# Ruby on Rails Modular Monolith Structure

Engines-based modular monolith. Domain separation without microservices complexity.

#rails #ruby #modular #engines #monolith #ddd

PNG PDF Copy Prompt

## Project Directory

```
myplatform/
  Gemfile  References engi…
  Gemfile.lock
  Rakefile
  config.ru
  app/  Shell app, thin…
    controllers/
      application_controller.rb
    views/
      layouts/
        application.html.erb
  engines/  Domain modules …
    core/  Shared code, ba…
      core.gemspec
      lib/
      app/
        models/
        services/  Base services
    users/  User management…
      users.gemspec
      lib/
        users.rb
        users/
      app/
        controllers/
          users/
        models/
          user.rb
        views/
        services/
      config/
        routes.rb
      db/
        migrate/
      spec/
    billing/  Payments and su…
      billing.gemspec
      lib/
        billing.rb
        billing/
      app/
        controllers/
        models/
          subscription.rb
          invoice.rb
        services/
        jobs/
      config/
        routes.rb
      db/
      spec/
    notifications/  Email, push, in…
      notifications.gemspec
      lib/
      app/
        mailers/
        jobs/
        services/
      spec/
  config/
    application.rb
    routes.rb  Mounts engine r…
    database.yml
    credentials.yml.enc
    environments/
    initializers/
  db/  Main app migrat…
    migrate/
    seeds.rb
    schema.rb
  spec/  Integration tes…
    rails_helper.rb
    integration/
    support/
  lib/
  log/
  public/
  tmp/
  bin/
```

## Why This Structure?

A modular monolith uses Rails engines to create domain boundaries without the operational complexity of microservices. Each engine is a gem with its own models, controllers, and tests. Teams can work independently while sharing a single deployment.

## Key Directories

**`engines/`** - Each subdirectory is a Rails engine (gem)

**`engines/core/`** - Shared base classes and utilities

**`engines/*/app/`** - Full MVC stack per domain

**`engines/*/db/migrate/`** - Domain-specific migrations

**`config/routes.rb`** - Mounts engine routes at paths

## Engine Configuration

```
# Gemfile
gem 'core', path: 'engines/core'
gem 'users', path: 'engines/users'
gem 'billing', path: 'engines/billing'

# config/routes.rb
mount Users::Engine, at: '/users'
mount Billing::Engine, at: '/billing'
```

## Getting Started

1. `rails new myplatform`
2. `rails plugin new engines/users --mountable`
3. Add engine to Gemfile as path gem
4. Mount engine routes in `config/routes.rb`
5. `rails users:install:migrations && rails db:migrate`

## When To Use This

- Multiple teams working on one codebase
- Clear domain boundaries (users, billing, orders)
- Want microservices benefits without complexity
- Preparing for potential future service extraction
- Large apps needing code isolation

## Trade-offs

**`Setup overhead`** - Creating engines requires boilerplate

**`Cross-engine refs`** - Need careful dependency management

**`Single deploy`** - Still one app to deploy and scale

**`Shared database`** - No true data isolation between domains

## Testing Strategy

**`Engine tests`** - Each engine has its own `spec/` folder

**`Integration tests`** - Main app tests cross-engine flows

**`Dependency testing`** - Mock other engines at boundaries

**`CI strategy`** - Run all engine tests, then integration