

Tabela de funcionalidades

Aspecto	Elixir/Phoenix	Django	Golang	Ruby on Rails
Desempenho	Excelente	Médio	Excelente	Médio
Escalabilidade	Excelente	Médio	Excelente	Médio
Produtividade inicial	Boa	Muito boa	ruim	boa
Curva de aprendizado	média/alta	baixa	média	baixa
Sistemas de tempo real	Excelente	Limitado	Bom	Limitado
Ecossistema	Médio	Excelente	Médio	Bom
Comunidade	Média/baixa	Grande	Média	Grande

Tabela de Testes

Linguagem	Vantagens nos testes	Desvantagens nos testes
Elixir com Phoenix	<p>Ferramentas de teste integradas no núcleo da linguagem (ExUnit).</p> <ul style="list-style-type: none"> - Facilidade em escrever testes paralelos devido à concorrência nativa do Elixir. - Suporte direto para <i>mocks</i> e <i>stubs</i> com bibliotecas adicionais como Mox. - Testes funcionais e unitários podem ser combinados com testes assíncronos. - Feedback rápido devido à compilação eficiente 	<p>Curva de aprendizado maior para iniciantes que vêm de linguagens orientadas a objetos.</p> <ul style="list-style-type: none"> - Ferramentas de testes mais avançadas (e.g., <i>property-based testing</i>) exigem bibliotecas externas, como StreamData.
Django	<p>Extenso suporte a bibliotecas de testes (e.g., pytest, unittest).</p> <ul style="list-style-type: none"> - Integração fácil com ferramentas externas (e.g., Selenium para testes de front-end). - Documentação rica e uma comunidade ativa para casos de teste comuns. 	<p>Os testes geralmente são mais lentos devido à natureza interpretada da linguagem.</p> <ul style="list-style-type: none"> - Menor suporte nativo para paralelismo, impactando a execução de muitos testes simultâneos.
Ruby on Rails	<p>Convenções fortes, como geração automática de testes com scaffolding.</p> <ul style="list-style-type: none"> - Integração com ferramentas como RSpec, que possuem uma sintaxe muito legível (estilo BDD). - Ambiente rico para testes de integração com Capybara e FactoryBot. 	<p>Testes mais lentos devido à natureza single-threaded do Ruby.</p> <ul style="list-style-type: none"> - Configuração de testes paralelos é mais complexa.
Golang	<p>Excelente para testes de desempenho devido à velocidade da linguagem.</p> <ul style="list-style-type: none"> - Geração de cobertura de código nativa (<code>go test -cover</code>). 	<p>Sintaxe de testes menos expressiva (comparada a linguagens de alto nível como Python e Ruby).</p>

		- Ferramentas para testes de integração são limitadas.
--	--	--------------------------------------------------------

Elixir com Phoenix

- Filosofia: Concurrency-first (otimizado para desempenho em tempo real), escalabilidade e tolerância a falhas, herdado do Erlang.

Pontos Fortes

1. Escalabilidade:
 - a. Baseado no Erlang/OTP, Phoenix é excepcional em concurrency (milhões de conexões simultâneas).
 - b. Ideal para sistemas em tempo real, como chats, jogos multiplayer e streams.
2. Desempenho:
 - a. Altamente eficiente em aplicativos de I/O intensivo e conexões longas.
 - b. Menos consumo de recursos comparado a Django e Rails.
3. Tempo Real:
 - a. Com Phoenix Channels, suporta WebSockets nativamente, tornando simples a criação de aplicativos em tempo real.
4. Tolerância a Falhas:
 - a. Supervisores e a arquitetura OTP ajudam a lidar com falhas automaticamente.
5. Produtividade:
 - a. Boa combinação de desempenho e facilidade de desenvolvimento.
 - b. O código é funcional e explícito, mas também expressivo.
6. Comunidade Crescente:
 - a. Embora menor que Django e Rails, a comunidade está crescendo e é muito ativa.

Pontos Fracos

1. Curva de Aprendizado:
 - a. A programação funcional em Elixir é diferente do paradigma OO (usado em Django e Rails).
 - b. Pode ser confuso para iniciantes.
2. Menor Ecossistema:
 - a. Faltam bibliotecas em comparação ao Python e Ruby.
 - b. Pode exigir mais trabalho para funcionalidades específicas.
3. Menor Adoção:

- a. Embora em crescimento, Elixir ainda é menos comum em projetos comerciais e tem menos vagas no mercado.

4. Menor Foco em Backend Tradicional:

- a. Apesar de poderoso, frameworks como Django e Rails são mais completos para CRUDs tradicionais e sistemas administrativos.

PRINCIPAIS COMANDOS ELIXIR COM PHOENIX:

1. Inicialização de Projetos e banco de dados

- `mix phx.new nome_do_projeto -->` cria projeto
 - `--no-html`: Remove arquivos relacionados a HTML e views (ideal para APIs)
 - `--no-assets`: Remove configuração para assets estáticos (CSS, JS, etc)
 - `--no-ecto`: Não inclui o Ecto (gerenciador de banco de dados)
 - `--database mysql`: Configura o projeto para usar MySQL em vez de PostgreSQL
 - `--umbrella`: Cria um projeto com estrutura umbrella (para apps modulares)
- `mix phx.server -->` iniciar servidor
- `mix compile -->` compila projeto (nunca usei)
- `mix ecto.create -->` cria BD
- `mix ecto.drop -->` apaga BD
- `mix ecto.migrate -->` realizar migrações
- `mix ecto.rollback -->` reverter migrações (nunca usei)
- `mix ecto.gen.migration nome_da_migracao -->` criar nova migração
- `mix ecto.migrations -->` listar migrações
- `mix ecto.reset -->` resetar BD

2. Geração de Contextos e demais arquivos

- `mix phx.gen.context Contexto NomeSchema nome_tabela campo:tipo campo2:tipo -->` context + schema

- `mix phx.gen.json Contexto NomeSchema nome_tabela campo:tipo campo2:tipo --> API completa`
- `mix phx.gen.schema NomeSchema nome_tabela campo:tipo campo2:tipo --> gera apenas schema`

3. Dependências

- `mix deps.get --> npm install`
- `mix deps.compile --> compilar dependencias`
- `mix deps.clean --all --> limpa e recompila`

4. Realização de testes pré prontos

- `mix test --> realiza teste`
 - `--cover --> gera um relatório`
 - `--log` habilita logs de `IO.puts` e `Logger`
- `mix test caminho/para/arquivo_test.exs --> realiza teste de um arquivo específico`
- `--env test --> manipular banco de dados teste (ecto.create, migrate, etc)`

5. Formatação e organização do código

- `mix format --> formata e organiza o código`