Rails Engine

A Rails application is actually just a "supercharged" Rails Engine, with the `Rails::Application` class inheriting a lot of its behaviour from `Rails::Engine`.

An engine can include all the same features as a regular Rails application, including generators, migrations, tests, and configuration options. It can also have its own dependencies, such as gems or other engines.

Rails Engines can be considered miniature applications that provide functionality to their **host applications**.

Therefore, engines and applications can be thought of as almost the same thing.

Why Rails Engine and what for?

Modularising your monolith using Rails Engines.

A Rails engine is a self-contained piece of functionality that can be added to an existing Rails application, or used as the foundation for a new Rails application.

A Rails Engine is a miniature Rails application that can be packaged and reused across multiple projects.

Rails engines provide a modular way to organise and reuse code in a Rails application. They allow you to encapsulate related functionality (such as authentication, payments, or notifications) in a separate namespace, with its own models, views, controllers, routes, and assets.

% 我認為，Rails Engine 可以用來做基本面或大功能模塊，例如：Authentication、Authorisation、Payment、Admin、Email、Help desk。這些大概被所有app通用的基本面，都應該做成 Rails Engine。每個 Rails Engine 就是一個 code repo。所有Engine 共用一個main數據庫。20240112 %

Considerations

- Be mindful of interdependencies between engines, as this can complicate development and deployment.
- Consistent authentication and authorization across engines are crucial, especially for user experience and security.
- Shared services (like user sessions, notifications, etc.) need careful planning to work seamlessly across different engines.

Use a Rails Engine for each big feature and subdomain

Using Rails engines for different functional modules of your application is an excellent approach, especially if you are considering organizing these modules under different subdomains of your main site (e.g., `payment.app.com`, `employee.app.com`, `inventory.app.com`, `stores.app.com`). This approach aligns well with domain-driven design principles, where different domains (like payment, inventory, etc.) are clearly demarcated and managed.

Here's how and why this can be beneficial:

1. **Modularity:**

   Rails engines allow you to build modular components within your Rails application. Each engine can be a self-contained app with its own models, views, controllers, and routes. This modularity makes it easier to manage and develop different aspects of your application independently.

2. **Subdomain Integration:**

   You can configure each engine to serve a specific subdomain. This is done through **routing**, where each subdomain is directed to the corresponding engine. This helps in logically separating different parts of your application, both from a user perspective and from a development standpoint.

3. **Development and Maintenance:**

   Separate teams can work on different engines without stepping on each other's toes. This is particularly useful for large applications or teams. It simplifies maintenance since updates to one module (engine) can often be made independently of others.

4. **Deployment:**
   While each engine can be a part of the larger monolithic application, you have the option to deploy them separately if performance demands it. This can be a step towards a more microservices-oriented architecture.

5. **Performance Considerations:**

   This can also lead to improved performance, as each subdomain/engine can be optimized for its specific functionality.

6. **Reuse and Extensibility:**

   Engines can be reused across different applications, making them a great way to build and maintain common features. They provide a level of extensibility that's hard to achieve in a tightly coupled monolithic application.

How to use a Rails Engine?

It's important to keep in mind at all times that the application should always take precedence over its engines. An application is the object that has final say in what goes on in its environment. The engine should only be enhancing it, rather than changing it drastically.

To call an engine method in a Rails application, you can simply require the engine and call its methods as you would with any other Ruby class.

Steps

- Create each function as a separate Rails engine.
- Configure your application's routing to direct requests to the appropriate engine based on the subdomain.
- Ensure each engine is self-contained but also integrates smoothly with the main application and other engines.

How to use routing to direct each subdomain

In Ruby on Rails, directing different subdomains to specific engines involves using the application's routing system to map requests based on the subdomain to the appropriate engine. Here's how it generally works:

Setting Up Subdomain Routing

Rails routes can be constrained by subdomains using the `constraints` method. This allows you to specify that certain routes should only be accessible from specific subdomains.

For example, suppose you have engines for 'payments', 'employees', 'inventory', and 'stores', each serving `payments.app.com`, `employees.app.com`, `inventory.app.com`, and `stores.app.com`, respectively. You can set up your routes to direct requests to these subdomains to the corresponding engines:

```
# In config/routes.rb of your main Rails application

Rails.application.routes.draw do

     constraints subdomain: 'payments' do
          mount PaymentEngine::Engine, at: '/'
     end

     constraints subdomain: 'employees' do
         mount EmployeeEngine::Engine, at: '/'
     end
```

```
    constraints subdomain: 'inventory' do
        mount InventoryEngine::Engine, at: '/'
    end

    constraints subdomain: 'stores' do
        mount StoreEngine::Engine, at: '/'
    end

    # Other routes for your main application
end
```

Key Points:

- **Mounting Engines:** Each engine is 'mounted' onto a specific route within the main application. In this case, we're mounting each engine at the root path (`at: '/'`) but constraining them to different subdomains.
- **Subdomain Constraints:** The `constraints` method is used to specify the subdomain that each block of routes will respond to. Any request coming to a specific subdomain is routed to the corresponding engine.
- **Namespacing and Isolation:** If engines are namespaced and isolated, they can have their own separate routing, models, controllers, etc., which helps in keeping the functionality and data of different engines separate and organized.

Additional Configuration

- **DNS Configuration:** Ensure that your DNS settings are configured to direct these subdomains to your Rails application's server.
- **Environment Configuration:** Remember to account for subdomain routing in different environments (development, test, production), as local and production environments might handle subdomains differently.
- **Testing:** It's important to include tests for your routing configuration to ensure that subdomain constraints are working as expected.

By using subdomain-based routing to direct traffic to different engines, you can effectively modularize your application, making it more manageable and scalable. This setup allows different teams to work on different engines without interfering with each other, and also provides a clear separation of concerns within the application.

How can multiple engines share a main database?

Sharing a single large relational database among multiple Rails engines is a common scenario, especially in monolithic applications where different components (engines) need to interact with the same dataset. Here's how you can set this up and manage it effectively:

1. **Common Database Configuration:** All engines can use the same database configuration specified in the main application. This means they all connect to the same database server and use the same credentials. The database.yml file in the main Rails application typically contains this configuration, and it's shared across all engines.
2. **Migrations and Schema Management:** Each engine can have its own set of database migrations. These migrations reside within the engine but are run in the context of the main application to modify the shared database schema. Ensure that migrations in different engines do not conflict with each other, particularly in naming and table alterations.
3. **Model Sharing and Overriding:** Engines can define their own models, which interact with the shared database. If necessary, engines can also extend or override models defined in other engines or the main application. It's important to manage these interactions carefully to avoid conflicts or unexpected behavior.
4. **Namespacing:** To avoid collisions between similar model names or database tables across different engines, consider namespacing your models within each engine. For example, a `User` model in a `Payment` engine could be namespaced as `Payment::User`.
5. **Data Integrity and Referential Integrity:** Since multiple engines are accessing the same database, maintaining data integrity is crucial. This includes proper use of foreign keys, constraints, and transactional operations. Consistent validation logic across engines is important to maintain data quality.
6. **Database Access Patterns:** Be mindful of how different engines access the database. Overlapping or conflicting access patterns (like concurrent writes to the same tables) need careful management. Optimizing database queries and indexes is crucial, especially as the application scales.
7. **Shared Libraries or Concerns:** For common functionality related to database interaction (like common scopes, query methods, etc.), you can create shared libraries or concerns that can be used by models across different engines.
8. **Database Connections Pooling:** Rails manages a pool of database connections. Ensure that the pool size is configured to handle the load from all engines, especially under high traffic.
9. **APIs for Inter-Engine Communication:** If direct database access is not desirable or feasible (for instance, for decoupling purposes), engines can expose APIs that other engines can use to request data or trigger actions.
10. **Testing:** When testing engines, you must ensure that the tests are aware of the shared database structure. This might involve setting up a test database that mirrors the structure of the production database.

By carefully managing these aspects, you can ensure that multiple Rails engines work harmoniously together, leveraging the same database efficiently and effectively. This approach allows for modular development while maintaining a unified data layer.

How to convert a Rails app into an engine

Converting an existing Rails application (like "Storefront") into an engine so that it can be integrated into another larger Rails application (like "Backbone") involves several steps. Here's a general guide on what you need to do:

1. Convert to Engine

- **Create Engine File:** In your Storefront application, create a new file in `lib/storefront/engine.rb`. This file defines the engine. Example:

```
module Storefront
     class Engine < ::Rails::Engine
     end
end
```

- **Modify the Gemspec File:** Convert the Storefront's `storefront.gemspec` file. This involves setting up dependencies correctly and ensuring it is treated like a gem. Then, update dependencies, if necessary, to reflect those used in your main app.
- **Namespacing:** Namespace the controllers, models, and other modules to avoid conflicts with Backbone. For example, `Storefront::ProductsController` instead of just `ProductsController`.

2. Adjust Routes

- **Namespace Routes:** Modify `config/routes.rb` in Storefront to nest routes within a module. This helps to isolate Storefront's routes from Backbone's routes. Example:

```
Storefront::Engine.routes.draw do
       # your routes here
end
```

3. Update Database Migrations

- **Isolate Migrations:** If Storefront has migrations, you'll want to isolate them so that they can be run within the context of the Backbone application. In `engine.rb`, add `isolate_namespace Storefront` and a path to load migrations.

4. Update Assets and Views

- **Asset Pipeline:** Ensure that Storefront's assets (JavaScripts, CSS, images) are properly namespaced and included in the asset pipeline. You might need to adjust asset paths and helper methods in views and layouts.

5. Modify Initializers and Configurations

- **Isolate Initializers:** If Storefront has initializers, review and update them as necessary. Make sure they don't conflict with Backbone's settings.

6. Update Tests

- **Test Suite:** Update your test suite to reflect changes. Ensure all tests pass with the new engine structure.

7. Integrate into Backbone

- **Add Engine to Backbone:** Add Storefront as a gem dependency in Backbone's `Gemfile`. If you are not publishing to a gem server, you can point to a local path or Git repository. Example: `gem 'storefront', path: '../storefront'`
- **Mount Engine:** In Backbone's `config/routes.rb`, mount the Storefront engine. Example:

```
Rails.application.routes.draw do
       mount Storefront::Engine, at: '/storefront'
       # other routes...
end
```

8. Configuration and Environmental Adjustments

- **Environment Files:** Ensure environment-specific configurations in Storefront are compatible or properly overridden in Backbone.
- **Database Configuration:** If using different databases, ensure the models refer to the correct database. If sharing the same database, be careful with table names and relationships.

Final Steps

- **Manual Testing:** Thoroughly test the integrated application to ensure that all components work together seamlessly.
- **Deployment:** Update the deployment process to include the new engine.

By following these steps, you can effectively convert your "Storefront" application into a Rails engine and integrate it into your company's main "Backbone" application. This process involves careful refactoring, namespacing, and testing to ensure seamless integration and functionality within the larger application ecosystem.

A Monolith of multiple engines vs Microservices

Comparing multiple Rails engines within a monolithic application versus a microservices architecture is essential for understanding their advantages, disadvantages, and best use cases.

Multiple Engines in a Monolith

Advantages:

- **Shared Resources:** Engines in a monolith share the same resources, like the database, memory, and runtime environment.
- **Simplified Deployment:** The entire application, including all its engines, can be deployed as a single unit, simplifying deployment and hosting.
- **Ease of Communication:** Inter-engine communication is straightforward since they operate within the same application environment.
- **Shared Data Model:** Engines can easily share models and data, which can be advantageous for tightly coupled applications.

Disadvantages:

- **Coupled Deployment:** Updating one part of the application often requires redeploying the entire application.
- **Resource Intensity:** The entire application, with all its engines, can be resource-intensive as it grows.
- **Monolithic Complexity:** Even with modularization, the application can still become complex and difficult to manage as the number of engines grows.

Best Used For:

- Applications where different modules are tightly integrated and share a lot of functionality and data.
- Teams transitioning from a monolithic architecture to a more modular one without fully adopting microservices.

Microservices Architecture

Microservices architecture involves developing a suite of small, independent services, each running in its own process and communicating with lightweight mechanisms, often HTTP APIs.

Advantages:

- **Scalability:** Each microservice can be scaled independently based on demand.
- **Technological Flexibility:** Different microservices can use different technologies (programming languages, databases, etc.).
- **Resilience:** Failure in one microservice doesn't necessarily bring down the entire application.
- **Independent Development and Deployment:** Microservices can be developed, deployed, and scaled independently.

Disadvantages:

- **Complex Communication:** Inter-service communication over the network is more complex than within a monolithic application.
- **Data Duplication:** Microservices may need to duplicate data, which can lead to consistency challenges.
- **Operational Complexity:** Managing multiple services requires robust infrastructure, monitoring, and logging.
- **Development Overhead:** Building and maintaining a microservices-based application can be more complex and require more coordination.

Best Used For:

- Large, complex applications where different components have distinct functionalities and little overlap.
- Teams that require a lot of flexibility in terms of technology and scalability.

Summary

- **Multiple Engines in a Monolith** are best for applications where modularity is desired within the confines of a single, unified application. This approach provides modularization benefits without the complexity of managing a distributed system.
- **Microservices Architecture** is ideal for large-scale, complex applications where independent services, scalability, and flexibility are paramount. This approach, however, requires a more sophisticated approach to system design and operations.

Choosing between these two approaches depends on the specific needs of your application, the scale at which you operate, the expertise of your team, and the resources available to manage the infrastructure.

Why did Github chose monolith over microservices?

GitHub's decision to maintain a monolithic architecture over adopting microservices is a significant choice, especially in an era where many large-scale applications are moving towards microservices. There are several reasons why a company like GitHub might choose to stick with a monolithic architecture:

1. Simplicity and Cohesion

- **Easier Codebase Management:** In a monolith, all the code resides in a single repository, making it easier to manage, test, and deploy.
- **Cohesive Development Workflow:** Developers can work on a single application, which simplifies setup, debugging, and cross-functional changes.

2. Performance and Reliability

- **Internal Communication Overhead:** Monoliths can avoid the network latency and complexity associated with the inter-service communication in microservices.
- **Reliability:** With a well-maintained monolithic codebase, the potential for service disruptions due to interdependency issues or network problems can be reduced.

3. Evolution of the Application

- **Historical Growth:** GitHub started as a smaller application and grew over time. Evolving from a monolith to microservices can be challenging and may not be necessary if the monolith is well-structured.
- **Incremental Refactoring:** GitHub can gradually refactor and modularize their monolith internally without needing to fully transition to microservices.

4. Organizational Structure

- **Team Dynamics:** GitHub's team structure might be well-suited to a monolithic architecture, where cross-functional teams can collaborate more easily on a single codebase.
- **Controlled Scaling:** GitHub might have found that a monolithic architecture suffices for their scale of operations and user base, without the need for the complexity of microservices.

5. Technical Debt and Resource Allocation

- **Avoiding the Overhead of Microservices:** Adopting microservices isn't free of cost; it requires significant investment in terms of infrastructure, tooling, and expertise.
- **Managing Technical Debt:** It might be more feasible for GitHub to manage and pay down technical debt within a monolithic structure than to undertake the substantial effort of transitioning to a microservices architecture.

6. Continuous Delivery and Integration

- **Simplified Deployment:** Continuous integration and deployment can be more straightforward with a monolith, as there's only one application to build, test, and deploy.
- **Feature Development and Rollback:** New features and rollbacks can be managed more seamlessly when they are part of a single codebase.

Conclusion

The choice between a monolithic architecture and microservices architecture is highly dependent on the specific needs, history, and context of the application and organization. For GitHub, the benefits of maintaining a cohesive, unified codebase, along with the operational simplicity and reliability of a monolith, likely outweigh the advantages that a transition to microservices would bring. This decision underscores that while microservices offer certain benefits, they are not a one-size-fits-all solution, and a well-structured monolith can be equally effective for certain applications and organizations.

In a monolith of multiple engines, what does the main app do?

In a monolithic application composed of multiple Rails engines, the main app serves as the central coordinating unit that ties everything together. It has several key responsibilities and roles:

1. Application Framework and Infrastructure:

- **Core Framework:** The main app provides the fundamental Rails framework upon which the engines operate. This includes the Rails runtime environment, configuration settings, and initializations.
- **Shared Resources:** It manages shared resources like database connections, cache stores, job queues, and other infrastructure components that are used by the engines.

2. Integration Point for Engines:

- **Mounting Engines:** The main app is responsible for mounting each engine. This involves specifying where in the application's URL structure each engine's routes will be accessible.
- **Orchestrating Engines:** The main app often handles the high-level orchestration of how different engines interact with each other, including shared authentication, authorization, and cross-cutting concerns like logging and monitoring.

3. Common Functionalities and Shared Code:

- **Shared Models and Libraries:** If there are common models or libraries that are used by multiple engines, the main app is typically where these are placed.
- **Utility Services:** It can provide utility services and shared business logic that are essential across engines.

4. Entry Point for Requests:

- **Routing and Middleware:** The main app is usually the entry point for all web requests. It handles top-level routing and middleware, directing requests to the appropriate engine.
- **Error Handling and Security:** Global error handling and security measures are often managed at the main app level.

5. Overarching Business Logic:

- **High-Level Business Processes:** For business logic that spans across multiple domains represented by different engines, the main app can act as the coordinator or integrator of these processes.
- **Workflow Management:** It may handle workflows that require coordination of services provided by multiple engines.

6. User Interface and Layout:

- **Shared Layouts and Assets:** If there's a common look and feel across different parts of the application, the main app typically manages these shared layouts, stylesheets, and JavaScript assets.
- **Navigation and Global Features:** Features like global navigation, footers, and other cross-cutting UI components are managed by the main app.

7. Data Integrity and Consistency:

- **Database Migrations:** The main app usually handles database migrations, especially for shared database models.
- **Data Consistency:** It ensures data consistency and integrity across the entire application, including the data used or affected by various engines.

Responsibility Allocation:

- **Core vs. Modular Functionality:** Core functionality and infrastructure should reside in the main app, while modular, domain-specific functionality should be encapsulated in individual engines.
- **Avoid Duplication:** Ensure that there's no duplication of functionality between the main app and the engines. Each piece should have a clear, distinct responsibility.

Conclusion

The main app in a multiple-engine monolithic application serves as the foundation and orchestrator, ensuring that the application runs cohesively and efficiently. It provides shared services and infrastructure, integrates and coordinates the engines, and maintains the overarching consistency and integrity of the application. In doing so, it allows the engines to focus on their specific domain responsibilities, promoting modularity and maintainability.

Plan the Orchestration, or Grow into a Monolith from a single app?

The decision between planning a structured orchestration from the beginning versus allowing it to evolve from an initial single Rails application depends on various factors, including the scope of your project, the anticipated scale, and the resources available. Both approaches have their merits and potential drawbacks:

Planning the Orchestration from the Beginning

Advantages:

- **Clear Architecture:** You have a clear understanding of the application's architecture from the start, which can guide development and prevent architectural drift.
- **Scalability and Maintainability:** A well-planned architecture can be more scalable and maintainable, as each component's role and interactions are defined early.

- **Efficient Development:** Knowing the structure upfront can lead to more efficient development, as developers have a clear roadmap and understanding of where to place new features.

Disadvantages:

- **Initial Complexity:** It might introduce complexity early in the project, which could slow down initial development and increase the learning curve for new team members.
- **Potential Overengineering:** There's a risk of overengineering or building a more complex system than necessary, especially if the future scale and requirements of the application are not fully known.

Letting It Grow from a Single Rails App

Advantages:

- **Simplicity and Speed:** Starting with a simple application allows for quicker initial development and easier early-stage adjustments.
- **Evolution Based on Needs:** The application can evolve naturally based on actual needs and requirements, which can be more practical and less theoretical.
- **Reduced Initial Overhead:** Less upfront planning reduces initial overhead and allows focusing on core functionalities first.

Disadvantages:

- **Potential Refactoring:** As the application grows, significant refactoring might be needed to modularize and scale effectively, which can be costly and time-consuming.
- **Architectural Challenges:** Without an initial plan, the application might face architectural challenges later, such as tightly coupled components, that are harder to untangle.

Recommendations

- **Assess Project Scope and Future Scale:** If you anticipate that the application will need to scale significantly or encompass a wide range of functionalities, it might be wise to plan the orchestration from the beginning.
- **Start with a Modular Mindset:** Even if you start with a single app, design it with modularity in mind. This can make future transitions to a more orchestrated or engine-based structure smoother.
- **Iterative Approach:** Consider an iterative approach where you start simple but constantly evaluate and refactor the architecture as the application grows and requirements become clearer.
- **Balance Flexibility and Planning:** Aim for a balance between being flexible to changes and having a structured plan that guides the evolution of the application.

Ultimately, the best approach depends on your specific circumstances, including your team's expertise, the expected lifespan and complexity of the application, and how well you can predict future requirements. In many cases, a hybrid approach that starts with a simpler design but with a mindful eye towards future scalability and modularity can be effective.

~~Rails Plugin~~

Engines are also closely related to plugins. The two share a common lib directory structure, and are both generated using the rails plugin new generator.

Rails plugins were an old way to add reusable functionality to Rails applications. The use of plugins has been deprecated in favour of gems and engines since Rails 3.2. The `vendor/plugins` feature was officially removed in Rails 4.0.