Garbage Collection Area
24/11/2025

Comparative Study:

Technology choices for an online automation platform

This study aims to find the best technologies and practices to develop, and deploy, a multi-platform light-weight automation application.

Contents

1. Introduction

As to define the best suited technologies for an online lightweight automation application, we first need to define the key features that such application would need to have.

- Basic authentication to register an account
- Ability to connect to third-party providers using OAuth2.
- Ability to detect changes in third-party API returns.
- Ability to listen to third-party API webhooks.

Thus we can derive the key features needed for technologies well fitted for such application.

The backend language should have a wide ecosystem of already implemented, or easy to implement, OAuth2 providers and consumers. It need to be fast to compile, easy to run, and light in server resources utilization. It should also have great support for common backend architecture (REST, CRUD) and API describing standards like OpenAPI, preferably v3 to leverage 'possibly NULL' capabilities.

The frontend framework should be quick to prototype and iterate with. It should provide an easy way to setup a OpenAPI consumer client. Ideally the choose framework should be bundled with an easy web server router and facilitate the handling of session, cookies and user data.

2. Backend language

2.1. Explored solutions

2.1.1. Rust

With the rapid growth of the Rust ecosystem and community, Rust can now be considered a valid candidate for backend development. However, it still has several practical drawbacks for this specific kind of web API project.

A first limitation is the current state of Rust ORMs. Diesel is a popular choice, but its incomplete or limited async support makes it less suitable for typical web API workloads, where asynchronous database access is a core requirement. SeaORM is another widely used option that does provide async and more dynamic capabilities, but it is significantly less type-safe, which undermines one of Rust's main advantages: strong compile-time guarantees.

Rust's asynchronous model itself also introduces complexity. The guarantees provided by the compiler and the async runtime are valuable, but they come at the cost of a steeper learning curve and more boilerplate compared to simpler concurrency models. This can slow down iteration speed during the early phases of the project, when requirements change quickly and rapid prototyping is important.

On the positive side, Rust has no garbage collector, which allows for efficient and predictable memory usage. It also offers excellent facilities for error handling, encouraging explicit and exhaustive handling of failure cases. This can lead to very robust backend codebases in the long term.

Overall, while Rust provides strong performance characteristics and safety guarantees, the relative immaturity and fragmentation of its web and ORM ecosystem, combined with the complexity of its async model, make it less suitable for a lightweight, fast-moving automation backend compared to other options.

2.1.2. TS/JS (Node.js with Express)

TypeScript and JavaScript, running on Node.js with an Express-based stack, are also natural candidates for building a lightweight automation backend. The ecosystem around Node is extremely mature for HTTP services and integrations with third-party APIs.

On the positive side, Express (or similar frameworks such as Fastify or NestJS) provides a straightforward way to build REST and CRUD endpoints. The Node ecosystem offers a large number of SDKs and client libraries for popular third-party services, including ready-made OAuth2 integrations. This makes it easy to

add new providers and to consume their APIs, which is central for an automation platform that must connect to many external services.

Using TypeScript on top of Node improves maintainability by adding static typing and better tooling (auto-completion, refactoring support, type-level contracts). Existing tools and libraries can also help generate and consume OpenAPI specifications, bridging the gap between strongly typed server code and client code generation.

However, there are trade-offs. The sheer size and fragmentation of the JavaScript ecosystem can introduce maintenance and security concerns: many dependencies are small, highly specialized packages, often maintained by individuals, which can increase the risk of supply-chain issues or abandoned libraries. Managing and keeping this dependency tree up-to-date requires ongoing attention.

From a runtime perspective, Node.js uses a single-threaded event loop model with asynchronous I/O, which is efficient for I/O-bound workloads but makes CPU-bound tasks more challenging without additional worker infrastructure. Although this is not a critical drawback for an IFTTT-style automation service (which is mostly I/O-bound), it adds some complexity if more intensive processing is introduced later.

Finally, while JavaScript is widely known, the flexibility of the language and the lack of enforced conventions can lead to inconsistent codebases without strict internal guidelines. Even with TypeScript, it is easy to fall back to untyped or loosely typed patterns if discipline is not maintained.

In summary, a Node.js + Express + TypeScript stack offers excellent library support, especially for HTTP and third-party integrations, and can be very productive for rapid development. Nevertheless, the ecosystem's fragmentation and the operational overhead of managing many dependencies make it somewhat less attractive compared to more opinionated and self-contained backend stacks for this particular project.

2.1.3. GoLang (Go)

Go is a strong candidate for the backend of a lightweight automation platform because it directly addresses most of the requirements identified earlier: a mature HTTP ecosystem, good OAuth2 support, efficient concurrency for dealing with multiple third-party services, and low operational overhead.

On the ecosystem and OAuth2 side, Go benefits from a long history as a backend-oriented language. The standard library offers a robust `net/http` package, and common third-party packages (for example `golang.org/x/oauth2`) provide ready-made OAuth2 client implementations. Many major SaaS providers publish reference examples or SDKs in Go, which makes the integration of new providers relatively straightforward. Even when no dedicated SDK is available, Go's HTTP primitives make it easy to implement OAuth2 flows and to consume REST APIs directly.

Performance and resource usage are also well aligned with the needs of an automation service. Go compiles to a single static binary per service, with very fast compilation times compared to many other compiled languages. Deployment is simplified: there is no need for a separate runtime or virtual machine, and the resulting binaries are lightweight enough to run efficiently in containers or on small virtual machines. The built-in garbage collector is optimized for server workloads and, combined with Go's relatively simple type system, generally offers predictable performance without extensive tuning.

A central design goal of Go is server-side concurrency, which is especially relevant for an "IFTTT-like" automation application that must frequently call multiple third-party APIs, process webhooks, and schedule polling tasks. Goroutines and channels provide a simple yet powerful concurrency model, allowing the backend to handle a large number of concurrent HTTP requests, webhook callbacks, and scheduled jobs without complex boilerplate. This helps in implementing features like "detect changes in third-party API returns" via periodic polling and parallel requests, while keeping code readable and maintainable.

Regarding common backend architectures, Go has a mature ecosystem of web frameworks and libraries for REST and CRUD APIs (such as Echo, Gin, Chi, Fiber, etc.). These frameworks integrate cleanly with OpenAPI v3 tooling, making it possible to generate OpenAPI specifications from code or, conversely, generate server

stubs from an existing specification. There are also libraries dedicated to request/response validation, input binding, and error handling that align well with a typical web API stack.

Go's tooling further reinforces its suitability: `go test` encourages unit and integration testing from the start; `go fmt` and `go vet` enforce a consistent code style and catch common mistakes; and the module system (`go mod`) simplifies dependency management. This strong tooling support reduces friction during development and helps maintain code quality as the project grows.

In contrast with languages that emphasize complex type systems or metaprogramming, Go intentionally keeps language features minimal and explicit. This makes onboarding new contributors easier: developers familiar with general backend concepts can typically become productive in Go in a relatively short time. For a project that may evolve quickly and require frequent iteration on API contracts and integrations, this simplicity is a significant advantage.

Finally, Go's adoption in cloud-native environments (Kubernetes, Docker, many CNCF projects) makes it a natural fit for a distributed automation platform. Native support for building small container images, good observability libraries (logging, metrics, tracing), and first-class support in most cloud providers' tooling reduce operational complexity, which is critical for an application expected to run continuously and reliably while orchestrating many external services.

Taken together—mature HTTP and OAuth2 libraries, efficient concurrency, strong tooling, straightforward deployment, and good alignment with modern cloud-native practices—Go provides a well-balanced compromise between performance, developer productivity, and simplicity, making it a well-justified choice as the backend language for this automation application.

2.2. Conclusion

Considering all these aspects together, Go offers the most balanced compromise for this project. Rust provides strong safety guarantees and performance, but its web and ORM ecosystems—and the complexity of its async model—slow down iteration. The Node.js + TypeScript stack has excellent library coverage and fast prototyping, but comes with a fragmented dependency ecosystem and more operational overhead over time. Go, on the other hand, combines a mature and focused HTTP ecosystem, straightforward concurrency with goroutines, simple deployment as static binaries, and solid tooling around testing and OpenAPI integration. This balance between performance, simplicity, and maintainability is why Go was ultimately chosen as the backend language for the automation application.

3. Frontend framework

3.1. Explored solutions

3.1.1. Angular

Angular is a full-fledged framework maintained by Google, designed to provide a highly structured approach to building large-scale web applications. It includes a powerful CLI, a dependency injection system, built-in routing, form handling, and a very opinionated architecture based on modules, components, and services.

On the positive side, Angular offers a strong sense of structure from the beginning of a project. The framework enforces conventions around how features should be organized, which can be useful for very large teams or when multiple applications must follow the same architectural patterns. TypeScript support is first-class, and the template system integrates well with reactive patterns through RxJS. For an enterprise-grade application, Angular's batteries-included philosophy can reduce the need to pick and assemble many third-party libraries.

However, this level of structure and abstraction also comes with drawbacks for a lightweight automation platform. The learning curve is relatively steep: developers must understand concepts like modules, decorators, change detection strategies, dependency injection, and RxJS streams before they can be fully productive.

The template syntax and the amount of boilerplate required to set up even simple forms or components can slow down early prototyping.

From the perspective of consuming OpenAPI specifications and integrating with a Go backend, Angular does provide workable solutions (for example code generators that produce strongly-typed clients). Nevertheless, much of the surrounding tooling is optimized for large, monolithic frontends, whereas this project benefits more from a lean, rapidly evolving codebase. The framework's fixed structure sometimes feels heavier than necessary for a relatively simple configuration-driven UI that primarily orchestrates API calls and forms.

In summary, Angular is robust and well suited for big, highly structured enterprise applications, but its verbosity, steep learning curve, and heavy-weight architecture make it less attractive for a fast-moving, lightweight automation frontend.

3.1.2. Vue

Vue is a progressive framework that focuses on approachability and incrementally adoptable architecture. It allows developers to start with simple, template-based components and evolve towards a more structured setup as the application grows (using single-file components, Vue Router, Pinia, etc.).

One of Vue's key strengths is its gentle learning curve: the template syntax is close to plain HTML, and the reactivity model is easy to grasp for developers with basic JavaScript experience. This can be beneficial when onboarding contributors who are less familiar with modern frontend frameworks. The ecosystem includes official solutions for routing and state management, which helps maintain consistency across projects.

Vue can integrate well with REST backends and OpenAPI-generated clients. Single-file components make it straightforward to colocate template, logic, and styles. For an automation application, this allows building clear, reusable components for things like "connect provider", "configure trigger", or "configure action" widgets. The Composition API adds a more explicit and composable way to share logic across components, which is useful for handling cross-cutting concerns such as authentication state or API error handling.

However, Vue's ecosystem is somewhat less dominant in the tooling space than React's. While there are good integrations for TypeScript and OpenAPI, many third-party libraries, UI kits, and example codebases still focus primarily on React first. For a project that relies heavily on quickly assembling existing building blocks (form libraries, schema-driven UIs, code generators, etc.), this means that some solutions appear later for Vue or are less actively maintained.

Furthermore, the progressive nature of Vue can be a double-edged sword: different teams or contributors might adopt different patterns (Options API vs Composition API, various state management approaches), which can lead to inconsistencies unless strict conventions are enforced from the beginning.

Overall, Vue is an attractive option with an approachable learning curve and solid tooling, but the relative scarcity of first-class ecosystem support compared to React makes it a slightly weaker fit for an automation platform that must lean heavily on existing libraries and integrations.

3.1.3. React

React is a component-based library focused on building user interfaces using a declarative model. Rather than being a full "framework", React provides the view layer and lets developers assemble the rest of the stack (routing, state management, data fetching) from a large ecosystem of third-party libraries.

For this project, React's main advantage lies in that ecosystem. Many tools and libraries are designed around React first: form frameworks, UI component libraries, headless components, and schema-driven form generators. In the context of a lightweight automation platform, this is particularly valuable for building complex configuration screens and wizards that guide users through connecting providers, defining triggers, and setting up actions.

React works well with TypeScript, enabling strongly-typed components and hooks. Combined with OpenAPI tooling, this makes it possible to generate fully typed API clients and reuse those types end-to-end: from Go backend models, to OpenAPI definitions, to TypeScript types in the frontend. This alignment significantly

reduces the risk of mismatches between backend responses and frontend expectations, and improves refactorability as the API evolves.

The modern React ecosystem also provides well-established patterns for data fetching (e.g. hooks-based libraries), client-side routing, and state management. This makes it straightforward to implement features like:

- authenticated routes and redirects,
- session management and token refresh logic,
- optimistic UI updates when modifying automations or connections,
- reusable "resource list / detail" patterns for CRUD-style configuration screens.

From a developer-experience perspective, React's functional component model and hooks offer a flexible way to encapsulate logic such as "consume an OpenAPI client and load resource X", or "subscribe to a WebSocket / Server-Sent Event channel for live updates on automation runs". This matches well with the dynamic nature of an automation app, where the UI often reflects the state of multiple asynchronous operations (test runs, webhook deliveries, polling status, etc.).

Finally, React's ubiquity in the industry makes it easier to onboard new developers and to find external resources, examples, and ready-made solutions when integrating new providers or redesigning parts of the interface. The long-term viability of the stack is strengthened by this broad adoption.

3.2. Conclusion

After considering Angular, Vue, and React, the final choice for the web frontend is React.

Angular was set aside because its heavy, highly opinionated structure and boilerplate would slow down iteration for what is fundamentally a lightweight, configuration-centric interface. Vue offers an approachable and elegant developer experience, but its ecosystem—while solid—is less central than React's for the specific needs of this project (OpenAPI-driven clients, advanced form handling, and a wide range of ready-made UI solutions).

React, on the other hand, combines a mature, widely adopted component model with a very rich ecosystem of libraries for forms, routing, data fetching, and OpenAPI integration. Its strong compatibility with TypeScript, and the ability to generate typed clients directly from the Go backend's OpenAPI specification, make it the most suitable choice. This allows the frontend to evolve quickly while preserving type safety and maintainability, which is essential for an automation platform that will continuously integrate new third-party services and automation features.

4. Mobile app language and framework

4.1. Explored solutions

4.1.1. Java + XML (Legacy Android View System)

For a long time, the standard for Android development involved writing business logic in Java and defining user interfaces via XML layouts. While this approach is well documented by over a decade of StackOverflow answers, it presents significant drawbacks for modern development standards.

The primary drawback encountered with this stack is the separation of concerns, which is often more cumbersome than helpful. Developers must constantly context-switch between Java code and XML markup. Simple tasks, such as binding data to views, require boilerplate code (referencing IDs, setting listeners), even with improvements like ViewBinding.

Also, creating rich, fluid user experiences, good animations and transitions is notoriously difficult in the legacy View system. It often requires complex "Animator" logic or defining additional XML resource files, making the UI code rigid and hard to maintain. For a project aiming for a modern look and feel, the steep learning curve of the XML attribute system and the verbosity of Java were deemed to inefficient compared to modern alternatives.

4.1.2. Swift (Native iOS)

Swift is the industry standard for native iOS development. It is a modern, type-safe, and fast language that replaced Objective-C.

On the positive side, Swift offers excellent performance and deep integration with the Apple ecosystem. It provides direct access to low-level hardware features and ensures a native look and feel that users expect on iOS devices. The language itself is expressive, with features like optionals and type inference that prevent common runtime errors.

However, the choice of Swift comes with a significant hardware constraint: it strictly requires a macOS environment for development (XCode) and an iOS phone for reliable testing. As the development environment for this project did not include an iOS phone, choosing Swift would have introduced unnecessary friction in the development workflow (requiring emulators or other hacks). Consequently, despite its technical merits, Swift was ruled out due to hardware unavailability.

4.1.3. Kotlin + Jetpack Compose

Kotlin is Google's preferred language for Android development, fully interoperable with Java but designed to address its shortcomings. Jetpack Compose is the modern toolkit for building native UI, designed to simplify and accelerate UI development on Android.

To illustrate the difference in complexity between Kotlin + Jetpack Compose and Java + XML, consider the task of creating a button that shows a message when clicked.

The Old Way (Java + XML) :

```
<Button
    android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click Me" />

Button button = findViewById(R.id.myButton);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        System.out.println("Hello World!");
    }
});
```

The Modern Way (Kotlin + Compose) :

```
Button(onClick = { println("Hello World!") }) {
    Text("Click Me")
}
```

Kotlin offers a significant improvement over Java regarding syntax and safety. Features like null safety are built into the type system, virtually eliminating 'NullPointerExceptions'. The language is concise, data classes and extension functions allow for writing expressive code with much less boilerplate than Java.

Jetpack Compose shifts the UI paradigm from imperative (manipulating XML views) to declarative. This means the UI is described in Kotlin code as a function of the current state. This approach has several distinct advantages:

- **Unified Language:** Both logic and UI are written in Kotlin, removing the XML context switch.
- **Ease of Animation:** Compose provides simple, high-level APIs for animations. creating complex motion layouts requires significantly less code than the View system.
- **Reusability:** UI components are just functions, making them easy to break down, reuse, and test in isolation.

The combination of Kotlin's modern language features and Compose's reactive UI model creates a highly productive environment, allowing for rapid iteration on UI designs without the legacy baggage of the Android View system.

4.2. Conclusion

After evaluating the options, Kotlin combined with Jetpack Compose was chosen as the definitive stack for the mobile application. The Java and XML approach was rejected due to its verbosity and the complexity involved in implementing modern animations. Swift, while powerful, was not viable due to the lack of necessary Apple hardware. Kotlin provides the safety and conciseness missing in Java, while Jetpack Compose offers a modern, declarative way to build interfaces that align perfectly with the project's need for a responsive and maintainable UI.

Here's a possible fleshed-out version of your database section, consistent with the rest of your document and your focus on security and automation-style workloads.

—

5. Database engine

The automation platform needs a database engine that can reliably store user accounts, OAuth2 credentials (or tokens), triggers, actions, and execution logs. The engine must support transactional consistency for configuration changes, flexible enough schemas to accommodate different automation types, and strong security and observability features. Both NoSQL and SQL engines were considered.

5.1. NoSQL engines

NoSQL databases typically offer flexible schemas and horizontal scalability, which can be attractive for rapidly evolving products and large-scale logging.

5.1.1. MongoDB

MongoDB is a widely adopted document store that excels at handling semi-structured data and rapid prototyping. Its flexible document model fits well with heterogeneous payloads, such as webhook bodies from different providers or loosely structured logs. It also has a mature ecosystem, good driver support for Go, and built-in replication and sharding features.

However, MongoDB's strengths are less aligned with some of the core needs of this project:

- Many central entities (users, providers, triggers, actions, automation workflows) benefit from strong relational modeling and transactional guarantees.
- Complex queries over relations (e.g., filtering executions by user, by connected provider, or by workflow) are generally more naturally expressed and optimized in a relational model.
- While MongoDB does support multi-document ACID transactions, its traditional usage patterns and indexing strategies can be more complex to tune for strongly relational workloads.

Given that our data model is predominantly relational and benefits from clear constraints (foreign keys, uniqueness, etc.), MongoDB was not selected as the primary datastore, although it could be considered in the future for auxiliary use cases (e.g., large-volume, unstructured event logs).

5.2. SQL engines

Relational databases remain the most suitable choice for the core of this application. They provide:

- Strong consistency and well-understood transaction semantics.
- Explicit constraints (foreign keys, unique indexes) for data integrity.
- Mature tooling for migrations, backups, observability, and performance tuning.
- Native support for complex queries, joins, and aggregations.

Within SQL engines, both PostgreSQL and MySQL were evaluated.

5.2.1. PostgreSQL

PostgreSQL is a modern, feature-rich relational database that prioritizes standards compliance and robustness. It offers:

- Advanced data types (JSONB, arrays, enums, ranges) that can model more flexible parts of the domain without fully giving up relational structure.
- Strong transactional guarantees, including full ACID compliance and robust concurrency control (MVCC).
- Mature support for security features (role-based access control, row-level security, fine-grained privileges).
- Good observability and tuning capabilities, with extensive metrics and query planning insight.

PostgreSQL integrates well with the Go ecosystem and GORM, and its JSONB capabilities are particularly useful for storing provider-specific metadata or unstructured configuration blobs without leaving the relational model. For an automation app that must mix structured entities (users, workflows, triggers) with semi-structured payloads (provider settings, webhook bodies), PostgreSQL provides a good balance between flexibility and rigor.

Given these properties, PostgreSQL is a strong fit for the primary database of the project.

5.2.2. MySQL

MySQL (and its derivatives such as MariaDB) is another widely used relational engine with strong performance, a large community, and good support in hosting providers. It handles typical CRUD workloads efficiently and has a mature ecosystem of tools and client libraries, including for Go and GORM.

However, for this particular project, MySQL is slightly less attractive than PostgreSQL for several reasons:

- Historically, PostgreSQL has offered more advanced SQL features and richer data types, which are beneficial for modeling complex automation and configuration data.
- JSON support in MySQL exists but is generally less flexible than PostgreSQL's JSONB implementation for indexing and querying.
- Some advanced features we may want to leverage in the future (e.g., more complex constraints or row-level security patterns) are more naturally implemented or better supported in PostgreSQL.

As a result, while MySQL would be a perfectly viable option and can support the application's core needs, PostgreSQL is preferred for its richer feature set and better alignment with a mixed structured/semi-structured data model.

6. Security considerations

6.1. ORM choice and database security

We chose GORM as our backend object-relational mapper because it integrates naturally with the Go ecosystem and is widely adopted and maintained. GORM abstracts most common CRUD operations while still allowing explicit control over the generated SQL, which simplifies development without hiding important database details. From a security standpoint, GORM encourages parameterized queries by default: values passed into query-building methods are properly escaped and bound as parameters, which significantly reduces the risk of SQL injection and related attacks.

6.2. Supply chain attacks

To mitigate supply chain risks on the backend, we rely on Go's native module system (go mod). The Go toolchain maintains a checksum database and verifies module content against recorded checksums, helping detect tampering of dependencies. In addition, we pin dependencies to specific versions or tags in go.mod, which prevents accidental upgrades to unreviewed releases and provides a reproducible build environment that is less exposed to malicious or compromised packages.

6.3. Sessions replay attacks

To limit the impact of potential session replay attacks, we rely on short-lived session tokens combined with secure transport and storage mechanisms. All authenticated traffic is served exclusively over HTTPS,

preventing attackers from trivially capturing tokens in transit. Session identifiers are stored in HTTP-only and Secure cookies, which reduces exposure to client-side scripts. Additionally, session invalidation on logout and periodic rotation of session tokens further narrow the time window during which a stolen token can be reused. These measures together aim to make replaying captured sessions significantly more difficult and less impactful.

6.4. DoS attack

To mitigate denial-of-service (DoS) risks, the backend will be fronted by rate-limiting and basic traffic-shaping mechanisms. Requests per IP or per account can be capped for critical endpoints, while timeouts and sensible limits on request size and concurrency help protect server resources from being exhausted by malicious or misconfigured clients. In a production environment, these application-level protections can be complemented by infrastructure-level safeguards (such as load balancers or WAF services) to absorb or block unusually high volumes of traffic.

For particularly sensitive routes such as user creation and authentication (/register, /login, password reset, etc.), we additionally require CAPTCHA validation. This adds a human-verification step that makes it harder to automate large-scale account creation or brute-force attempts, thereby reducing the feasibility of certain classes of DoS and credential-stuffing attacks targeting these endpoints.

7. Overall application conclusion

For the complete application stack, Go has been chosen for the backend, React for the web frontend, and Jetpack Compose with Kotlin for the mobile application. This combination provides a coherent, modern, and maintainable architecture.

On the backend, Go offers a simple deployment model (single static binaries), efficient concurrency for handling API calls and webhooks, and a focused ecosystem for building REST/OpenAPI services. This makes it well suited to an automation platform that must reliably orchestrate many external services with minimal operational overhead.

On the web frontend, React provides a mature component model, a rich ecosystem of UI libraries, and excellent support for working with REST and OpenAPI-generated clients. It allows rapid iteration on user interfaces and flows, which is essential for an application that must guide users through configuring complex automations and third-party integrations.

For mobile, Jetpack Compose with Kotlin offers a modern, declarative UI toolkit tightly integrated with the Android platform, along with strong tooling and language support. Kotlin's concise syntax and Compose's reactive UI model make it easier to keep the mobile client aligned with the backend API and the web frontend in terms of features and user experience.

Together, these three technologies strike a balance between developer productivity, performance, and long-term maintainability. Go provides a stable, efficient core for business logic and integrations; React enables a flexible, responsive web experience; and Jetpack Compose with Kotlin delivers a native-feeling, future-proof mobile application.