THE USA JOURNALS
THE AMERICAN JOURNAL OF ENGINEERING AND TECHNOLOGY (ISSN – 2689-0984)
VOLUME 06 ISSUE09

PUBLISHED DATE: - 14-09-2024
DOI: - https://doi.org/10.37547/tajet/Volume06Issue09-08
PAGE NO.: - 56-66

RESEARCH ARTICLE
Open Access

STUDY OF ARCHITECTURAL FEATURES AND PRACTICAL APPLICATION OF OBJECT-ORIENTED STATE-MANAGER REFLEXIO IN THE CONTEXT OF MODERN WEB DEVELOPMENT

Konstantin Astapov
Rambler&Co - Frontend Development Team Lead for Public Mail, Moscow
Russia

Abstract
This study examines Reflexio, an innovative state management solution for scalable web applications. The research aims to analyze Reflexio's architecture, key concepts, and practical applications, comparing it with existing solutions. The methodology involves a detailed analysis of Reflexio's core features, including its object-oriented approach, event-driven reactivity, and multi-stage event processing. The results demonstrate Reflexio's effectiveness in addressing common challenges in state management, such as excessive UI re-renders and high coupling between application domains. The comparative analysis reveals Reflexio's unique advantages in modularity, performance, and complex business logic handling. The study concludes that Reflexio represents a significant advancement in state management, offering powerful tools for creating maintainable and scalable web applications, particularly in complex corporate environments. This research contributes to the field by providing insights into a novel approach that combines object-oriented programming principles with reactive state management, potentially shaping future web development practices.

Keywords State management, web applications, object-oriented programming, event-driven architecture, scalability, modularity, reactivity, business logic, performance optimization, corporate software development.

INTRODUCTION

In the era of rapid web technology advancements, state management in scalable applications has become an increasingly complex task. Modern web applications are characterized by high interactivity, complex business logic, and the need to process large volumes of data in real-time [2]. In this context, traditional approaches to state management often prove insufficient, leading to a range of challenges in the development and maintenance of applications.

One of the key issues is the difficulty in separating business logic from the user interface, which complicates the scalability of applications and increases the risk of errors. Additionally, existing solutions frequently result in excessive user interface updates, which negatively impacts application performance.

Another significant problem is the high coupling between different domains of the application, which complicates the development and testing processes and hinders the implementation of new functionalities. These challenges are particularly acute in the context of large-scale enterprise applications, where high flexibility and scalability are required.

In response to these challenges, there is a growing need for new approaches to state management that consider the specifics of modern web applications and offer effective solutions to existing problems. It is in this context that Reflexio was developed—an innovative state manager built on the foundation of Redux but offering a range of unique concepts and solutions [3].

The purpose of this article is to provide a detailed examination of the architecture and key concepts of Reflexio, analyze its advantages compared to existing solutions, and discuss the potential areas of application for this tool in modern web development. The article will explore how Reflexio addresses the main issues of state management in scalable web applications and what new opportunities it opens up for developers.

The study will analyze the theoretical foundations and concepts of Reflexio, thoroughly examine its architecture and implementation features, conduct a comparative analysis with existing solutions, and consider the practical aspects of applying Reflexio in real-world projects.

Theoretical Foundations, Concept, and Architecture of Reflexio

Reflexio represents an innovative approach to state management in scalable web applications [1]. This business-flow-oriented state manager is built on the foundation of Redux but offers a range of unique solutions to the challenges faced by developers of large enterprise systems.

One of the features of Reflexio is its independence from the user interface infrastructure, making it a versatile tool for various frameworks and libraries. This is particularly important in large-scale enterprise client applications, where modular and multi-layered separation plays a critical role.

Reflexio is designed to address three primary challenges in modern web development:

1. Separation of the business layer from the UI layer in the application.

2. Minimization of redundant UI re-renders caused by unnecessary intermediate state updates during the execution of business scenarios.

3. Reduction of high coupling between different application domains, which complicates development and code readability as the application scales.

Conceptually, Reflexio rethinks the approach to code organization in reactive applications. In typical reactive applications with a state manager, the architecture is built around events (actions) and handlers that modify the state. However, in large applications, this approach becomes inefficient due to the complexity of managing interactions between different events and their handlers.

Reflexio addresses this issue by introducing the concept of "Bytes." A Byte in Reflexio is a higher-order entity that groups related events and their handlers. This allows for more efficient code organization and management of complex interactions within the application.

Consider an example: a button press might initiate an asynchronous process that concludes with a specific action. In Reflexio, these logically related actions are grouped into a single Byte, significantly simplifying the development and maintenance of the code. Another example is form actions that are only available after a window is opened. Grouping such dependent actions into a Byte allows for more effective management of application functionality.

It is important to note that Reflexio implements event grouping using object-oriented programming (OOP) principles. Handlers in Reflexio are described as classes responsible for handling a group of events. This enables the use of OOP benefits, such as encapsulation, inheritance, and polymorphism, when working with application state.

Moreover, a Byte in Reflexio is not just a group of events but a complex structure that allows for the declarative description of an events handler in the form of a class called script. Its design also includes reducers for manipulating applications' state and configurations that define the script's instantiation strategy and affect the scope of events. Each Byte is assigned a unique name that defines the namespace for the group of events. Within this namespace, each action also receives a unique name, creating a two-tiered action naming system.

Reflexio's architecture extends the concept of Redux by adding an additional layer of abstraction. In Redux, any event first passes through middleware, where it can be pre-processed before reaching the reducer, which modifies the state. Reflexio develops this idea further: Bytes group several related reducers and middleware into specialized units that handle business logic and manage application state. In addition to middleware and reducers, Reflexio introduces a special event post-processing stage, called After Effects, which is integrated into these units. The reducers are represented as objects with action names as keys, and the middleware is organized as an object—an instance of a script class.

Thus, Reflexio offers a powerful and flexible toolkit for state management in scalable web applications. Its architecture, based on the concept of Bytes and OOP principles, enables the creation of more structured, scalable, and easily maintainable applications, especially in the context of complex enterprise systems [1].

Implementation Features

Reflexio, as an innovative state manager, possesses a number of unique implementation features that distinguish it from other state management solutions for scalable web applications [1]. Let's explore the key aspects of its implementation in more detail.

1. Use of OOP Principles:

Reflexio applies object-oriented programming (OOP) principles, but with significant differences from traditional approaches. The main distinction is that classes in Reflexio do not serve as reactive states of the application. Instead, they are exclusively responsible for logic related to handling actions and managing internal state.

The internal state of Bytes in Reflexio is encapsulated and inaccessible from outside of its corresponding scripts. This ensures a high degree of isolation and prevents unwanted side effects. On the other hand, the application state (reactive state) is implemented as a separate object that can be read from and written to by any script class through a special context provided by the dependency injection (DI) mechanism.

This approach allows for a clear separation between business logic and state management, which is crucial for creating scalable and maintainable applications. Furthermore, the set of tools for working with the event flow is available in scripts as a single object, also provided via DI. Here DI also provides additional flexibility when extending and overriding the basic Reflexio API.

Another important distinction is that classes in Reflexio are embedded within a separate business layer. In the application’s code, instances of these classes are not implicitly created, nor is there direct access to their properties and methods. Instead, scripts intercept events, and state changes are made by dispatching events handled by reducers. This ensures a strictly unidirectional flow of work with scripts, which greatly simplifies debugging and understanding the business logic of the application.

2. Use of Dependency Injection:

In the Reflexio API, the use of the Dependency Injection (DI) pattern is a fundamental rule. This not only makes business units independent of the direct implementation of side effects and auxiliary functions but also addresses a broader issue: the separation of the business logic layer into two distinct internal levels—the contract (or configuration) level and the implementation level.

The implementation level consists of script classes, where each class corresponds to a specific Byte. The contract and configuration level includes the declaration of the Bytes themselves (with configurations as arguments and reducers), as well as the declaration of slices and their initial state. In Reflexio, a Slice represents a portion of the application state. Its definition includes the initial state and a set of Bytes related to a specific state domain. This separation makes the application architecture significantly more structured and, therefore, more readable and predictable for development planning.

3. Feature Flags Out of the Box:

Another advantage of using OOP to describe business logic in Bytes is the ability to store intermediate data not related to the store's state in the internal state of the class. This allows script instances to dynamically enable and disable features as needed, supporting a feature-flag architecture without the need for additional tools or libraries.

4. Event-Driven Reactivity:

In the Reflexio concept, state does not have to be immutable. Notifications to store subscribers and the execution of afterEffects in scripts always occur in response to specific actions (with specific names) or to the name of a Byte (the group of actions corresponding to that Byte). It doesn't matter whether the state changes as a result or how it changes.

This approach, called "event-type driven" as opposed to "state-change driven," leads to an interesting consequence. So-called computed states can use segments of actual state linked together by a chain of actions to sequentially react to changes in each other's states. This provides more transparent tracking and debugging of computed state changes.

5. Event Bus:

The key entities in Reflexio—Slice and Byte—are created and stored at the business level, separately from UI components. Interaction with these entities from components occurs exclusively through event dispatch via the Event Bus. Including certain Bytes and Slices in the general bus at the level of the configurable application root automatically expands the set of available actions (the business API of the application) for any part of the application code without needing to import new objects from other files.

6. Multi-Stage Event Processing:

Action (event) processing in Reflexio occurs in three sequential stages:

1) Before state changes (in watch and init)

2) During state changes (in reducers)

3) After state changes (in afterEffects)

This approach provides greater flexibility in managing state processing and allows events to be classified into two types: those that affect state (and therefore subscribers and afterEffects) and those that do not affect state (used exclusively as messages between modules within the business layer).

7. Extensibility:

Reflexio allows for the extension and overriding of the base API using the benefits of OOP—DI and inheritance. Tools necessary within a class for working with actions and state are not imported directly from the library but are implicitly injected into the class through its constructor.

The base API can be replaced (while retaining its signature) or extended by altering the Byte configuration without affecting the business logic code. This makes it easy to create various plugins for specific state management tasks, as well as specialized templates for building Bytes.

Additional plugins in Reflexio provide only Bytes—the fundamental objects of the concept, which are included in the general action bus. This means that plugins do not create new APIs or objects that require importing and storing but simply add their action sets to the trigger, allowing interaction with the new functionality from any point in the application.

Thus, the implementation features of Reflexio provide high flexibility, scalability, and maintainability for applications, offering developers a powerful toolkit for creating complex web applications with clear and understandable architecture.

Comparative Analysis with Existing Solutions

Currently, there are numerous solutions available, each with its own advantages and limitations. Reflexio offers a unique set of features that warrant consideration in comparison with other popular solutions.

Redux, as one of the most widely used state management tools, provides a predictable state container for JavaScript applications [4]. However, unlike Reflexio, Redux does not offer built-in mechanisms for grouping related actions and handlers. This can lead to a "bloated" store and increased complexity of the codebase as the application grows. Reflexio addresses this issue through the concept of "Bytes," which encapsulate related logic and state.

MobX, another popular tool, uses reactive programming for state management. While MobX provides a more "magical" approach to reactivity, which can simplify the code, it may make it difficult to track state changes in large applications. Reflexio, with its explicit definition of actions and handlers, offers a more structured and predictable data flow, which is particularly important in large-scale enterprise applications [5].

Recoil, developed by Facebook, offers an atomic approach to state management. While this solution is well-suited for managing local component states, it may be less efficient when dealing with complex global application logic. Reflexio, with its Byte concept and ability to group related actions, provides more powerful tools for managing complex business processes [6].

An important aspect of comparison is the support for asynchronous operations. While Redux requires additional middleware (such as redux-thunk or redux-saga) to handle asynchronous actions effectively, Reflexio provides built-in mechanisms for handling asynchronous data flows through its scripts and event processing system.

From a performance perspective, Reflexio offers a unique approach to optimizing re-renders. By allowing actions that do not trigger state updates (through setting null in reducers), Reflexio enables finer control over when and which parts of the user interface should update. This can lead to significant performance improvements in complex applications compared to more traditional approaches.

Reflexio's approach to modularity and extensibility is also noteworthy. While many existing solutions require additional libraries or complex patterns to achieve modularity, Reflexio provides this "out of the box" through its system of Bytes and plugins based on byte-templates. This is especially valuable in the context of micro-frontends and large enterprise applications, where modularity and code reuse are critically important.

However, it is important to note that Reflexio, being a more specialized solution, may have a steeper learning curve compared to simpler solutions. Its power and flexibility come with the need to understand a range of concepts, such as Bytes, scripts, and multi-stage event processing. This may require additional time for developers to master, especially those accustomed to more traditional state management approaches.

Additionally, the Reflexio ecosystem, being newer compared to ecosystems like Redux or MobX, may be less developed in terms of available third-party tools, libraries, and learning resources. However, this drawback is offset by Reflexio's built-in extensibility and its ability to adapt to various use cases without the need for external dependencies.

In the context of typing, Reflexio offers deeper integration with type systems, particularly TypeScript, compared to some other solutions. This provides better support for static code analysis and can significantly reduce runtime errors.

Thus, the comparative analysis shows that Reflexio offers a unique set of capabilities, particularly well-suited for complex, scalable web applications. Its advantages in modularity, performance, and management of complex business processes make it an attractive choice for large enterprise projects. However, like any tool, Reflexio requires careful consideration of the specific project requirements and the team's readiness to adopt a new approach to state management.

Practical Application

Reflexio, as a powerful and flexible state manager, finds broad application in the development of scalable web applications. Let's take a closer look at the practical aspects of using Reflexio through the example of implementing modal window functionality [1].

Creating a Byte:

A key element of Reflexio's practical application is the creation of Bytes. Consider the example of creating a Byte for managing a modal window:

```
const floatWindowBite = Bite(
  {
    init: null,
    setState: openWindowReducer,
    openWindow: null,
    closeWindow: null,
    submit: null,
    done: null,
  },
  {
    watchScope: ['floatWindow'],
    script: FloatWindowScript,
    instance: 'stable',
    initOn: 'init',
  }
);
```

In this example, the `floatWindowBite` Byte is created, responsible for the functionality of the modal window. The Byte includes a set of actions (`init`, `setState`, `openWindow`, `closeWindow`, `submit`, `done`) and a script configuration.

Reducers and Actions: Notice the `case-reducers` object in the first parameter of the `Bite` function. For some keys (`init`, `openWindow`, `closeWindow`, `submit`, `done`), the value is set to `null`. This is an important feature of the Reflexio API design: such actions do not trigger state updates and do not notify store subscribers.

This design choice has several important consequences:

- Prevention of Excessive UI Re-renders: Actions with `null` reducers do not trigger UI updates, which enhances application performance.

- Safe Description of Business Logic: Developers can freely describe complex business logic in terms of actions without worrying about unnecessary side effects.

- Improved Reactive Connections: This approach facilitates building reactive connections between different Bytes, which is critical for creating complex, interconnected business processes.

Script Configuration: The second parameter of the `Bite` function contains the script configuration. Let's examine its key elements:

- `watchScope: ['floatWindow']`: Defines the events this Byte will "listen" to. In this case, the Byte responds to all events related to `floatWindow`.

- `script: FloatWindowScript`: Specifies the script class that will handle the Byte's logic.

- `instance: 'stable'`: Determines the instance creation strategy for the script. `'stable'` means that a single, persistent instance will be created.

- `initOn: 'init'`: Indicates the action that triggers the creation of the script instance.

In the `case-reducers` object, some keys have a value of `null`. According to the design of the library's API, this indicates that actions with these statuses do not trigger state updates or notify store subscribers. This approach prevents unnecessary UI rendering, allowing for the safe description of business logic in terms of actions without side effects. It also simplifies the creation of reactive connections between units of business logic (Bytes).

This method ensures the efficiency and responsiveness of the user interface even as the application's complexity increases. Using actions to manage state transitions without triggering excessive updates allows for clear and predictable flows between different parts of the application.

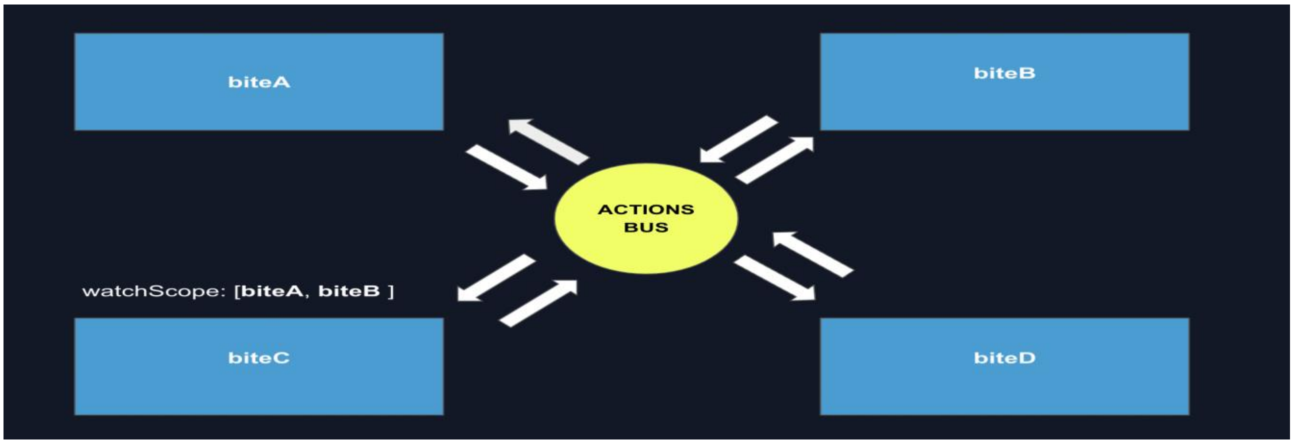


Figure 1 - Reactive connection between bites

Figure 1 illustrates a schema where different Bytes interact reactively with each other, exchanging actions through the bus. If reducers corresponding to those actions are null it is possible to build complex but readable reactive chains of inter-bites communications ensuring absence of unexpected store updates. The configuration parameter `watchScope` specifies the range of events that the Byte listens to. In this example, the scope is set to `[biteA, biteB]`, meaning it listens to all events from `biteA` and `biteB`.

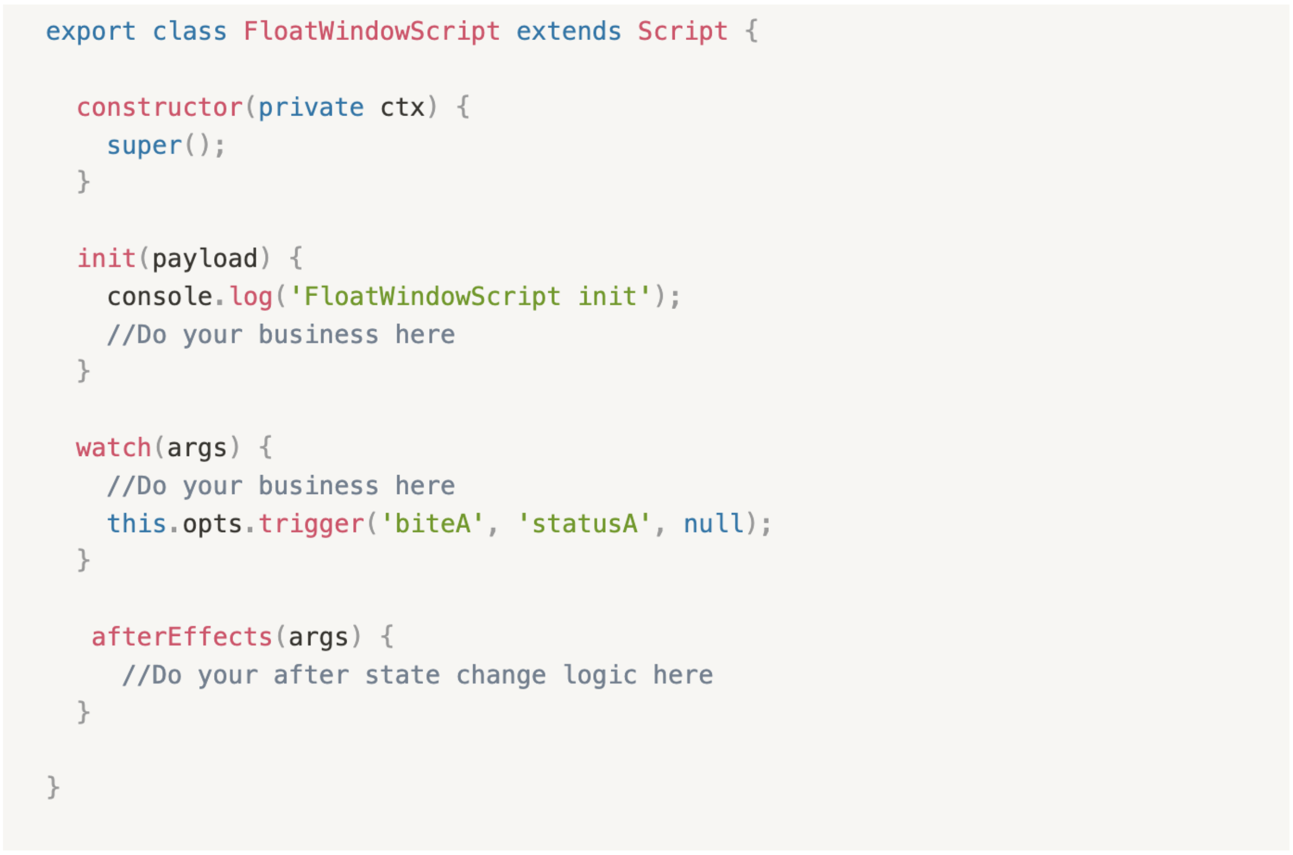
The script configuration parameters include:

- `initOn`: A string parameter that defines the action that initiates the creation of the script class instance during Byte initialization.

- `instance`: Defines the strategy for creating script class instances for the `initOn` action.

- `watchScope`: An array of strings or objects indicating the names of Bytes or specific events that the Byte will "listen" to.

The script structure is as follows:



```
export class FloatWindowScript extends Script {

  constructor(private ctx) {
    super();
  }

  init(payload) {
    console.log('FloatWindowScript init');
    //Do your business here
  }

  watch(args) {
    //Do your business here
    this.opts.trigger('biteA', 'statusA', null);
  }

   afterEffects(args) {
     //Do your after state change logic here
  }

}
```

The script contains important functions such as `init` and `watch`. The `init` function handles the initiating event, while `watch` processes all events related to the Byte. Various tools are available within the script for working with events, allowing you to intercept events, trigger other events, change state, read state, create promises that wait for events, and stop event propagation.

A key concept in Reflexio is organizing business logic so that most action processing occurs within the script, where the full benefits of OOP principles can be utilized. Meanwhile, reducers should remain as simple as possible, functioning primarily as basic setters. This division ensures the power and maintainability of business logic while effectively managing state changes through reducers.

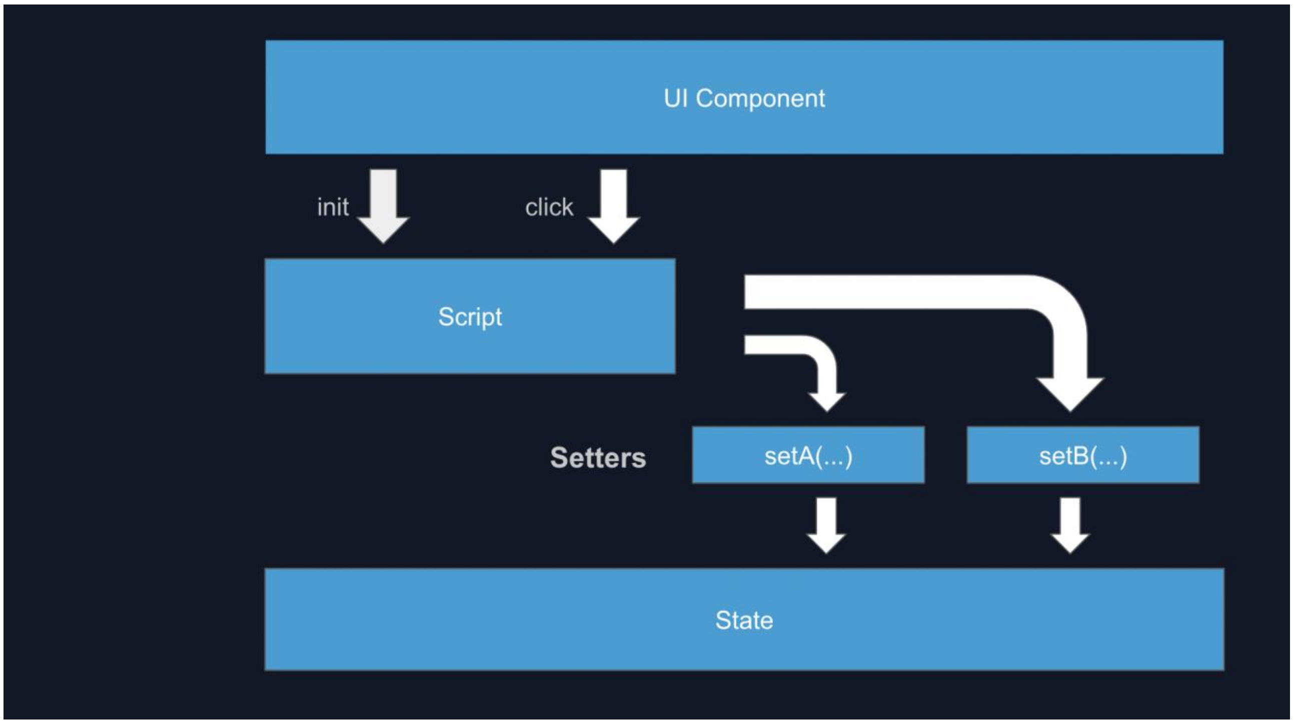


Figure 2 - Suggested schema of event handling with script and reducers

Figure 2 demonstrates the advantages of using OOP for describing business logic in Bytes:

- Managing Internal State: The internal state of the class can store intermediate data not directly related to the store's state.

- Dynamic Feature Management: Script instances can dynamically enable or disable features, supporting a feature-flag architecture.

Reflexio introduces an additional stage of action processing—`afterEffects`. This function is triggered after the state has been changed by the reducer and provides information about the event that caused the state change.

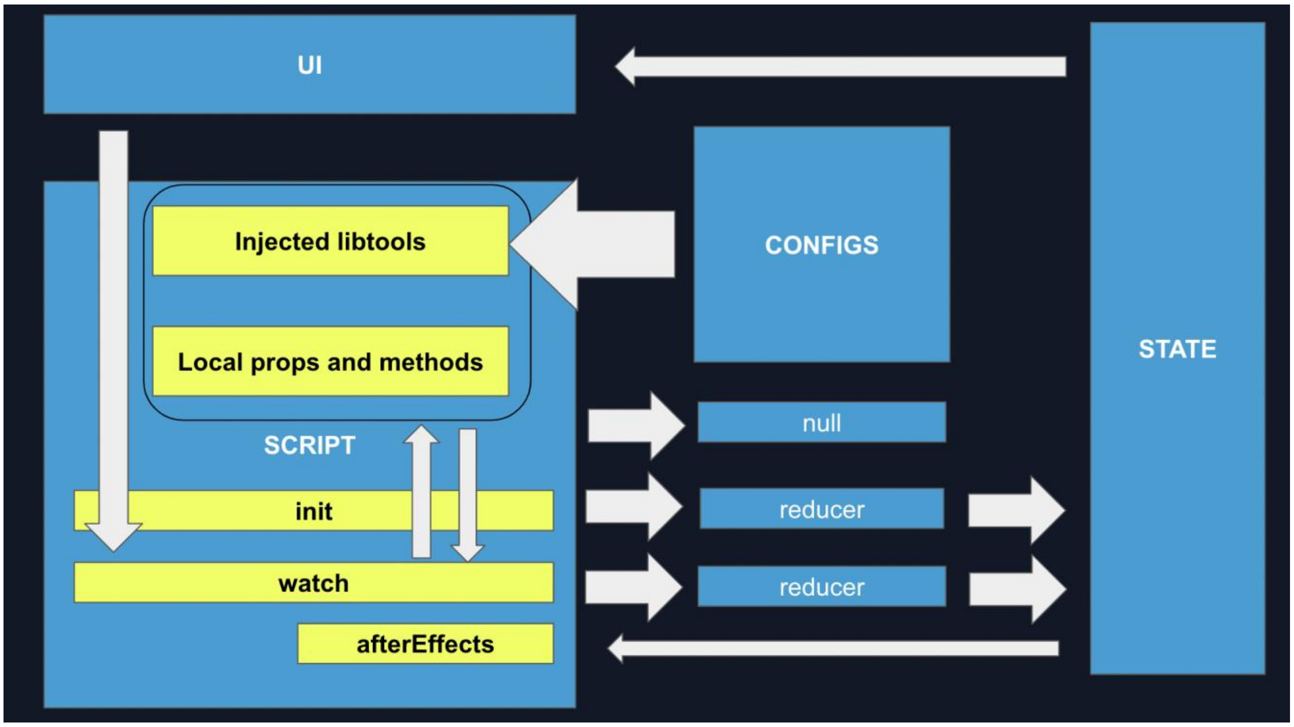


Figure 3 - Business flow in the Reflexio concept

Figure 3 illustrates the business flow in the Reflexio concept. Actions (events) are initiated by the user from the UI, then processed by the script, passing through `watch` and `init`, before modifying the state through reducers. Within the script, where all the business logic for the corresponding Byte is described, processing occurs using the internal embedded tools defined by the `Configs` object.

This flow ensures an efficient organization of business logic with clear stages of event processing, state changes, and subsequent processing, managed within a robust and modular structure.

The `addOpts` parameter within the Byte allows custom objects or functions to be passed into the class, making them accessible within the context of Byte instances. This is an example of custom use of dependency injection (DI). In `addOpts`, functions for calling external APIs can be specified, configured according to the environment, or mocked for testing and debugging. This method of functionality injection allows for the creation of factories and generic templates for Bytes, enhancing flexibility and code reusability in Reflexio.

CONCLUSION

Reflexio represents an innovative approach to state management in scalable web applications, offering solutions to several critical challenges faced by developers of large enterprise systems. Built on the principles of object-oriented programming and a reactive approach, Reflexio ensures an effective separation of business logic and the user interface, which is a fundamental requirement for creating maintainable and extensible applications.

Key concepts of Reflexio, such as Bytes and multi-stage event processing, provide developers with powerful tools for organizing complex business logic. The use of OOP principles in the context of state management enables the creation of more structured and comprehensible architectures, especially in large-scale projects.

The practical application of Reflexio demonstrates its ability to effectively address issues like excessive re-renders, high coupling between different application domains, and scaling complexity. The ability to dynamically manage functionality through the feature-flag mechanism and a flexible extension system via plugins makes Reflexio particularly appealing for long-term projects with evolving requirements.

Comparative analysis with existing solutions shows that Reflexio offers a unique set of features, particularly valuable in the context of complex enterprise applications. Its approach to modularity, performance, and asynchronous operation management sets it apart from more traditional state management tools.

However, like any advanced tool, Reflexio requires a certain investment in learning and adapting existing development practices. Its power and flexibility come with the need for a deep understanding of its concepts and architectural principles.

REFERENCES

1. Reflexio //Github. URL: https://github.com/rambler-digital-solutions/reflexio
2. Harlampidi V.K. Progressive Web Applications: A Review of Modern Methods, Tools, and Practices // Science Bulletin. - 2023. - Vol. 4. - No. 7 (64). - P. 401-421.
3. Johnson J. Designing with the mind in mind: simple guide to understanding user interface design guidelines. – Morgan Kaufmann, 2020.
4. Redux. URL: https://redux.js.org
5. MobX. URL: https://mobx.js.org/README.html
6. Recoil. URL: https://recoiljs.org