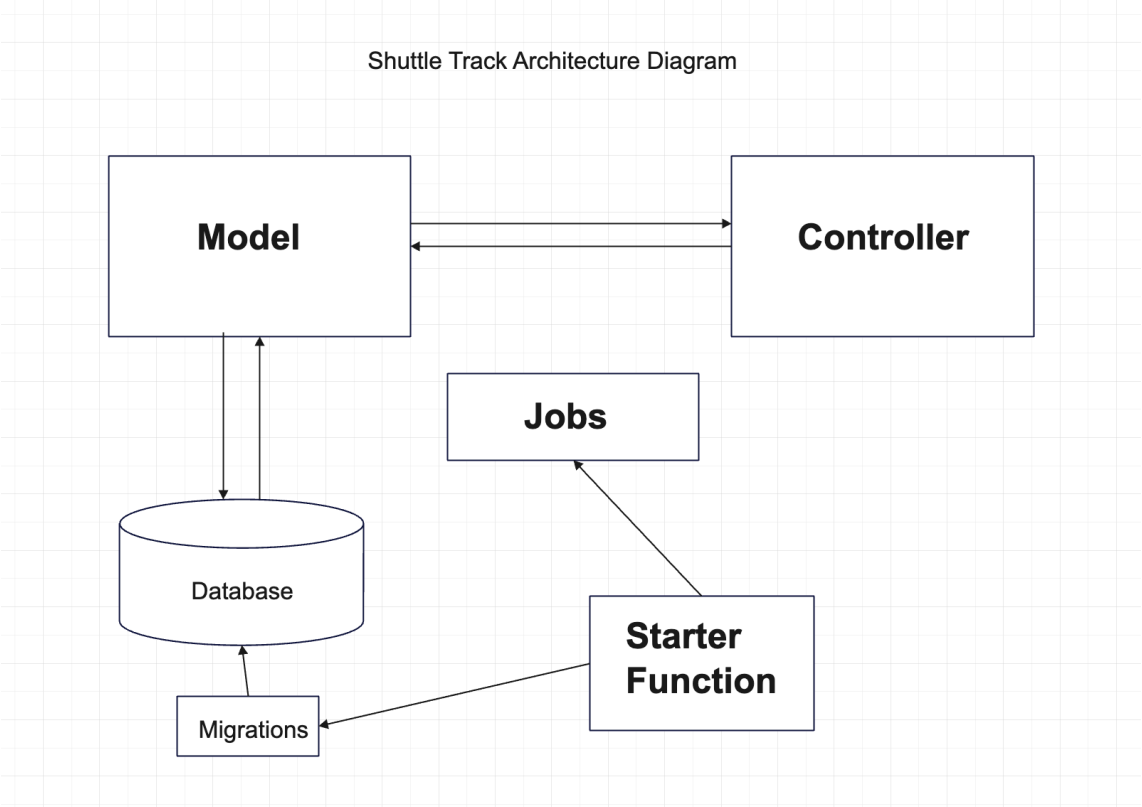
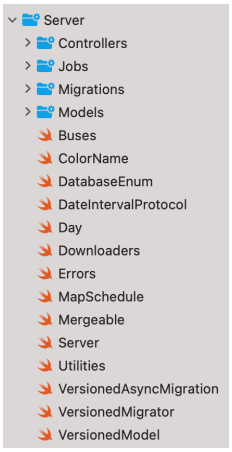
1. High-Level Overview

1. Architecture: Provide a visual representation of the system's architecture, including databases, services, and their interactions.
    a. Architecture Pattern:
        i. Model: The Model corresponds to the structure of the data stored in the database. It defines the type of data stored within the database. The model corresponds with the Controller and Database Component.
        ii. Controller: Controllers act as an interface with the Model component to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.
        iii. Other
        iv. Migrations:Migrations are a way of versioning the database schema and managing its evolution over time. Migrations allow developers to define changes to the database schema in code form, making it easier to apply changes across different environments consistently (development, staging, production) and to collaborate with other developers by keeping the database schema changes under version control.
        v. Jobs: Jobs, refer to operations or tasks that are executed outside the main flow of a web application, usually asynchronously. These can include sending emails, processing files, batch data processing, scheduled tasks, and more.



2. Technology stack:
    a. Swift
        i. Vapor is a web framework for Swift that allows developers to build web applications and APIs. It provides tools and libraries for routing, middleware, authentication, templating, and more.
        ii. Vapor is built on top of SwiftNIO, Apple's asynchronous event-driven networking framework, which enables high-performance, non-blocking I/O operations.
        iii. It follows the MVC (Model-View-Controller) pattern for organizing code and separating concerns.
        iv. Vapor supports asynchronous programming paradigms, making it efficient for handling large numbers of concurrent connections.
        v. It is open-source and has an active community contributing to its development and improvement.
    b. Fluent:
        i. Fluent is an ORM (Object-Relational Mapping) framework for Swift, designed specifically for working with databases in server-side applications.
        ii. It provides a fluent interface for interacting with databases, allowing developers to define models, query data, and perform database operations using Swift's type-safety and expressive syntax.
        iii. Fluent supports multiple database backends such as PostgreSQL, MySQL, SQLite, and more, enabling developers to choose the database that best fits their application requirements.
        iv. It handles tasks such as database migrations, schema creation, and data manipulation, abstracting away the complexity of working directly with raw SQL.
        v. Fluent seamlessly integrates with Vapor, making it easy to build database-driven web applications and APIs using Swift.
3. Code organization



a. When developing a software application, especially in web development frameworks such as Ruby on Rails, Laravel, Django, or Express, it's common to organize code into specific directories based on functionality. This structure helps in maintaining a clean codebase, making it easier for developers to navigate, understand, and extend the application. The typical directories you might encounter include models, controllers, jobs, and migrations like shuttle tracker.

2. Key Components

The shuttle tracker has the following controllers:

1. Analytics
    a. AnalyticsEntries
    b. AnalyticsEntry
2. AndroidRedirects
3. Announcement
    a. Announcements
4. Bus
    a. Buses
5. DataFeed
6. Log
    a. Logs
7. Milestone
    a. Milestones
8. Notifications
    a. NotificationDevice
    b. NotificationDevices
9. Redirects
10. Routes
11. Schedule
12. Stop
    a. Stops
13. SwiftUIRedirects
14. Version
15. WebDirects

Routes Controller:

1.
```swift
//
//  RoutesController.swift
//  Shuttle Tracker Server
//
//  Created by Gabriel Jacoby-Cooper on 1/12/24.
//

import Vapor

/// A structure that registers routes for shuttle routes.
/// - **Important**: This structure registers routes on the index path of its provided routes builder, so make sure to enclose it in a named routes group to avoid path collisions.
/// - **Remark**: In the context of this structure, the term "route" could refer to either an HTTP route or a shuttle route, depending on the local context.
struct RoutesController: RouteCollection {

    func boot(routes: any RoutesBuilder) throws {
        // In the context of this method, the term "route" refers to an HTTP route, not a shuttle route.
        routes.get(use: self.read(_:))
    }

    private func read(_ request: Request) async throws -> [Route] {
        // In the context of this method, the term "route" refers to a shuttle route, not an HTTP route.
        return try await Route
            .query(on: request.db(.sqlite))
            .all()
            .filter { (route) in
                return route.schedule.isActive
            }
    }

}

```

2.
```swift
return try await Route
    .query(on: request.db(.sqlite))
    .all()
    .filter { (route) in
        return route.schedule.isActive
    }
```

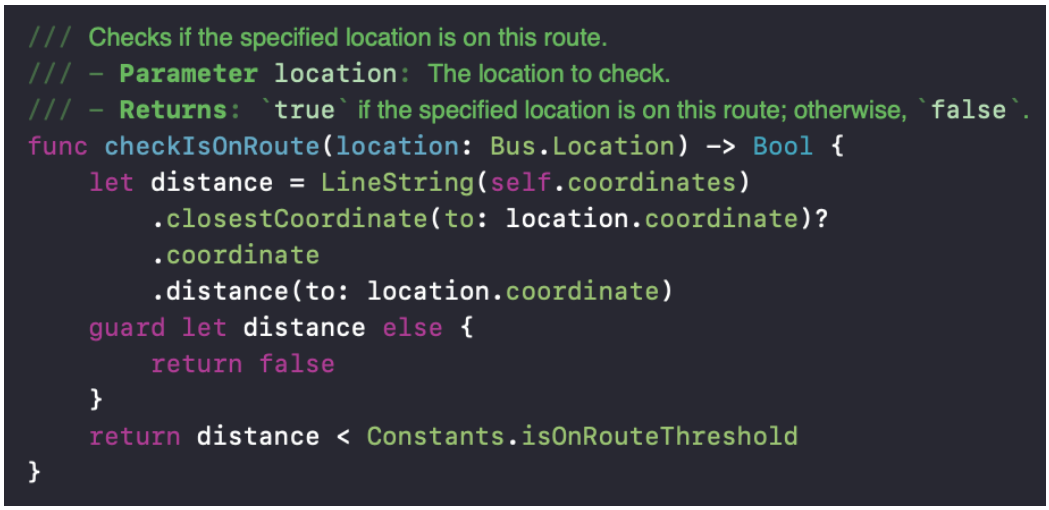
Example Model Controller

3.

```swift
import CoreGPX
import FluentKit
import JSONParser
import Turf
import Vapor

/// A representation of a shuttle route.
///
/// A route is represented as a sequence of geospatial coordinates.
final class Route: Model, Content, Collection {

    static let schema = "routes"

    let startIndex = 0

    var endIndex: Int {
        get {
            return self.coordinates.count
        }
    }

    @ID
    var id: UUID?

    /// The user-facing display name of this route.
    @Field(key: "name")
    var name: String

    /// The waypoint coordinates that define this route.
    @Field(key: "coordinates")
    var coordinates: [Coordinate]

    /// A schedule that determines when this route is active.
    @Field(key: "schedule")
    var schedule: MapSchedule

    /// The name of the color that clients should use to draw the route.
    @Field(key: "color_name")
    var colorName: ColorName

    init() { }
```

4.

```swift
/// Creates a route object from a GPX route with a custom name, schedule, and color name.
/// - Parameters:
///   - name: The user-facing display name of the route.
///   - gpxRoute: The GPX route from which to create a route object.
///   - schedule: The schedule for when the route will be active.
///   - colorName: The name of the color that clients should use to draw the route.
convenience init(name: String, from gpxRoute: GPXRoute, schedule: MapSchedule, colorName: ColorName) {
    let coordinates = gpxRoute.points.compactMap { (gpxRoutePoint) in
        return Coordinate(from: gpxRoutePoint)
    }
    self.init(
        name: name,
        coordinates: coordinates,
        schedule: schedule,
        colorName: colorName
    )
}
```

5. 

```swift
/// Checks if the specified location is on this route.
/// - Parameter location: The location to check.
/// - Returns: `true` if the specified location is on this route; otherwise, `false`.
func checkIsOnRoute(location: Bus.Location) -> Bool {
    let distance = LineString(self.coordinates)
        .closestCoordinate(to: location.coordinate)?
        .coordinate
        .distance(to: location.coordinate)
    guard let distance else {
        return false
    }
    return distance < Constants.isOnRouteThreshold
}
```

3. API Documentation

If your project exposes an API, provide detailed API documentation that includes:

1. Endpoints: List all the API endpoints, their URLs, and methods (GET, POST, etc.).
2. Parameters: Describe any parameters that can be passed to each endpoint, including data types and whether they are required or optional.
3. Responses: Document the possible responses for each endpoint, including status codes and data structures.