


Modular Monoliths with Rails Engines

by Anton / @antulik

What is a Rails Engine?

- <https://guides.rubyonrails.org/engines.html>

More at rubyonrails.org: [Blog](#) | [Guides](#) | [API](#) | [Forum](#) | [Contribute on GitHub](#)

RAILSGUIDES

[Home](#) [Guides Index](#) [Contribute](#)

Getting Started with Engines

In this guide you will learn about engines and how they can be used to provide additional functionality to their host applications through a clean and very easy-to-use interface.

After reading this guide, you will know:

- ✔ What makes an engine.
- ✔ How to generate an engine.
- ✔ How to build features for the engine.
- ✔ How to hook the engine into an application.
- ✔ How to override engine functionality in the application.
- ✔ How to avoid loading Rails frameworks with Load and Configuration Hooks.

1 What are Engines?

Engines can be considered miniature applications that provide functionality to their host applications. A Rails application is actually just a "supercharged" engine, with the `Rails::Application` class inheriting a lot of its behavior from `Rails::Engine`.

Therefore, engines and applications can be thought of as almost the same thing, just with subtle

Chapters

1. **What are Engines?**
2. **Generating an Engine**
 - [Inside an Engine](#)
3. **Providing Engine Functionality**
 - [Generating an Article Resource](#)
 - [Generating a Comments Resource](#)
4. **Hooking Into an Application**
 - [Mounting the Engine](#)
 - [Engine Setup](#)
 - [Using a Class Provided by the Application](#)
 - [Configuring an Engine](#)
5. **Testing an Engine**
 - [Functional Tests](#)
6. **Improving Engine Functionality**
 - [Overriding Models and Controllers](#)
 - [Autoloading and Engines](#)
 - [Overriding Views](#)
 - [Routes](#)

The architecture spectrum



Default
Rails
architecture

Lean
Engines

Lean
Engines
+ Packwerk

Rails
engines
(local or
published)

Microservices

Getting started with engines in 1 minute

```
# apps/zom_app/lib/engine.rb
module ZomApp
  class Engine < Rails::Engine
  end
end
```

```
# config/application.rb
# Load engines
Dir["apps/*/lib/engine.rb"].each do |path|
  require_relative "../" + path
end
```

Reference:

All you need is Rails (Engines): Compartmentalising your Monolith

by Julián Pinzón Eslava

- <https://www.youtube.com/watch?v=StDoHXO8H6E>
- https://github.com/pinzonjulian/all_you_need_is_rails_engines

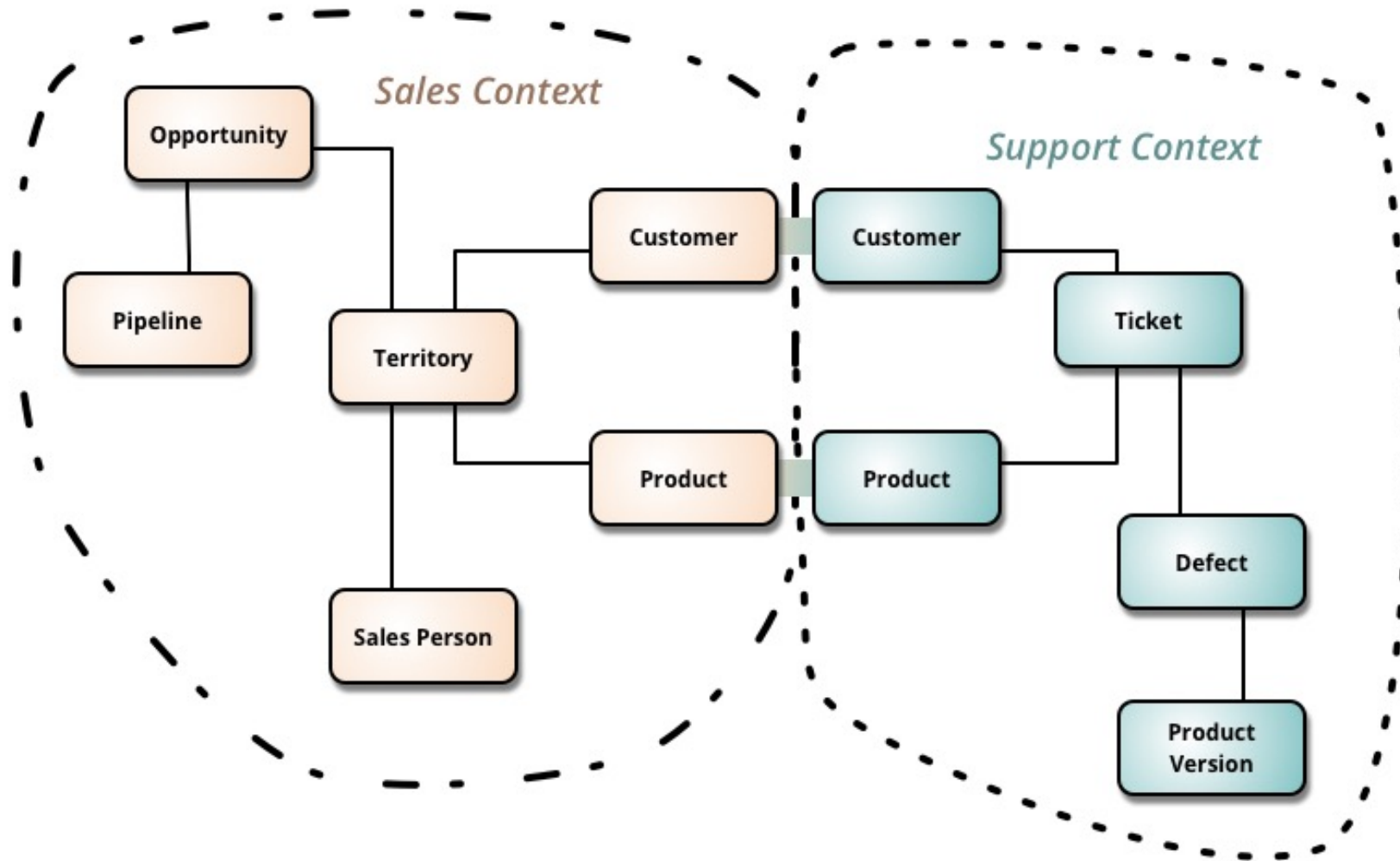
My new Rails structure

- aka Modular Monolith, aka Lean Engines
- empty `/app`
- `/apps` folder
- `core` engine
- single `routes.rb` file
- single db migration list
- specs are separate per engine

Why separate?

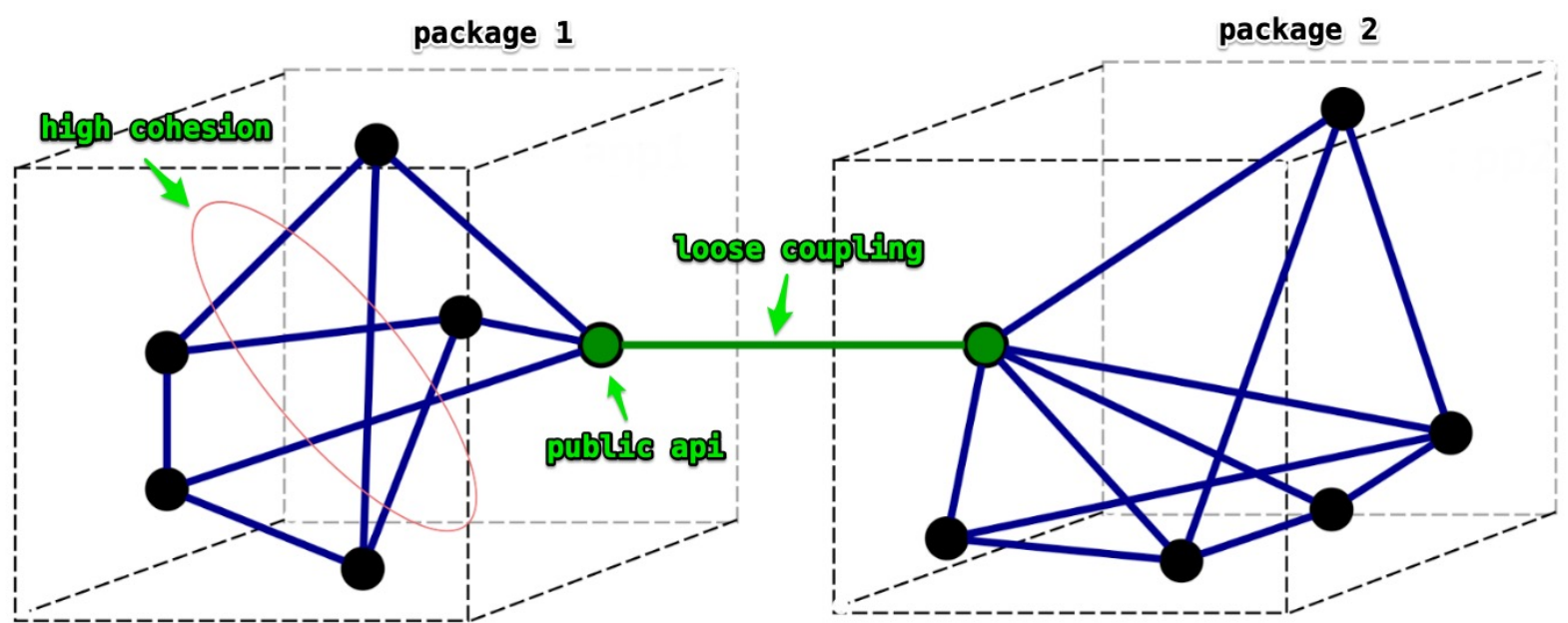
Domain Driven Design and Bounded Context

- <https://martinfowler.com/bliki/BoundedContext.html>

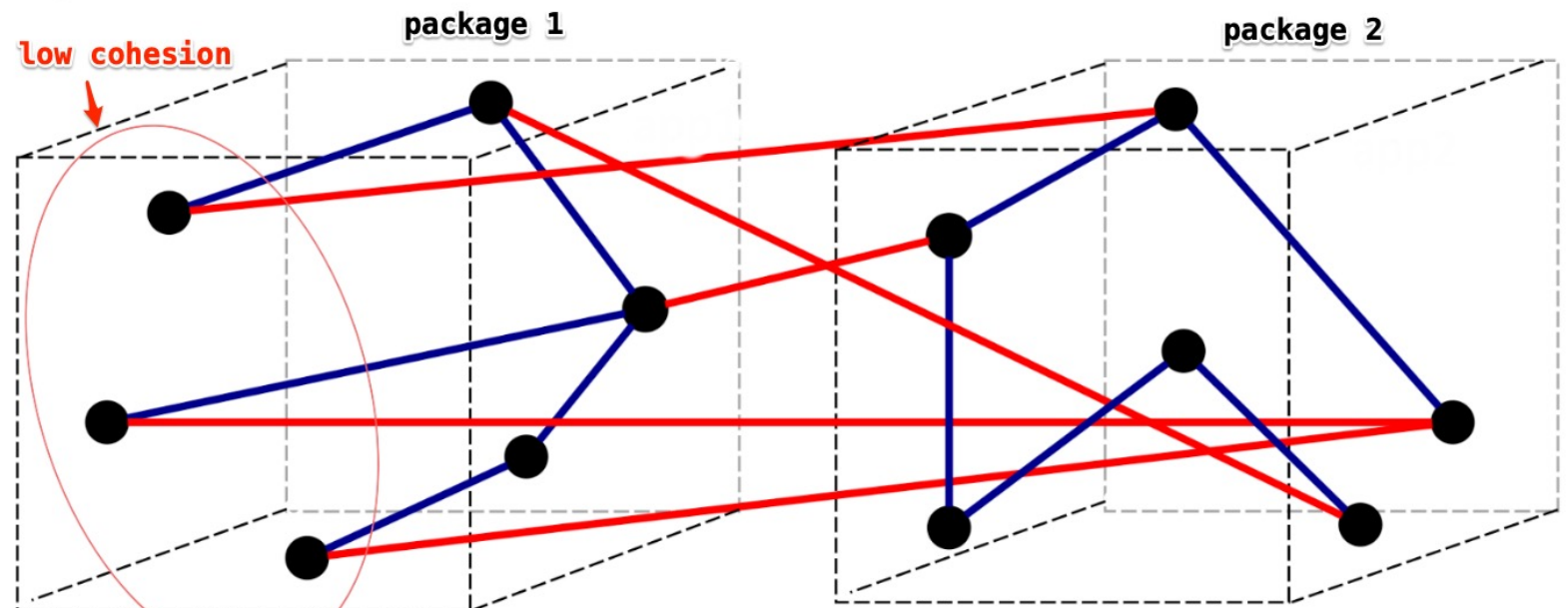


Why separate?

- High cohesion
- Loose coupling



a) Good



b) Bad

Why separate?

- lower cognitive load
- clear responsibility for dev teams
- factories are simple and efficient
- testing is easier
 - clear what you should be stubbing in tests

Why separate?

My experience

Engine folder structure forced me to think about code dependencies upfront and come up with a better system design.

My Development Principle: Optimise for Change

There is always a trade-off

Optimise for Change 1

Engines don't use single namespace (controversial)

```
# this:  
class ZomUser  
  
# not this:  
class Zom::User
```

- aka Shopify approach
- easy transition from Rails Monolith
- follows Rails naming convention
 - `has_one :zom_user` just works
- clear code context, no more multiple `User` models or `user` variables.

Optimise for Change 2

Use name prefixes

- `ZomUser`, `zom_user` and `TriUser`, `tri_user`
- consistency across relationships, local variables, param names

Optimise for Change 3

No module nesting (aka flat class definitions)

```
# this:
module A::B
  p Module.nesting # [A::B]
end

# not this:
module A
  module C
    p Module.nesting # [A::C, A]
  end
end
```

Lessons

Lesson 1: No ruby code in `app` folder

- Engine separation leads to better system and code design

Lesson 2: You will get it wrong. Design for change.

Lesson 3: Lean Engines are very easy to get started and transition to.

Lesson 4: Mountable engines are harder to change if you got boundary wrong.

Avoid mountable engines for Rails app (unless you know what you're doing)

Mountable engine feature improves code isolation and reduces code conflicts across engines. However it's harder to work with if system design is still evolving.

```
module Blorgh
  class Engine < ::Rails::Engine
    isolate_namespace Blorgh # <== avoid mountable engine
  end
end
```

Note: mountable engines are better suited for gems.

Lesson 5: All Rails apps larger than "tiny" will benefit from Lean Engines.

Lesson 6: Not everything works out of the box and some config is required.

- e.g. assets, gems, specs, previews

Lesson 7: Better architecture is not free, and there is additional but small effort required.

Lesson 8: OMG coding is fun again

Yet to answer: cross boundary logic

- Needs a clear convention for where to place logic which integrates two engines.
- E.g. who owns `UserPostsController` ? `ProfileApp` or `PostApp` engine?

Future discussion: packwerk

The architecture spectrum



Default
Rails
architecture

Lean
Engines

Lean
Engines
+ Packwerk

Rails
engines
(local or
published)

Microservices

Reference

- Slides done with <https://marpit.marp.app/>
- Cohesion image is from <https://github.com/Shopify/packwerk/blob/main/USAGE.md>
- Architecture spectrum image is from <https://www.youtube.com/watch?v=StDoHXO8H6E>

Thank you

Discussion. What's your experience?

e.g.

- Your experience with Rails Engines
- Nested modules vs flat structure
- Dealing with multiple `User` models
- Domain Driver Design and bounded context
- Dealing with application and code complexity
- Your experience with packwerk