



Objetivo

Explorar paradigmas especializados con sus lenguajes nativos: lógico, concurrente, dirigido por eventos y reactivo.

Paradigma Lógico

Basado en hechos y reglas. El motor infiere conclusiones automáticamente.

Prolog (lenguaje lógico puro)

```
% Hechos: relaciones familiares
padre(juan, ana).
padre(juan, luis).
padre(pedro, juan).
madre(maria, ana).
madre(maria, luis).

% Reglas: definiciones lógicas
hermano(X, Y) :-
    padre(Z, X),
    padre(Z, Y),
    X \= Y.

abuelo(X, Z) :-
    padre(X, Y),
    padre(Y, Z).

% Consultas:
% ?- hermano(ana, luis).      % true
```

```
% ?- abuelo(pedro, ana).      % true
% ?- abuelo(X, ana).          % X = pedro
```

Paradigma Concurrente

Múltiples tareas ejecutándose simultáneamente.

Go (concurrency con goroutines)

```
package main

import (
    "fmt"
    "sync"
    "time"
)

// Procesamiento concurrente con canales
func procesarURL(url string, ch chan<- string) {
    // Simular procesamiento
    time.Sleep(time.Millisecond * 100)
    ch <- fmt.Sprintf("Procesado: %s", url)
}

func main() {
    urls := []string{"url1", "url2", "url3", "url4"}
    ch := make(chan string, len(urls))

    // Lanzar goroutines
    for _, url := range urls {
        go procesarURL(url, ch)
    }

    // Recoger resultados
    for i := 0; i < len(urls); i++ {
        resultado := <-ch
        fmt.Println(resultado)
    }
}
```

Erlang/Elixir (Actor Model)

```
defmodule ProcesadorConcurrente do
    # Procesar lista de URLs concurrentemente
    def procesar_urls(urls) do
        urls
        |> Enum.map(&Task.async(&fn -> procesar_url(&1) end))
        |> Enum.map(&Task.await/1)
    end
```

```

defp procesar_url(url) do
  # Simular trabajo
  :timer.sleep(100)
  "Procesado: #{url}"
end

# Uso
ProcesadorConcurrente.procesar_urls(["url1", "url2", "url3"])

```

Rust (ownership + concurrencia segura)

```

use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let urls = vec!["url1", "url2", "url3", "url4"];
    let (tx, rx) = mpsc::channel();

    // Crear hilos para cada URL
    for url in urls {
        let tx_clone = tx.clone();
        thread::spawn(move || {
            thread::sleep(Duration::from_millis(100));
            tx_clone.send(format!("Procesado: {}", url)).unwrap();
        });
    }

    // Recoger resultados
    drop(tx); // Cerrar el canal principal
    for resultado in rx {
        println!("{}", resultado);
    }
}

```

Casos de uso: Sistemas distribuidos, microservicios, procesamiento paralelo.

Paradigma Dirigido por Eventos ⚙️

El flujo se controla mediante eventos y reacciones.

Node.js EventEmitter (nativo)

```

const EventEmitter = require('events');

class SistemaVentas extends EventEmitter {
  constructor() {
    super();
    this.configurarEventos();
  }
}

```

```
}

configurarEventos() {
  this.on('usuarioRegistrado', this.enviarBienvenida);
  this.on('pedidoCreado', this.procesarPago);
  this.on('pagoCompletado', this.enviarConfirmacion);
}

registrarUsuario(usuario) {
  // Lógica de registro...
  this.emit('usuarioRegistrado', usuario);
}

crearPedido(pedido) {
  // Lógica de pedido...
  this.emit('pedidoCreado', pedido);
}

enviarBienvenida(usuario) {
  console.log(`✉ Bienvenida enviada a ${usuario.email}`);
}

procesarPago(pedido) {
  console.log(`💳 Procesando pago de €${pedido.total}`);
  setTimeout(() => {
    this.emit('pagoCompletado', {
      pedidoId: pedido.id,
      exito: true
    });
  }, 1000);
}

enviarConfirmacion(pago) {
  console.log(`☑ Confirmación enviada para ${pago.pedidoId}`);
}

// Uso
const sistema = new SistemaVentas();
sistema.registrarUsuario({ email: 'user@email.com' });
sistema.crearPedido({ id: 'P123', total: 99.99 });
```