**Advisory Report**

### **Executive Summary**

Swisscom’s **TravelMate** platform lets employees work abroad for up to ten days a year, yet today neither staff nor tribe administrators can see whether those “**Open Days**” are used. The result is wasted quota, last-minute reallocations, and zero insight for planning.  
We analysed four implementation approaches and three technology stacks for each layer (backend, frontend, database) against nine non-functional requirements (performance, scalability, security, maintainability, cost, etc.). Decision matrices summarise the evidence.

* **Backend.** Extending the existing **Java / Spring Boot** codebase—optionally writing the new module in **Kotlin** for cleaner, better-tested code—scores highest on effort, risk and performance. A separate micro-service was ruled out as over-engineered for a feature of this size.
* **Frontend.** Although React or Vue win in a green-field comparison, the current admin portal is **Angular** with a shared component library; re-use avoids visual drift and duplicated effort.
* **Database.** PostgreSQL tops the green-field ranking, but the present system already runs **MongoDB**, which meets data-volume and reporting needs once properly indexed—so no second datastore is introduced.

**Recommendation:** Extend TravelMate in-place: add an “Open Days Tracking” module (written in Kotlin where practicable), surface the data in the existing Angular dashboard, and evolve the MongoDB schema. This path minimizes integration risk, leverages current CI/CD pipelines, and can be delivered inside the planned release window. Immediate next steps are to:

1. uplift unit-test coverage on affected Spring services,
2. remove the AnyOrg integration and replace it with another service.

This strategy delivers transparency for administrators, prevents unused-day waste, and positions TravelMate for future incremental enhancements without architectural change.

**1. Introduction**

**1.1 Context**

TravelMate is an internal business travel management system used at Swisscom to facilitate international work travel for employees. The application enables employees to work from an international office for up to 10 days per year, with costs covered by the organization. Travel days are centrally managed at the "tribe" level (a group of teams), and each tribe has a fixed annual quota (180 days) that can be allocated and booked by employees.

The current version of TravelMate does not offer visibility into whether allocated Open Days (designated travel days) were used. This results in inefficiencies such as underutilization of travel days, inability to reallocate unused days, and a lack of insight for administrators.

To address this gap, a new feature called Open Days Tracking is proposed. It aims to enhance transparency and administrative control by enabling the tracking, reporting, and reallocation of travel days.

**1.2 Purpose**

The purpose of this advisory report is to evaluate multiple technical solutions for implementing the Open Days Tracking feature. This includes a comparison of tools, languages, architectural approaches, and integration strategies. The report provides a recommendation based on key quality aspects such as scalability, maintainability, performance, security, and alignment with organizational constraints and development practices.

The target audience includes technical stakeholders (developers, software architects), product owners, and decision-makers responsible for feature planning and implementation within Swisscom.

**1.3 Scope**

This report focuses on advising the design and implementation of the Open Days Tracking feature in terms of:

* Backend architecture
* Frontend strategy
* Database
* External integrations
* System quality

This report does not provide a finalized technical design or project timeline but instead focuses on strategic architectural direction and technology choices.

**2. Requirements Overview**

The Open Days Tracking feature aims to improve transparency and control over the allocation and utilization of travel days within the TravelMate system. It must support both business goals and technical constraints while integrating smoothly with existing and external systems.

**2.1 Business Requirements**

* Provide a clear overview of which Open Days have been used, remain unused, or have been reallocated.
* Ensure tribe administrators (reviewers) have access to accurate and up-to-date information on Open Day usage.
* Allow administrators to manually reallocate unused Open Days to improve resource efficiency.
* Enable better planning and decision-making through reporting and visibility features.
* Reduce waste of allocated travel days by ensuring they are used effectively.

These requirements will form the basis for evaluating different architectural approaches and technology options in the following sections of this report.

**2.2 Non-functional requirements**

The implementation of the Open Days Tracking feature must meet several quality attributes that are critical to the long-term success, usability, and maintainability of the system. These are derived directly from the non-functional requirements defined in section 3.3 of the Software Requirements Specification (SRS) and will be used to evaluate architectural and technical decisions in the remainder of this report.

***2.2.1 Performance (SRS 3.3.1)***

* The dashboard displaying Open Days should load within 2 seconds for 95% of requests.
* Queries that determine Open Day status should complete in under 1.5 second for standard data loads (up to 250 travel requests per unit).
* API calls should not block UI operations and must be handled asynchronously.

***2.2.2 Reliability (SRS 3.3.2)***

* The system should maintain 99.5% uptime, ensuring availability for administrators during working hours.
* The feature must include tests to ensure nothing breaks.

***2.2.3 Availability (SRS 3.3.3)***

* The feature should be accessible 24/7, except during scheduled maintenance.

***2.2.4 Security (SRS 3.3.4)***

* Only authorized administrators should be able to reallocate unused days.
* User authentication and access control should follow Swisscom's internal security policies.

***2.2.5 Maintainability (SRS 3.3.5)***

* The system should have 80%+ unit test coverage to ensure code reliability.
* Tests should be automated, covering key functionalities such as Open Day status determination and API integrations.
* The feature should follow modular coding principles to allow easy modifications and improvements.

***2.2.6 Scalability (SRS 3.3.6)***

* The system should handle at least 10,000 Travels across multiple units without performance degradation.
* The architecture should support horizontal scaling if additional units or teams require tracking.

***2.2.7 Compliance (SRS 3.3.7)***

* The system must comply with Swisscom's internal IT policies regarding data handling and retention.

**3. Solution Options & Evaluation**

This section evaluates four potential approaches to implementing the Open Days Tracking feature. The comparison is based on key quality attributes (security, scalability, performance, maintainability, privacy, and reliability) and considers the current state of the TravelMate system: a well-structured backend with low test coverage.

**3.1 Option A – Extend the existing TravelMate Backend**

**3.1.1 Description**  
Implement the feature directly in the existing Java Spring Boot backend of TravelMate. Extend current models and services to support Open Day status tracking. UI enhancements would be made within the existing Angular application.

**Pros**

* Minimal overhead — no new systems, infra, or deployment processes required.
* Keeps logic close to existing travel and booking features.
* Full access to current data models and authentication flow.
* Suitable for a feature of this size and complexity.

**Cons**

* Risk of regression due to low test coverage.
* Requires careful validation and possibly manual QA.
* Changes may need to be coordinated with others working in the same codebase.

**3.1.2 Assessment**

|  |  |
| --- | --- |
| **Quality Aspect** | **Evaluation** |
| Security | Reuse existing auth. |
| Scalability | Limited in long term, but acceptable for feature size. |
| Performance | Efficient, no inter-service calls. |
| Maintainability | Depends on clean integration and added tests. |
| Compliance | Existing data protections apply. |
| Reliability | Some risk due to limited tests; mitigable with manual QA. |
| Availability | Shares runtime with the existing TravelMate backend. Any issues in main system will affect this feature. |

**When to choose:**  
This is the preferred option if the affected areas of code are stable and can be extended with minimal disruption. Given the feature’s scope, this is the most pragmatic choice if properly tested.

**3.2 Option B – Modular Subsystem Within TravelMate**

**3.2.1 Description**  
Create a self-contained module or package inside the TravelMate backend, with clearly separated services, APIs, and data models. This isolates the logic somewhat while remaining part of the same codebase and deployment.

**Pros**

* Better testability and structure than Option A.
* Still benefits from shared authentication and data.
* More modular design supports future extensions.

**Cons**

* Slightly higher development effort and overhead.
* Still shares deployment and runtime with the main system.

**3.2.2 Assessment**

|  |  |
| --- | --- |
| **Quality Aspect** | **Evaluation** |
| Security | Reuses existing auth. |
| Scalability | Medium – still coupled but logically separated. |
| Performance | Local service calls; no latency overhead. |
| Maintainability | Better test boundary than Option A. |
| Compliance | Existing data protections apply. |
| Availability | Shares runtime with the existing TravelMate backend. Any issues in main system will affect this feature. |

**When to choose:**  
Use this option if the TravelMate codebase is stable but complex, and if you want to introduce testable, modular structure while avoiding new infrastructure.

**3.3 Option C – Fully Decoupled Microservice**

**3.3.1 Description**  
Implement the feature as a standalone service its own API and database. Integrate with TravelMate via API calls or embedded UI components.

**Pros**

* Completely isolated from the main system, no regression risk.
* Allows use of modern frameworks and clean testing practices.
* Scales independently if needed in future.

**Cons**

* Overhead is disproportionate to feature size.
* Requires separate deployment, monitoring, and authentication handling.
* Introduces system complexity for a relatively small gain.
* Must migrate a lot of existing logic as well.

**3.3.2 Assessment**

|  |  |
| --- | --- |
| **Quality Aspect** | **Evaluation** |
| Security | High – can be isolated tightly. But have to reimplement the auth |
| Scalability | Strong, but likely overkill. |
| Performance | Slightly lower due to cross-service latency. |
| Maintainability | Very high – independent codebase. |
| Compliance | Easier to enforce tight data boundaries. |
| Availability | High — the feature is isolated in its own microservice. Failures in TravelMate or unrelated systems do not impact availability of Open Days Tracking. |

**When to choose:**  
Best suited for larger features, or if TravelMate is unstable or hard to extend.

**3.4 Option D – Prototype-First (for Feasibility Validation)**

**3.4.1 Description**  
Develop a lightweight prototype (e.g., FastAPI, NestJS) to validate integration with AnyOrg and explore the UI/UX. This could serve as a technical spike or an early demonstration for stakeholders.

**Pros**

* Rapid experimentation.
* Helps de-risk tricky parts (e.g., flaky API behavior).

**Cons**

* Not production ready.
* Duplicated effort if discarded.

**3.4.2 Assessment**

|  |  |
| --- | --- |
| **Quality Aspect** | **Evaluation** |
| Security | Minimal, unless explicitly built in. |
| Scalability | Not intended for production load. |
| Performance | Basic-level acceptable. |
| Maintainability | Prototype code often lacks structure/tests. |
| Compliance | Depends on data handling approach. |
| Availability | Since it is a POC it won’t be available for actual use. |

**When to choose:**  
Use when you need to validate assumptions quickly. This option is a backup plan when something comes up that makes the other points impossible.

**3.5 Conclusion**  
After evaluating all four options, the team decided to implement the Open Days Tracking feature directly within the existing TravelMate backend with a modular approach in mind (Option B). This so changes to the existing logic won’t have direct effect on this module, since the existing models won’t be directly used.

One prerequisite for this decision is that the existing functionality must be properly tested before integration begins. The current backend has low test coverage in key areas, so improving this is necessary to avoid regressions and ensure safety.

Additionally, the team expressed a clear preference to keep this feature accessible and maintainable within the existing TravelMate system. There is no appetite for managing a separate application or microservice for a feature of this size. Instead, the goal is to build it in a way that fits naturally into the current architecture with a modular design.

In summary, extending the existing backend with a modular approach is the most pragmatic and aligned choice, provided we invest in test coverage and follow clean, modular implementation practices.

**4. Technology Evaluation**

### **4.1 Backend Technologies**

#### **Greenfield Perspective**

If the Open Days Tracking feature were developed independently from the existing system, selecting the right backend technology would be a critical architectural decision. In this analysis, we compare **Java (Spring Boot)**, **Kotlin (Spring Boot or Ktor)**, and **Node.js (with NestJS)**—three mature and widely used platforms with strong ecosystems [4].

These technologies were selected because they represent three distinct backend paradigms:

* Java with Spring Boot provides a well-established enterprise-grade framework with deep support for security, transactions, and integrations.
* Kotlin builds on the Java ecosystem with a more expressive syntax, better safety features, and improved developer ergonomics.
* Node.js with NestJS offers a highly asynchronous, TypeScript-based approach that aligns well with frontend-heavy teams and fast-paced API development.

The goal is to determine which would be most suitable in a greenfield context based on non-functional criteria such as performance, scalability, and maintainability.

#### **4.1.1 Framework Comparison: Java vs. Kotlin vs. Node.js**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Criterion** | **Java + Spring Boot** | **Score** | **Kotlin (Spring Boot / Ktor)** | **Score** | **Node.js + NestJS** | **Score** |
| Performance | JVM JIT + tuned GC; strong performance benchmarks [1] | 4 | Same JVM base; negligible std-lib overhead | 4 | Excellent async I/O; weaker CPU-bound performance [3] | 3 |
| Scalability | Scales well vertically; mature support for horizontal scale | 4 | Same JVM infrastructure | 4 | High concurrency; scalable via event loop | 4 |
| Maintainability / Testability | Verbose syntax, but strong patterns available | 3 | Null-safety, fewer LOC, more expressive [2] | 4 | Good modularity; less standardization than JVM | 3 |
| Security / Privacy | Spring Security, mature policy frameworks | 4 | Inherits Spring stack or similar | 4 | Requires extra effort; more manual setup | 3 |
| Reliability / Operability | Mature JVM stack; well-known monitoring tools | 4 | Same JVM tools and guarantees | 4 | Needs more manual setup for observability | 3 |
| Team cost & ramp-up | Widely known; in-house expertise already available | 4 | Low barrier for Java teams; natural next step | 4 | Steeper ramp-up for backend TypeScript | 3 |
| Prototype speed | Slower due to boilerplate; tooling helps | 3 | Concise syntax, quick interop with Java | 4 | Fastest dev cycles; ideal for rapid iteration | 5 |

**Average scores:**  
Kotlin: 4.00  
Java: 3.71  
Node.js: 3.43

Kotlin offers the best balance of maintainability, safety, and performance in a greenfield setting. It retains all the strengths of the JVM ecosystem while reducing boilerplate and improving code clarity. Java remains a reliable and stable choice, especially for teams already using Spring Boot. Node.js is excellent for fast development and async-heavy applications but has less built-in support for enterprise features like security and monitoring.

#### **4.1.2 Within the Existing System and Conclusion**

The TravelMate backend is currently built in Java using Spring Boot. The codebase is well-structured, though it has low test coverage in some areas. For this reason, integrating Open Days Tracking directly into the existing backend is the most practical and least disruptive path.

I discussed this with the backend team, and we agreed that continuing with Java is the safest and most efficient approach. However, there is interest in introducing Kotlin selectively for new modules. Kotlin allows cleaner syntax and improved maintainability without requiring a full rewrite, since it runs seamlessly alongside Java on the JVM. Yet the entire team does not want to include Kotlin because it has a “steeper” learning curve.

Changing to Node.js or another ecosystem would involve substantial migration effort and introduce risks that aren’t justified by the scope of this feature.

In summary, while Kotlin or Node.js might be attractive in a greenfield scenario, sticking with Java (potentially enhanced with Kotlin) is the right choice within the current architecture.

**4.2 Frontend Technologies**

**Greenfield Perspective**  
If the Open Days Tracking feature were to be developed as a standalone admin application, choosing the right frontend framework would be a key architectural decision. React, Vue, and Angular are included in this comparison because they are the most widely adopted frameworks for building modern web applications, each with strong community support, proven scalability, and long-term viability.

These three represent distinct design philosophies:

* React offers flexibility and a rich ecosystem with minimal assumptions about application structure.
* Vue emphasizes simplicity and rapid development with a more approachable learning curve.
* Angular provides a complete, structured solution with integrated tooling and strong enterprise features.

They are compared here to determine which would be best suited for building a maintainable, performant admin interface in a greenfield scenario, before narrowing the decision based on TravelMate’s existing technology stack.

**4.2.1 Framework Comparison: React vs. Vue vs. Angular**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Criterion** | **React** | **Score** | **Vue 3** | **Score** | **Angular** | **Score** |
| Performance / bundle size | React 18 streaming; Preact option [4] | 4 | ≈30 kB core gzip [4] | 4 | Larger ≈75 kB; zone.js overhead [4] | 3 |
| Learning curve | JSX & hooks are powerful but steeper to learn | 3 | SFC + Options API are intuitive | 4 | RxJS, DI, and module system require more onboarding | 2 |
| Ecosystem & community | Largest ecosystem; strong community support [5] | 5 | Smaller, growing ecosystem [5] | 3 | Mature and enterprise-focused ecosystem [6] | 4 |
| Scalability & architecture | Needs added patterns like Redux for structure | 3 | Flexible, some architectural support via Vuex | 3 | CLI, DI, and modules support scalable design | 5 |
| Maintainability | Highly flexible but team-dependent conventions | 4 | Lightweight structure, less formal guidance | 3 | Strong consistency, built-in testing and style guide | 4 |
| TypeScript support | First-class support, but verbose generics | 4 | Optional integration | 3 | Mandatory use, including template typing | 5 |
| Prototype speed | Fast reload with CRA or Vite | 4 | Fastest dev loop with Vite + SFC | 5 | Slower setup, heavier configuration | 2 |

**Average scores:**  
React: 3.86  
Vue 3: 3.57  
Angular: 3.57  
  
React stands out for its flexibility, fast development cycles, and broad ecosystem. It is a strong choice for greenfield projects focused on dashboards or highly interactive UIs. Vue is attractive for teams that prioritize simplicity and fast onboarding, especially for smaller applications. Angular offers built-in architectural guidance, testing tools, and a scalable structure that is well-suited for larger enterprise-grade applications.

**4.2.2 Within the Existing System**  
Although React and Vue offer appealing benefits in a greenfield context—particularly for speed, flexibility, and ease of use—those options are not viable within the current TravelMate system. The frontend is built with Angular and relies heavily on an internal component library developed specifically for Angular applications.

I discussed the framework choice with the senior frontend developer, and we agreed that despite the potential advantages of other technologies, we are effectively required to continue using Angular. It is the company standard, and the component library is tightly coupled to Angular’s architecture. Switching to a different framework would mean abandoning the shared design system, duplicating effort, and significantly increasing maintenance overhead.

For these reasons, Angular is the only practical choice. It ensures consistency in UI and development practices, supports seamless integration with the existing dashboard, and aligns with internal frontend development standards.

In summary, while Angular may not be the most lightweight or flexible option, it is the correct and necessary one in the current environment.

### 4.3 Database Technologies

#### **Greenfield Perspective**

The Open Days Tracking feature involves structured, interrelated entities: tribes, employees, travel records, and open day statuses. It must enforce business rules (e.g., a 180-day tribe limit), support complex filtering (e.g., by date or usage status), and deliver reporting (e.g., unused days per tribe). These requirements strongly point to the need for a **relational data model.**

#### **4.3.1 Relational vs. Non-Relational Databases**

A foundational architectural decision is the choice between **relational databases** (e.g., PostgreSQL, MySQL/MariaDB) and **non-relational/document-oriented databases**(e.g., MongoDB). This decision should be based on the structure of the data, consistency requirements, and reporting needs.

**Relational databases** are inherently well-suited for applications like Open Days Tracking, which:

* Require **structured schemas and enforceable integrity constraints**
* Depend on **relationships** between multiple entities (e.g., employees belong to tribes, tribes have quotas)
* Demand **complex queries** for filtering, grouping, and aggregation
* Must uphold **data consistency** across multiple operations

|  |  |  |
| --- | --- | --- |
| **Criterion** | **Relational (SQL)** | **Non-Relational (NoSQL - MongoDB)** |
| **Data relationships** | Strongly supported via joins, foreign keys | Must be enforced manually |
| **Schema enforcement** | Strict, declarative constraints | Flexible, but validation must be external |
| **Query & Reporting** | Powerful SQL, joins, CTEs, window funcs | Aggregation pipeline less intuitive |
| **Data consistency** | ACID-compliant | Consistency handled in application logic |
| **Suitability for this use case** | Excellent – structured data, clear rules | Weak – high maintenance overhead |

In contrast, **non-relational databases** like MongoDB emphasize flexibility and speed, but delegate all integrity enforcement and relationships to the application layer. While this allows for rapid development, it increases the risk of **data inconsistency** and makes **reporting more complex**.

Given the need for **data integrity, structured relationships, and robust reporting,** a relational database is clearly the better fit. MongoDB is therefore excluded from greenfield consideration, although it may still be viable in the current system for pragmatic reasons.

### **4.3.2 PostgreSQL vs. MySQL/MariaDB — Summary and Scoring**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criterion** | **PostgreSQL** | **Score** | **MySQL / MariaDB** | **Score** |
| Schema & Integrity | Rich constraints, foreign keys, strong types | 5 | Good support, but fewer advanced constraints | 4 |
| Query Capabilities | CTEs, window functions, JSONB | 5 | Standard SQL; fewer analytical features | 3 |
| Performance | High with tuning and indexing | 4 | Strong defaults, stable general performance | 4 |
| Scalability | Vertical scaling; horizontal via extensions | 3 | Partitioning support less advanced | 3 |
| Maintainability | Large ecosystem; steep learning curve | 3 | Easier for small-to-medium teams | 4 |
| Tooling & Ecosystem | Excellent, especially for analytics | 5 | Good, especially for admin and CMS use cases | 4 |

**Average Scores:**

* **PostgreSQL: 4.17**
* **MySQL / MariaDB: 3.67**

PostgreSQL scores higher overall, particularly in areas critical for the Open Days Tracking feature—namely schema enforcement, advanced querying, and tooling for analytics. MySQL/MariaDB still performs well and may be easier for teams with limited SQL depth, but PostgreSQL is a better fit for complex, evolving features in a greenfield setup.

**4.3.3 Conclusion:**  
I brought up the trade-offs between relational and non-relational databases in a discussion with the backend engineer who originally chose MongoDB. He acknowledged the strengths of a relational approach—especially for enforcing consistency and supporting complex reporting—but still preferred the flexibility MongoDB offers. After considering these points as a team, we decided to proceed with the existing MongoDB setup. While PostgreSQL would be the preferred choice in a greenfield scenario, switching databases isn’t planned. With the right schema design and indexing, MongoDB remains a workable and efficient option for this feature.

**5. Recommendation**

Based on the analysis of both greenfield and existing-system scenarios, the following approach is recommended for implementing the Open Days Tracking feature.

**5.1 Recommended Approach**

Given that the feature is relatively small in scope, and that the existing TravelMate system is well-structured (though with low test coverage), the best path forward is to implement the feature within the existing TravelMate backend and frontend, using the technologies already in place:

* **Backend:** Although the TravelMate backend is well-structured, it is not modular, and the Open Days Tracking feature is tightly coupled to existing functionality. In an ideal scenario, the feature would be implemented using a modular design or isolated service boundary to improve maintainability and testability. However, given the current structure and time constraints, this may not be feasible without significant refactoring. Therefore, the most realistic approach is to extend the existing backend carefully, while applying modular principles within the implementation where possible. Also making sure to refactor where possible. To do this we need to increase test coverage within the system. Kotlin may be selectively introduced to support a cleaner, more modern implementation.
* **Frontend:** Integrate the Open Days Tracking views into the current Angular application using the internal component library. This ensures visual consistency and leverages shared services.
* **Database:** Ideally, a relational database such as PostgreSQL would be the first choice for this type of feature due to its strong support for relationships, constraints, and reporting capabilities. However, given that the existing system uses MongoDB, and the feature's data structure is relatively simple, MongoDB will suffice. Extend the current MongoDB schema to accommodate the Open Day status fields. Use indexing and embedded documents where appropriate to maintain query performance. MongoDB can fulfill the feature's non-functional requirements, including performance, scalability, and data privacy, when properly structured and indexed. It supports fast read/write operations, flexible schema evolution, and integration with existing authentication and access controls.

This approach minimizes integration and infrastructure overhead, leverages existing deployment pipelines and team knowledge, and supports all the required non-functional goals: performance, privacy, maintainability, and scalability.

This approach minimizes integration and infrastructure overhead, leverages existing deployment pipelines and team knowledge, and supports all the required non-functional goals: performance, privacy, maintainability, and scalability.

**6. Implementation Advice**

Although the recommended solution minimizes architectural disruption by extending the existing TravelMate backend, several implementation concerns must be addressed early to ensure maintainability, testability, and compliance with Swisscom’s internal policies.

**6.1 Refactor and Centralize User Enrichment Logic**

Due to internal privacy policies, TravelMate is not allowed to persist data from users who are not actively using the system. As a result, multiple parts of the backend rely on enrichment logic to dynamically retrieve employee data from external microservices such as AnyOrg or WFIDB.

Currently, these enrichment methods are scattered across various services, making them hard to reuse and increasing coupling. As Open Days Tracking will depend on this same logic, it is strongly advised to refactor these enrichers into a centralized, cohesive module. This will improve long-term maintainability and promote reuse across different features, while aligning with software design principles such as high cohesion and low coupling.

**6.2 Align Authorization Strategy via Polymorphic Access Layer**

WFIDB introduces a different authorization mechanism compared to other microservices, which currently rely on JWT-based access. While both approaches aim to authenticate API requests, their token formats and validation flows differ.

To avoid duplicating logic and breaking separation of concerns, implement a polymorphic access layer that abstracts the authorization mechanism. This allows the shared logic to be reused where applicable, while letting specific implementations diverge where necessary. This improves clarity and reduces future rework as additional microservices are onboarded.

**6.3 Improve Test Coverage Before Integrating New Features**

The TravelMate backend has low unit-test coverage, and the Open Days Tracking feature is highly dependent on existing logic that is currently untested. This introduces high risk: changes elsewhere in the system may unintentionally affect the feature’s behavior.

Before implementing new functionality, automated tests should be added for existing modules that will be reused or modified. This includes:

• Enrichment logic

• Tribe and booking-related services

• Date and quota allocation components

Without proper test scaffolding, regressions will be difficult to detect and diagnose. Proactively adding tests reduces integration risk and improves confidence in future releases.

**6.4 Implement Feature Flag Support via Internal Microservices**

To ensure controlled rollout and safe integration of the Open Days Tracking feature, it is essential to support feature toggling. At this time, the TravelMate backend does not yet support integration with Swisscom’s internal microservices for managing feature flags.

To enable progressive delivery and isolate unfinished or restricted functionality, we advise implementing a lightweight integration with the internal feature flag service. This will allow the team to:

• Enable or disable the feature per environment (e.g., staging vs. production)

• Gradually roll out the feature to specific user groups or teams

• Prevent incomplete functionality from being exposed to users prematurely

The implementation should abstract the feature flag check behind a simple utility interface, allowing feature logic to be toggled without scattering conditional checks throughout the codebase. This also improves testability and ensures that the presence or absence of the feature flag logic does not tightly couple core business logic to infrastructure-specific services.

Without this mechanism, there is no safe way to merge or deploy the feature incrementally — which increases deployment risk and hinders agile delivery.

**6.5 Refactor DatesDashboardComponent Before Extension**

The dates-dashboard.component.ts file currently contains over 1000 lines of code and combines multiple responsibilities in a single component. This complexity makes it difficult to maintain, extend, and test.

Since the Open Days Tracking feature must be implemented within this component, it is necessary to refactor it into a more modular structure before introducing new logic. However, this refactoring also introduces risk: the component plays a critical role in the system, and changes at this scale can easily introduce regressions or unintended side effects.

Despite these risks, postponing the refactoring would lead to even more complexity, reduce code quality, and increase long-term maintenance cost. To balance safety and sustainability, the refactoring should be approached carefully, with proper test coverage and review.

### **6.6 Conclusion**

All implementation concerns described in this chapter were discussed with my team. We agreed on the need to refactor the enrichment logic, improve test coverage, support multiple authorization strategies, and introduce feature flag support to isolate the Open Days Tracking functionality.

However, the proposed refactoring of the dates-dashboard.component.ts was not accepted. The component is actively being modified by the team during the timeframe of my project, and introducing structural changes now is considered too risky. In recent weeks, there have been several challenges in communication with stakeholders, and the team is hesitant to take on any changes that could delay the delivery or cause visible issues. Therefore, the decision was made to postpone the refactoring until after my project, in order to reduce project risk and maintain stakeholder confidence.

**References**

[1] A. Beaudouin, “A 2024 benchmark of main Web API frameworks,” Okami101 Blog, Dec. 26, 2023. [Online]. Available: <https://blog.okami101.io/2023/12/a-2024-benchmark-of-main-web-api-frameworks/>

[2] Dan, “Kotlin: A massive leap forward,” ProAndroidDev (via Medium), Feb. 9, 2018. [Online]. Available: <https://proandroiddev.com/kotlin-a-massive-leap-forward-78251531f616>

[3] “NestJS vs Spring Boot performance,” Restack.io, 2023. [Online]. Available: <https://www.restack.io/p/nestjs-vs-spring-boot-performance>

[4] Ramotion, “Top 10 backend programming languages to use in 2023.” [Online]. Available: <https://www.ramotion.com/blog/backend-languages/>

[5] GeeksforGeeks, “React vs Angular vs Vue: Which framework is the best?” 2024. [Online]. Available: <https://www.geeksforgeeks.org/react-vs-angular-vs-vue-which-framework-is-the-best/>

[6] Radixweb, “Angular vs React vs Vue: Which frontend framework to choose in 2023?” 2023. [Online]. Available: <https://radixweb.com/blog/angular-vs-react-vs-vue>

[7] ViteJS, Angular Documentation, React 18 Release Notes – for performance and tooling benchmarks.