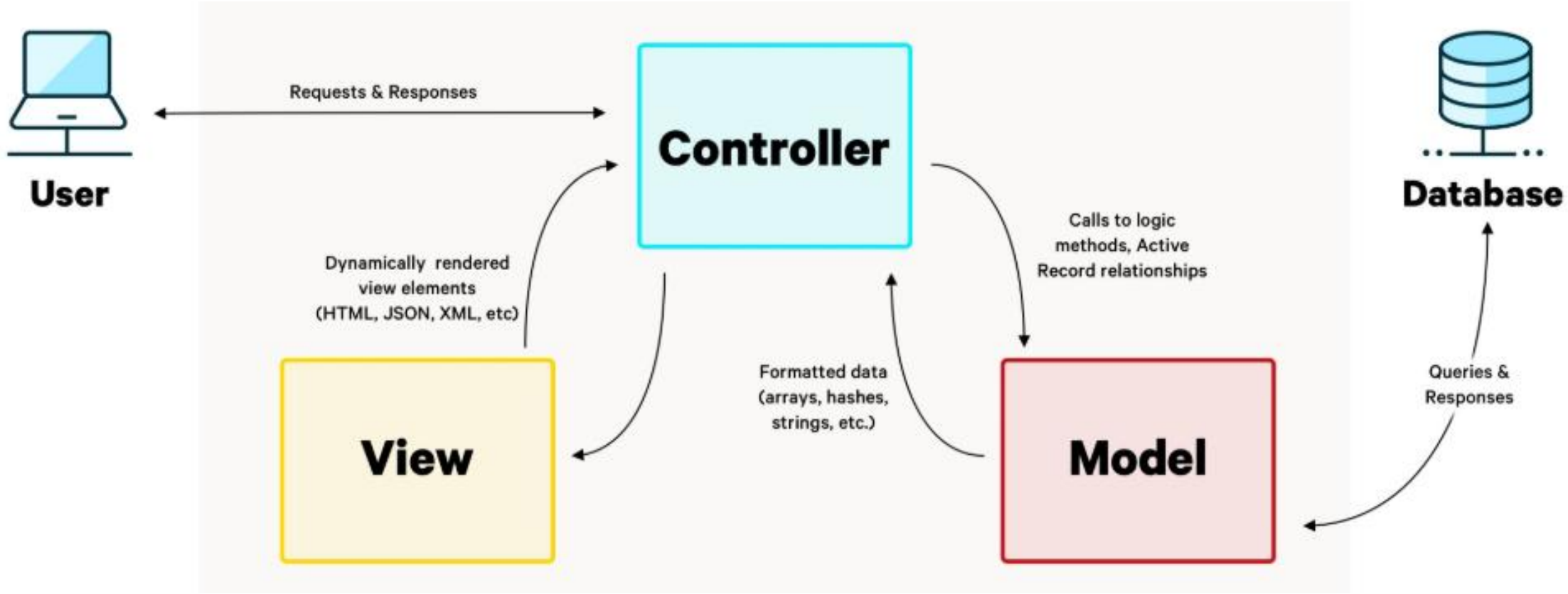
Rails

Temario de la clase de hoy

- Cómo instalar Rails, crear una nueva aplicación Rails y conectar su aplicación a una base de datos.
- El diseño general de una aplicación Rails.
- Los principios básicos de MVC (Modelo, Vista, Controlador).
- Cómo generar rápidamente un CRUD.

Fundamentos del modelo-vista-controlador

Con MVC, tenemos tres conceptos principales donde reside la mayor parte de nuestro código:



Instalación Linux

Si tienen Windows es similar, pero usando WLS (Ubuntu)

```
# Install dependencies with apt
$ sudo apt update
$ sudo apt install build-essential rustc libssl-dev libyaml-dev zlib1g-dev
libgmp-dev git

# Install Mise version manager
$ curl https://mise.run | sh
$ echo 'eval "$(~/.local/bin/mise activate)"' >> ~/.bashrc
$ source ~/.bashrc

# Install Ruby globally with Mise
$ mise use -g ruby@3
```

```
$ gem install rails
```

Primera aplicación

El comando **rails new** genera la base de una nueva aplicación Rails
Por ejemplo, para crear una aplicación llamada store, ejecutamos el siguiente comando:

```
$ rails new store
```

Después de crear la aplicación, cambiamos de directorio:

```
$ cd store
```

Ejecutamos para iniciar el servidor Rails

```
$ bin/rails server
```

Primera aplicación

Rails incluye varios comandos para simplificar la tarea. Ejecútalo rails –help para verlos todos.
rails new genera la base de una nueva aplicación Rails para usted, así que comencemos por ahí.
Para crear nuestra storeaplicación, ejecute el siguiente comando en su terminal:

```
$ rails new store
```

Después de crear su nueva aplicación, cambie a su directorio:

```
$ cd store
```

Comencemos de manera sencilla e iniciemos nuestro servidor Rails por primera vez.
En su terminal, ejecute el siguiente comando en el `store`directorio:

```
$ bin/rails server
```

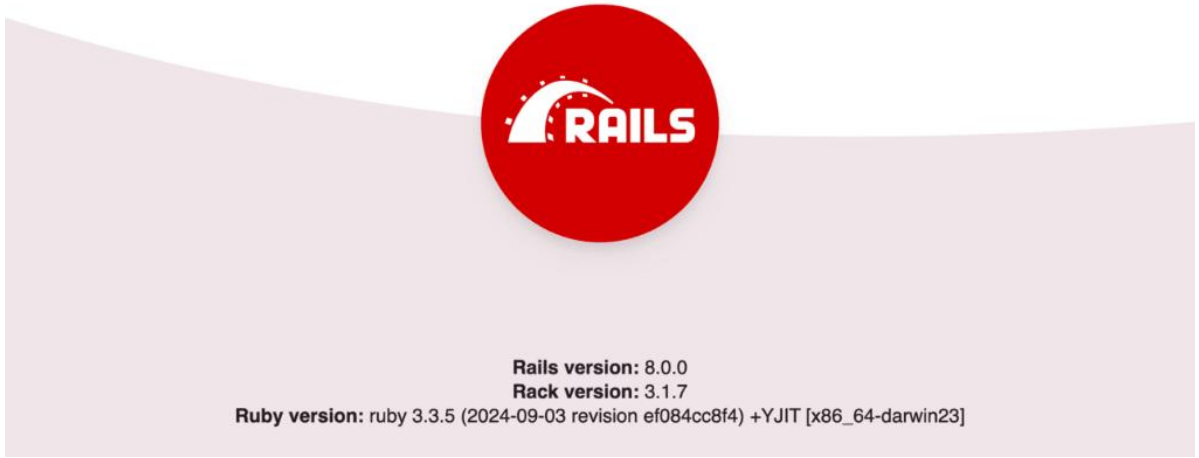
Primera aplicación

Esto iniciará un servidor web llamado Puma

```
=> Booting Puma
=> Rails 8.0.0 application starting in development
=> Run `bin/rails server --help` for more startup options
Puma starting in single mode...
* Puma version: 6.4.3 (ruby 3.3.5-p100) ("The Eagle of Durango")
*  Min threads: 3
*  Max threads: 3
*  Environment: development
*          PID: 12345
* Listening on http://127.0.0.1:3000
* Listening on http://[::1]:3000
Use Ctrl-C to stop
```

Primera aplicación

Para ver la aplicación Rails, abrimos **http://localhost:3000** en nuestro navegador y nos tiene que figurar la página de bienvenida predeterminada de Rails:



Para detener el servidor Rails en cualquier momento, presionamos `Ctrl-C` en la terminal.

Creación de un modelo de base de datos

Rails utiliza Active Record para mapear los objetos a bases de datos relacionales.
Nuestra aplicación utiliza SQLite, la versión predeterminada de Rails.
Comencemos agregando una tabla de base de datos a nuestra aplicación Rails para agregar productos a nuestra sencilla tienda de comercio electrónico.

```
$ bin/rails generate model Product name:string
```

Este comando le indica a **Rails** que genere un modelo llamado **Product** que tenga una columna llamada **name** de tipo de **string** en la base de datos.

Este comando hace varias cosas. Crea...
1.Una migración en la **db/migratecarpeta**.
2.Un modelo de registro activo en **app/models/product.rb**.
3.Test para este modelo.

Migraciones de bases de datos

En tu editor de código, abre la migración que Rails creó para nosotros para que podamos ver qué hace. Esta se encuentra en **db/migrate/<timestamp>_create_products.rb**:

```ruby
class CreateProducts < ActiveRecord::Migration[8.0]
  def change
    create_table :products do |t|
      t.string :name

      t.timestamps
    end
  end
end
```

El siguiente comando ejecuta las migraciones

```
$ bin/rails db:migrate
```

Consola Rails

Ahora que hemos creado nuestra tabla de productos, podemos interactuar con ella en Rails. Probémosla.
Para ello, usaremos una función de Rails llamada *consola* . Esta es una herramienta interactiva y útil para probar nuestro código en nuestra aplicación Rails.

```
$ bin/rails console
```

Va a mostrar un mensaje como el siguiente:

```
Loading development environment (Rails 8.0.0)
store(dev)>
```

Aquí podemos escribir el código que se ejecutará al presionar Enter. Intentemos imprimir la versión de Rails:

```
store(dev)> Rails.version
=> "8.0.0"
```

Fundamentos del modelo de registro activo

```
class Product < ApplicationRecord
end
```

Podemos consultar desde la consola que atributos creo para este modelo

```
store(dev)> Product.column_names
```

```
=> ["id", "name", "created_at", "updated_at"]
```

Validaciones

Active Record proporciona *validaciones* que permiten garantizar que los datos insertados en la base de datos cumplan con ciertas reglas.
Por ejemplo, si queremos que el atributo **name** sea obligatorio

```
class Product < ApplicationRecord
  validates :name, presence: true
end
```

```
store(dev)> reload!
Reloading...
```

Intentemos crear un producto sin nombre en la consola Rails.

```
store(dev)> product = Product.new
store(dev)> product.save
=> false
```

Rutas CRUD

Para definir una ruta en Rails para nuestros productos agreguemos la siguiente ruta a **config/routes.rb**

```
resources :products
```

Automáticamente va a crear todas estas rutas

```
                                 Prefix Verb   URI Pattern
Controller#Action
                               products GET    /products(.:format)
products#index
                                        POST   /products(.:format)
products#create
                            new_product GET    /products/new(.:format)
products#new
                           edit_product GET    /products/:id/edit(.:format)
products#edit
                                product GET    /products/:id(.:format)
products#show
                                        PATCH  /products/:id(.:format)
products#update
                                        PUT    /products/:id(.:format)
products#update
                                        DELETE /products/:id(.:format)
products#destroy
```

Controladores

Luego de definir rutas para los productos, implementemos el controlador y las acciones para manejar las solicitudes a estas URL.
El siguiente comando generará el controlador de productos, como ya creamos las rutas en el paso anterior pedimos que omita esa parte.

```
$ bin/rails generate controller Products index --skip-routes
```

Este comando genera los siguientes archivos:

1. El propio controlador
2. Una carpeta de vistas para el controlador que generamos
3. Un archivo de vista para la acción que especificamos al generar el controlador
4. Un archivo de prueba para este controlador
5. Un archivo auxiliar para extraer lógica en nuestras vistas

Controladores

El controlador del producto

```
class ProductsController < ApplicationController
  def index
  end
end
```

La vista del producto

```
<h1>Products#index</h1>
<p>Find me in app/views/products/index.html.erb</p>
```

El punto de inicio

Si abrimos http://localhost:3000/products en el navegador, Rails renderizará el índice HTML de los productos.

Si agregamos en **config/routes.rb**, podemos indicarle a Rails que la ruta raíz debe representar la acción de índice de Productos agregando esta línea:

```
root "products#index"
```

Ahora, cuando visite http://localhost:3000 , Rails renderizará Products#index.

Variables de instancia

Llevemos esto un paso más allá y representemos algunos registros de nuestra base de datos.
En la index acción, agreguemos una consulta de base de datos y la asignemos a una variable de instancia. Rails usa variables de instancia (variables que empiezan por @) para compartir datos con las vistas.

```
class ProductsController < ApplicationController
  def index
    @products = Product.all
  end
end
```

En **app/views/products/index.html.erb**, podemos reemplazar el HTML con este ERB:

Listado de productos

Actualicémoslo app/views/products/index.html.erb para representar todos los nombres de nuestros productos.

```
<h1>Products</h1>

<div id="products">
  <% @products.each do |product| %>
    <div>
      <%= product.name %>
    </div>
  <% end %>
</div>
```

Mostrando productos individuales

Controller

```ruby
class ProductsController < ApplicationController
  def index
    @products = Product.all
  end

  def show
    @product = Product.find(params[:id])
  end
end
```

app/views/products/show.html.erb

```erb
<h1><%= @product.name %></h1>

<%= link_to "Back", products_path %>
```

Mostrando productos individuales

Sería útil que la página de índice enlazara con la página de presentación de cada producto para que podamos hacer clic en ellos y navegar.

```
<h1>Products</h1>

<div id="products">
  <% @products.each do |product| %>
    <div>
      <a href="/products/<%= product.id %>">
        <%= product.name %>
      </a>
    </div>
  <% end %>
</div>
```

Creación de Productos

Agregamos al listado de productos un link para agregar un nuevo producto

```
<h1>Products</h1>

<%= link_to "New product", new_product_path %>

<div id="products">
  <% @products.each do |product| %>
    <div>
      <%= link_to product.name, product %>
    </div>
  <% end %>
</div>
```

Creación de Productos

Vamos a crear **app/views/products/new.html.erb** para renderizar el formulario para este nuevo **Product**.

```
<h1>New product</h1>

<%= form_with model: @product do |form| %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>

  <div>
    <%= form.submit %>
  </div>
<% end %>
```

En esta vista, usamos el form_with asistente de Rails para generar un formulario HTML y crear productos. Este asistente utiliza un *generador de formularios* para gestionar elementos como tokens CSRF, generar la URL según la información proporcionada en **:model**

Creación de Productos

Creamos dos métodos en el controlador new (para renderizar el formulario) y créate (para procesar la acción de carga)
Tambien creamos un método producto_params para validar los parámetros de entrada

```ruby
class ProductsController < ApplicationController
  def index
    @products = Product.all
  end

  def show
    @product = Product.find(params[:id])
  end

  def new
    @product = Product.new
  end

  def create
    @product = Product.new(product_params)
    if @product.save
      redirect_to @product
    else
      render :new, status: :unprocessable_entity
    end
  end

  private
    def product_params
      params.expect(product: [ :name ])
    end
end
```

Edición de Productos

El proceso de edición de registros es muy similar al de creación de registros. En lugar de las acciones **new** y **create**, tendremos **edit** y **update**.

```ruby
def edit
  @product = Product.find(params[:id])
end

def update
  @product = Product.find(params[:id])
  if @product.update(product_params)
    redirect_to @product
  else
    render :edit, status: :unprocessable_entity
  end
end
```

Edición de Productos

A continuación podemos agregar un enlace Editar a **app/views/products/show.html.erb**

```
<h1><%= @product.name %></h1>

<%= link_to "Back", products_path %>
<%= link_to "Edit", edit_product_path(@product) %>
```

En el controller vamos a utilizar **before_action** para optimizarlo indicando que antes de las acciones determinada invoque el método set_product

```
class ProductsController < ApplicationController
  before_action :set_product, only: %i[ show edit update ]
```

```
def set_product
  @product = Product.find(params[:id])
end
```

Edición de Productos

Vamos a crear un partial que represente al formulario y que lo podamos reutilizar en las acciones de crear y editar.
También queremos reemplazar cualquier variable de instancia con una variable local, que podemos definir al renderizar el parcial. Para ello, reemplazaremos @product con product.

```
<%= form_with model: product do |form| %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>

  <div>
    <%= form.submit %>
  </div>
<% end %>
```

Edición de Productos

El uso de variables locales permite reutilizar los parciales varias veces en la misma página con un valor diferente cada vez. Esto resulta útil para representar listas de elementos como una página de índice.
Para usar este parcial en nuestra **app/views/products/new.html.erb** vista, podemos reemplazar el formulario con una llamada de renderizado:

```
<h1>New product</h1>

<%= render "form", product: @product %>
<%= link_to "Cancel", products_path %>
```

La vista de edición se vuelve prácticamente idéntica gracias al formulario parcial. Crearemos en **app/views/products/edit.html.erb** lo siguiente:

```
<h1>Edit product</h1>

<%= render "form", product: @product %>
<%= link_to "Cancel", @product %>
```

Eliminación de productos

La última función que vamos a agregar al controlador es

```
def destroy
  @product.destroy
  redirect_to products_path
end
```

Para que esto funcione, necesitamos agregar un botón Eliminar a app/views/products/show.html.erb:

Eliminación de productos

Para que esto funcione, necesitamos agregar un botón Eliminar a **app/views/products/show.html.erb**:

```
<h1><%= @product.name %></h1>

<%= link_to "Back", products_path %>
<%= link_to "Edit", edit_product_path(@product) %>
<%= button_to "Delete", @product, method: :delete, data: { turbo_confirm: "Are you sure?" } %>
```

button_to genera un formulario con un solo botón con el texto "Eliminar". Al hacer clic en este botón, se envía el formulario, que realiza una solicitud DELETE **/products/:id** que activa la acción destroy en nuestro controlador.
Utilizamos Turbo de JavaScript para los mensajes.