	`def create(conn, params) do`
	`  user_params = Map.take(params["user"], ["name", "email"])`
	`  # Access user_params`
	`end`

Ecto Integration

Defining Schemas

Schemas define the structure of your database tables. They reside in `lib/my_app/`. Example:

```
defmodule MyApp.User do
  use Ecto.Schema
  import Ecto.Changeset

  schema "users" do
    field :name, :string
    field :email, :string

    timestamps()
  end

  @required_fields ~w(name email)a

  def changeset(user, attrs \\ %{}) do
    user
    |> cast(attrs, [:name, :email])
    |>
validate_required(@required_fields)
    |> validate_format(:email, ~r/@/) #
Regex email validation
    |> unique_constraint(:email)
  end
end
```

- `use Ecto.Schema` - Imports Ecto schema functionality.
- `schema "table_name" do ... end` - Defines the database table associated with the schema.
- `field :field_name, :data_type` - Defines a field in the schema.
- `timestamps()` - Adds `inserted_at` and `updated_at` fields.

Changesets

Changesets are used to validate and cast data before saving it to the database.

```
import Ecto.Changeset

def changeset(user, attrs) do
  user
  |> cast(attrs, [:name, :email])
  |> validate_required([:name, :email])
  |> validate_format(:email, ~r/@/) #
Regex email validation
  |> unique_constraint(:email)
end
```

- `cast(data, params, permitted)` - Casts and filters parameters.
- `validate_required(changeset, fields)` - Validates that required fields are present.
- `validate_format(changeset, field, regex)` - Validates the format of a field using a regular expression.
- `unique_constraint(changeset, field)` - Validates the uniqueness of a field.

Repo Operations

Inserting data	`Repo.insert(changeset)`
Updating data	`Repo.update(changeset)`
Deleting data	`Repo.delete(struct)`

Retrieving data	`Repo.get(MyApp.User, id)`
	`Repo.all(MyApp.User)`
	`Repo.one(query)`

Templates and Views

Template Syntax

Phoenix templates use the `.eex` (Embedded Elixir) extension. Use `<%= ... %>` to output the result of an Elixir expression.

```
<h1>Hello, <%= @name %>!</h1>
```

Use `<% ... %>` to execute Elixir code without outputting anything.

```
<% if @show_greeting do %>
  <p>Welcome!</p>
<% end %>
```

- `<%= raw(html) %>` - Outputs raw HTML without escaping.
- `<%= render "_partial.html", assigns %>` - Renders a partial template.

View Modules

Views prepare data for templates. They are defined in `lib/my_app_web/views/`.

```
defmodule MyAppWeb.PageView do
  use MyAppWeb, :view

  def format_date(date) do
    Date.to_string(date)
  end
end
```

- `use MyAppWeb, :view` - Imports common view functions and macros.

Define helper functions to format data or perform other logic for your templates.

Layouts

Default Layout	The main layout is in `lib/my_app_web/templates/layout/app.html.eex`.
Rendering Content	Use `<%= @inner_content %>` to render the content of the template within the layout.

Custom Layouts	You can specify a different layout in your controller using `put_layout`.
	`conn`
	`|>`
	`put_layout("alternative")`
	`|> render(:index)`