Home Assistant Conceptual and Concrete Architecture

Ali Sina	218318428	alisina@my.yorku.ca
Arash Saffari	218791632	arashrt@my.yorku.ca
David Luu	216157463	luu332@my.yorku.ca
John Prabahar	219087279	don19@my.yorku.ca
Omer Omer	218636878	omeromer@my.yorku.ca
Siyan Sriganeshan	218707190	siyan@my.yorku.ca

24 November 2024

Abstract

This report provides a detailed analysis of the architecture of Home Assistant, an open-source platform for home automation. The report focuses on the core architectural patterns, subsystems, interactions between components, and key design principles that shape the platform's functionality and performance. At the heart of Home Assistant's design is the combination of Implicit Invocation and Microservices architecture. Implicit Invocation is implemented through an event-driven model, where components interact indirectly via an event bus, allowing for decoupled, scalable communication across the system. The event bus handles asynchronous communication between components, which includes devices, services, and automations, facilitating modularity and flexibility.

The Microservices architecture is demonstrated through the system's loosely coupled, self-contained services that communicate through lightweight, well-defined interfaces. These services are independent, allowing for easy scaling and maintenance. A key feature of Home Assistant is the ability to integrate third-party devices and services, which are managed through subsystems like Auth for user authentication, Backports for backward compatibility, and State Machine for managing the states of entities.

Additionally, design patterns such as Observer and Abstract Factory are used to manage state changes and provide flexible integration of new devices. The State Machine and Service Registry work together to maintain synchronization and execute the appropriate actions in response to events. The Timer subsystem ensures time-based automation, while Generated and Scripts handle runtime configurations and command-line operations.

The report highlights the key challenges faced during the development of Home Assistant, particularly the coordination issues in the open-source community, as well as the complexities of asynchronous programming and integration with diverse third-party devices. The comparison between conceptual and concrete architectures reveals differences in the implementation of the event-driven system, reflecting how practical constraints influence architectural choices.

The report concludes by emphasizing the importance of understanding both the conceptual and concrete architectural styles in software development. Home Assistant's modular, event-driven architecture provides a robust foundation for a scalable and maintainable home automation platform. The lessons learned from this analysis offer valuable insights into the design and evolution of complex software systems and their impact on both the development process and user experience.

Introduction and Overview

This report presents an in-depth analysis of the architecture of Home Assistant, an open-source home automation platform that integrates various smart devices and services to create a unified control system. The purpose of this report is to document and evaluate the architectural design of Home Assistant, including its core components, design patterns, and how these elements work together to deliver an efficient, scalable, and modular system. It also explores the transition from the conceptual design to the concrete implementation, highlighting key differences and lessons learned during the development process.

Home Assistant employs a blend of architectural styles, notably Implicit Invocation and Microservices, to ensure flexibility, scalability, and maintainability. The primary architectural pattern employed in Home Assistant is Implicit Invocation, which is facilitated through an event-driven model using an event bus. This design allows components within the system to communicate asynchronously, promoting loose coupling and enabling the platform to integrate a wide variety of third-party devices and services without being tightly dependent on their specific implementations. Microservices, on the other hand, ensure that various system functions

are modular and can evolve independently, contributing to the platform's extensibility and robustness.

The report is organized into several sections, each focusing on different aspects of Home Assistant's architecture. The first section introduces the primary design patterns and subsystems that make up Home Assistant's core architecture, explaining the roles and interactions of each. The second section provides a detailed overview of how these components interact within the system, including an analysis of key subsystems such as the State Machine, Timer, Service Registry, and the event bus. Following that, the report examines the use of design patterns, such as the Observer pattern and Abstract Factory pattern, and their relevance to the architecture. Additionally, the report compares the conceptual architecture to the concrete implementation, noting discrepancies that arose during development and modifications that were necessary as the system matured. Finally, the report discusses the concurrency model in use within Home Assistant, highlighting the challenges and advantages of asynchronous programming within a highly interactive and distributed system.

The Home Assistant architecture is built on a robust combination of event-driven and microservice based design patterns. The event bus serves as the central communication hub, allowing components to remain decoupled and facilitating the system's scalability. This architecture enables the easy integration of third-party devices and services, as components only need to know how to send and receive events, not how other components are implemented. Additionally, the system's modularity, supported by microservices principles, ensures that individual components or services can be updated or replaced without disrupting the rest of the system.

Through the comparison of the conceptual and concrete architectures, the report highlights how the design has evolved over time. While the conceptual architecture served as the blueprint, real-world implementation required adjustments and modifications due to practical constraints, such as performance considerations and the need for backwards compatibility. The concrete architecture reflects these changes and provides a clearer picture of how the system operates in practice.

The use of asynchronous programming, facilitated by Python's asyncio, is central to the platform's performance, allowing Home Assistant to handle multiple tasks concurrently without blocking. However, this also introduces challenges, particularly when dealing with synchronous third-party libraries and debugging asynchronous code.

In terms of development team dynamics, the report identifies several challenges, including coordination issues arising from Home Assistant's open-source nature and the diverse skill levels of contributors. These challenges are compounded by the complexity of maintaining a large number of integrations and the growing expectations of the community. Addressing these issues will require better tools for communication, testing, and collaboration, as well as a focus on reducing technical debt.

Overall, this report demonstrates how architectural styles and design patterns shape the functionality, scalability, and maintainability of Home Assistant. It underscores the importance of aligning conceptual design with concrete implementation and highlights the lessons learned from the development process, which can inform future software engineering projects.

Conceptual Architecture

The primary architectural style documented by the developers of Home Assistant is Implicit Invocation. More specifically, it uses event buses to communicate with its connected devices/services. Events are triggered through four main ways, either through; State Machine, Timer, Service Registry, directly [1]. First the components can set a state which could trigger an event. Second, Timer which can either trigger events episodically or one-off. Third, the services are for the complex events that are triggered, like turning off lights at night. Lastly, direct events are for simple events like the user turning off the light on the app. Figure 1 illustrates the paths in which these events can be triggered.

The second most prevalent architecture could be argued to be Microservices, as Home Assistant is essentially the middleman which connects all of the registered devices. These devices are all third party and function independently. As mentioned previously, they are accessed either directly or through complex service calls. Although it could be argued that a Service Oriented Archecture is in use, there are no such subsystems found that interact with interfaces, all components of Core communicate directly with each other. The software itself is straightforward, the logic is heavily based on managing the interactions between the components. Overall implementation of Home Assistant is using layering where it is abstracted in the order of Core, Supervisor, Operating System [2].

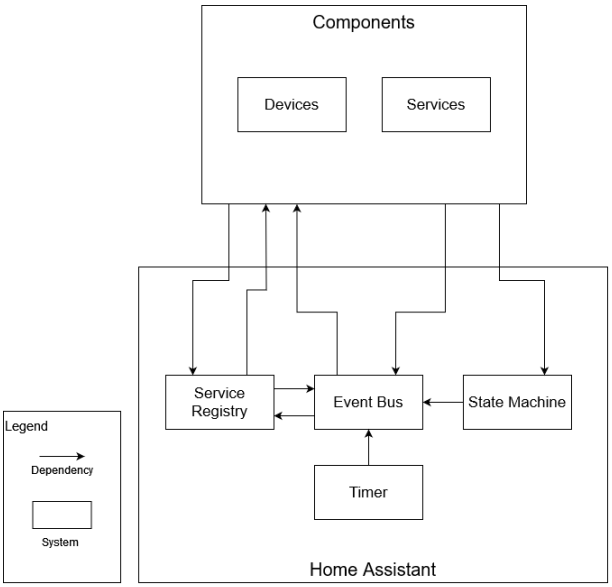

Figure 1: Flow of Event Triggers

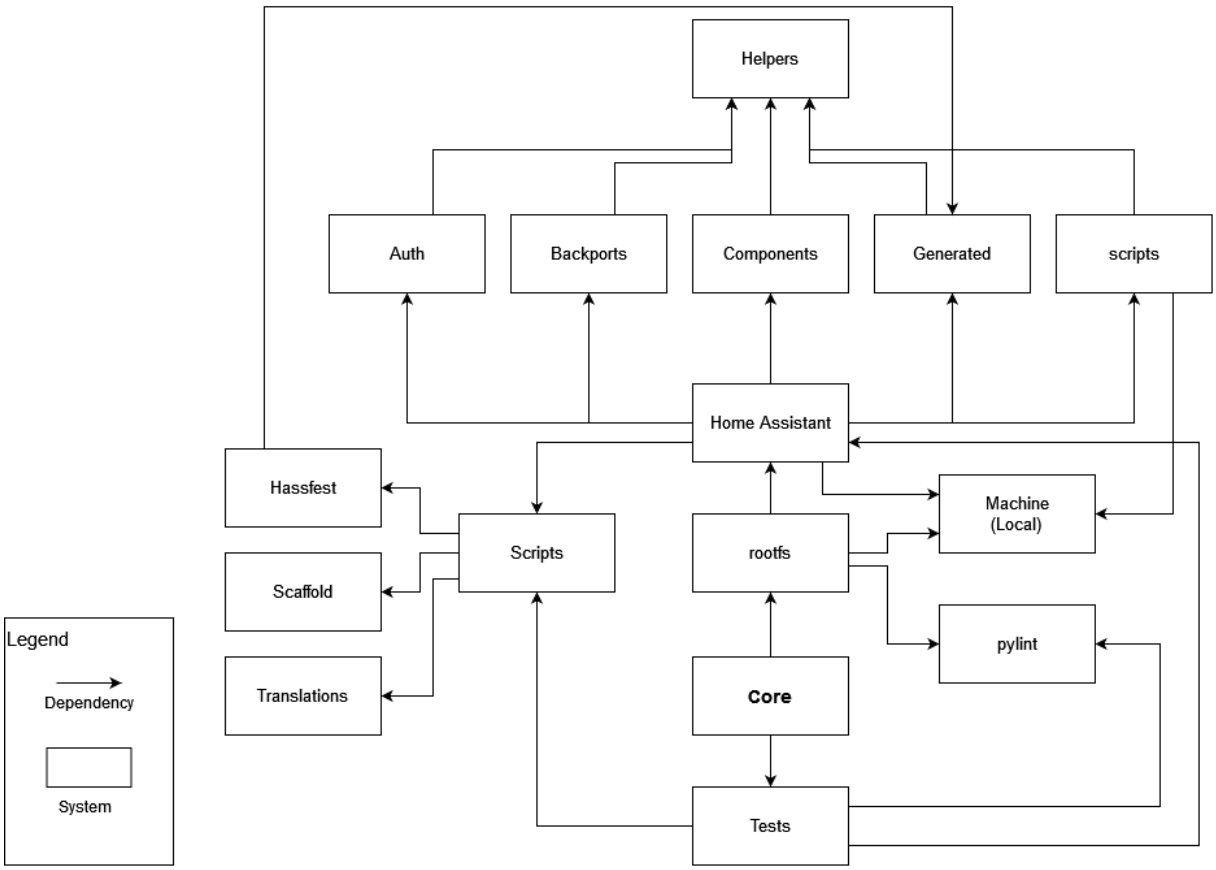

Figure 2: Subcomponents found in Home Assistant's core [10]

Subsystems And Interactions

Home Assistant: Core component that takes care of all the subsystems that concurrently work. Works by coordinating all the events raised by the subsystems and keeping them up to date and in sync with each other.

Components: Incorporates external devices and services, such as smart lights, sensors, or cloud APIs. Achieves communication with external devices/services using platform-specific APIs or protocols. Which then it exchanges information with the Home Assistant core by reporting their states or receiving commands.

Generated: Stores configuration files generated at runtime, reflecting the current state of the system or user-defined settings. Interaction occurs when these files are read by the core and components during initialization, and they are updated dynamically in response to system changes

Scripts: Provide command-line tools to interact with and manage Home Assistant from the terminal. Used to debug, setup and various other tasks.

Helpers: General sub-component used by other components reducing code reusing ensuring constant behavior throughout the system.

Hassfest: Manifest validator, ensuring everything is properly configured such as the states of the components. Helps catch errors early, ideally before deployment.

Scaffold: Blueprints and interfaces for creating new components, ensures coding practices. Used by developers to keep uniformity in code.

Translations: Multi language support for Home Assistant, primarily used by the front end to display User's preferred language.

Machine: Installation and configuration based on the device Home Assistant is installed in. Gives the ability to configure and install according to local machine.

Pylint: Check for syntax errors, and adhere to code guidelines set by developers, ensures the developers code is in order before submission to main branch.

Test: Testing components for stability and correctness by testing various parts of Home Assistant.

Design Patterns

In terms of design patterns used in Home Assistant, the Observer pattern plays a crucial role in managing the states of entities, such as devices. This pattern ensures that whenever a component undergoes a change in state, it must notify Home Assistant by calling specific methods, such as schedule_update_ha_state() or async_schedule_update_ha_state(). These methods trigger Home Assistant to update its internal log to reflect the changes made by the specific component. By doing so, the Observer pattern enhances the overall functionality and reliability of the Event Bus architecture, ensuring that the system can track and respond to state changes in real time. This creates a more efficient and synchronized system where components remain up to date with the current state of all connected devices.

In a similar vein, the Microservices architecture within Home Assistant is further supported by the Abstract Factory pattern. This pattern facilitates the seamless integration and configuration of new devices or components into the system by treating them as entities. When a new component is initialized, it is required to belong to a specific abstract class, such as an Ikea smart light, which would fall under the broader "light" abstract class. The use of this abstract class allows for the easy handling of various components without the need to depend on the specific implementation details of each device. This approach ensures that clients or services interacting with the components are not tightly coupled to concrete implementations but instead rely on the more flexible abstract class. As a result, it enhances modularity and scalability, enabling Home Assistant to integrate a diverse range of devices and services while keeping the system loosely coupled and adaptable to future changes.

Concrete Architecture

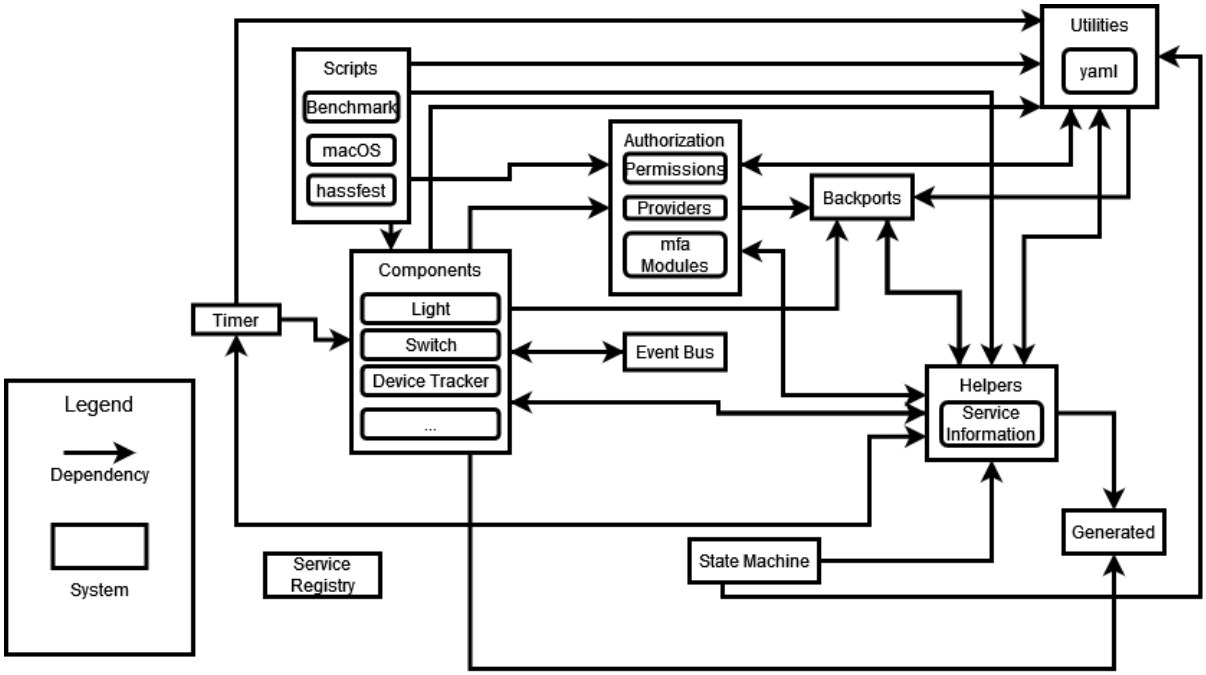

Figure 3: Final Concrete Architecture derived from Understand

The concrete architecture incorporates both event-driven implicit invocation and microservices principles. Through observing Figure 3, Implicit Invocation is evident in the structure of the entities, particularly the component entity, which houses the key features of Home Assistant and is directly linked to the event bus, acting as the event generator. This setup aligns with implicit invocation, as components do not directly call methods in other components. Instead, they generate events—such as user actions, system state changes, or messages—that other components can listen for and respond to. The event bus serves as an intermediary, facilitating the delivery of events from components to users. The components are also loosely coupled, meaning they can be added, removed, or replaced without impacting others, which showcases the flexibility of implicit invocation. Moreover, the scalability of the architecture is improved by the fact that components are not directly connected to all other components, and the event distribution system can be scaled independently. These features are a clear demonstration of the principles of implicit invocation.

This architecture is also in line with microservices principles, where services are self-contained, focused on specific functions, and can be developed, deployed, and scaled independently. The various systems in the model reflect this, as each microservice can be updated and deployed on its own without requiring a full system redeployment. Additionally, each subsystem is responsible for a distinct function or domain, as outlined in the previous section—for instance, the auth subsystem handles user authentication to secure Home Assistant, while backports ensures compatibility with older Python code. Furthermore, an analysis of the code reveals that subsystems communicate frequently, often through simple method calls, with loose coupling between them. This demonstrates that services do not need to be aware of each other's implementation details, a core characteristic of microservices. Overall, the event-driven structure strongly supports both event-driven and microservices architecture principles.

Subsystems And Interations

In addition to sub systems explained in conceptual part above, in concrete architecture, the following subsystems were identified:

- State Machine: The Home Assistant system's state machine is in charge of managing the states of the entities and any upcoming transitions between them. For instance, a light could have ”on” and ”off” states, and the State Machine component would control how they changed. [3]
- Utilities: The user-based system configurations that are provided through YAML files are referred to as utilities subsystems. With utility configuration features, for example, a user might schedule lights to turn on at sunset and off at morning. Tools for parsing, validation, and dynamic configuration updates are included in this module. [7]
- Backports: Backports ensure reliability of the system through providing backward compatibility with previous Python versions. To prevent system crashes, this module, for example, would translate code written using new Python features while the user is using an older version. [8]
- Timer: Timer manages all scheduling and timing duties that are essential for automation and time-based processes. Every second, it uses the event bus to transmit a time_changed event message to plan automation and even associated tasks. [9]
- Generated: Generated includes any dynamically generated files that are necessary for system and user session operations that are created during runtime. [7]

The interaction between subsystems begins with event generation, where components initiate events by producing data or signaling a change in state. These events serve as triggers for the system's response. Components are the core units of the system, representing entities or services that either generate or react to data. They trigger events through the creation of new data or signaling state changes, which enables asynchronous communication within the system. This approach allows components to notify others of occurrences without being directly connected, promoting independence. Each component operates autonomously, generating events as needed, without needing to know which other components will process them.

The event bus functions as a centralized hub that manages, receives, and directs events to the appropriate listeners or services. This centralization simplifies communication by relieving individual components from handling complex routing, while also managing tasks like event prioritization, retries, and ensuring event delivery.

The State Machine monitors and updates the states of entities based on incoming events. Each event may trigger a state change, with the State Machine ensuring the system evolves in a predictable and consistent manner. It enforces rules to prevent invalid or inconsistent state transitions, keeping the system synchronized with the events it processes.

The Service Registry links events to the services or actions that should be executed in response. When an event occurs, the Event Bus directs it to the appropriate service, which then processes the event by executing specific logic. This structure supports dynamic service discovery, modularity, scalability, and flexibility, as services can be added or modified without disrupting the overall system. It also ensures that the right actions are taken in response to particular events.

Finally, after an event is processed, the system communicates the outcome to the originating component or any relevant entity. This feedback, which could include status updates, results,

or triggers for further actions, ensures that components are aware of the outcomes of the events they initiated and can adjust accordingly. The feedback loop is essential for maintaining system coherence, allowing components to stay synchronized with the current state and adapt their behavior based on prior results, enhancing the system's responsiveness and flexibility.

Concurrency

The Home Assistant Core architecture is built around a highly concurrent model that utilizes Python's asyncio framework, with an event-driven design at its foundation. This architecture is structured to ensure efficient task management, where an event loop plays a central role in scheduling tasks such as device updates, API calls, and the execution of automation processes. By employing this event-driven approach, the system is able to perform non-blocking operations, which significantly enhances responsiveness and allows for seamless multitasking within the system.

A key feature of this design is the central event bus, which facilitates smooth communication between various components of the system. This bus enables the concurrent processing of multiple integrations and automations, ensuring that different system components can function independently while still maintaining overall system coherence. To further optimize performance, tasks that are resource-intensive or could potentially block the main event loop, such as CPU-heavy operations, are offloaded to thread pools or background threads. This ensures that these tasks do not interfere with the event loop's ability to handle other operations efficiently.

Additionally, process isolation is achieved through the use of Docker containers, which not only improve scalability but also contribute to the stability of the system by isolating different components and services. This containerization approach allows for better resource management and supports the deployment of Home Assistant in more complex environments, such as those with large device networks.

Despite these advantages, the architecture also presents certain challenges. For example, debugging asynchronous code can be difficult due to its non-linear nature, and managing third-party libraries that are synchronous may require additional workarounds, such as implementing threading solutions. Nonetheless, this event-driven and highly concurrent architecture supports the efficient handling of complex configurations and large-scale device networks with minimal performance overhead.

Team Issues

The Home Assistant Core architecture faces several challenges primarily stemming from its open-source nature and the global collaboration model it operates under. Since the project relies on contributions from a distributed team spread across various time zones, coordination becomes a significant hurdle. Contributors often face difficulties aligning on priorities, which can lead to disagreements and delays, particularly when it comes to reviewing and merging pull requests. This decentralized workflow can result in bottlenecks, where certain tasks or contributions may remain pending due to the asynchronous nature of collaboration.

In addition, the project experiences a wide variation in skill levels among contributors, which contributes to inconsistencies in code quality and the accumulation of technical debt. New contributors, in particular, face challenges with the complexities of asynchronous programming, which can make it difficult for them to contribute effectively without a steep learning curve. This issue is exacerbated by the maintenance of hundreds of integrations, each requiring constant updates and bug fixes, all while ensuring backward compatibility with older versions.

These ongoing maintenance tasks stretch the project's resources thin, particularly when considering the diversity of devices supported by the system. Testing becomes an especially complex endeavor as the system must be validated across a vast array of hardware and configurations, further straining the team's capacity.

Furthermore, high user expectations place additional pressure on the project, as the community expects frequent updates, new features, and seamless performance. This pressure is compounded by the risk of burnout among the maintainers, many of whom work on the project voluntarily, relying on limited community funding to support their efforts. Given these challenges, sustaining the growth and maintaining the quality of the project requires improvements in several key areas. Enhanced communication tools and practices are necessary to facilitate coordination across the global team. Standardized contribution guidelines would help streamline the process and ensure consistency in code quality. Additionally, automating testing processes can reduce the burden on contributors and improve the reliability of integrations. Active engagement with the community is also essential, ensuring that contributors feel supported and motivated, which will help alleviate the pressures of burnout and ensure the long-term success of the Home Assistant project.

Derivation Process

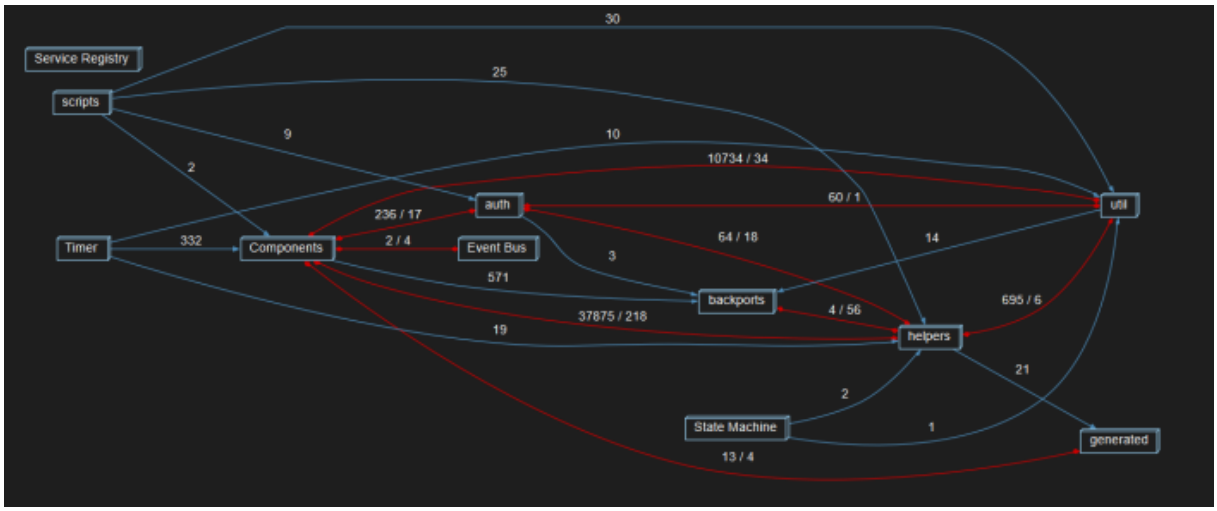

Figure 4: Concrete Architecture mock up in Understand

The concrete architecture was developed using Understand software, which generates flowcharts to illustrate relationships and creates a dictionary of variables and procedures from the source code. The architecture graph generated by Understand is presented in Figure 4.

The process began by identifying key architectural elements based on software requirements and the previously established conceptual architecture. Given that event-driven implicit invocation was the central architectural style in both the conceptual, it was crucial to include the event bus as a key component. This also extended to all components that interact with the event bus and rely on its functionality. These elements were organized into categories, including the timer, components, service registry, and state machine, as they are essential to Home Assistant's functionality and play key roles in interacting with the event bus. These components facilitate event handling and communication across the system, with the state machine maintaining the current status of entities and triggering state change notifications. The service registry listens

for events on the event bus to execute service actions, while the timer broadcasts time updates every second. Additionally, the auth, util, helpers, backports, and scripts classes were selected due to their significant interactions with these core components. The authentication system ensures proper user access, while the util module handles configuration and dynamic updates, and helpers aid in code reuse. The generated category covers configuration files created during execution, and scripts include command-line tools for managing Home Assistant tasks.

After identifying these key components, the entities and code were organized into a separate concrete folder. Some components, such as the timer, were found embedded within other entities (e.g., components), even when using Understand's search tool. Consequently, the timer was extracted into its own folder, while other elements like the service registry, event bus, and state machine were located as standalone classes in the core.py file. This led to an updated core architecture diagram, which revealed different connections compared to the conceptual model, as seen in both Figure 3 and 4. For example, the timer was connected to components instead of the event bus, the state machine interacted with helpers and util, and the service registry had no calls in other components or classes. These findings highlight the discrepancies between conceptual and actual software architecture during implementation.

Conceptual to Concrete

The Conceptual and Concrete architectures of Home Assistant both utilize Microservice and Implicit Invocation design patterns, but they differ in how these patterns are applied, as illustrated in Figures 2 and 3. In the Conceptual architecture, the Event Bus plays a central role, managing the states of all Home Assistant functions and coordinating with the State Machine, Service Registry, and Timer to facilitate its operation. In contrast, the Event Bus in the Concrete architecture is less prominent, primarily connecting to the Components subsystem. One key difference between the two architectures is the absence of broader dependencies around the Event Bus in the Concrete architecture, highlighting its more focused role. This divergence stems from the Conceptual architecture serving as a high-level blueprint, outlining the intended structure of the system, while the Concrete architecture reflects practical adjustments and modifications made during the development process.

Additionally, the Conceptual architecture incorporates elements of the Pipe-Filter design style, as seen in how components are connected. However, the Concrete architecture does not exhibit this feature, with some components instead forming two-way dependencies. This further illustrates the difference between the conceptual representation, which emphasizes the overall structure, and the Concrete architecture, which reflects the actual implementation and interaction of subsystems within Home Assistant.

Usecase

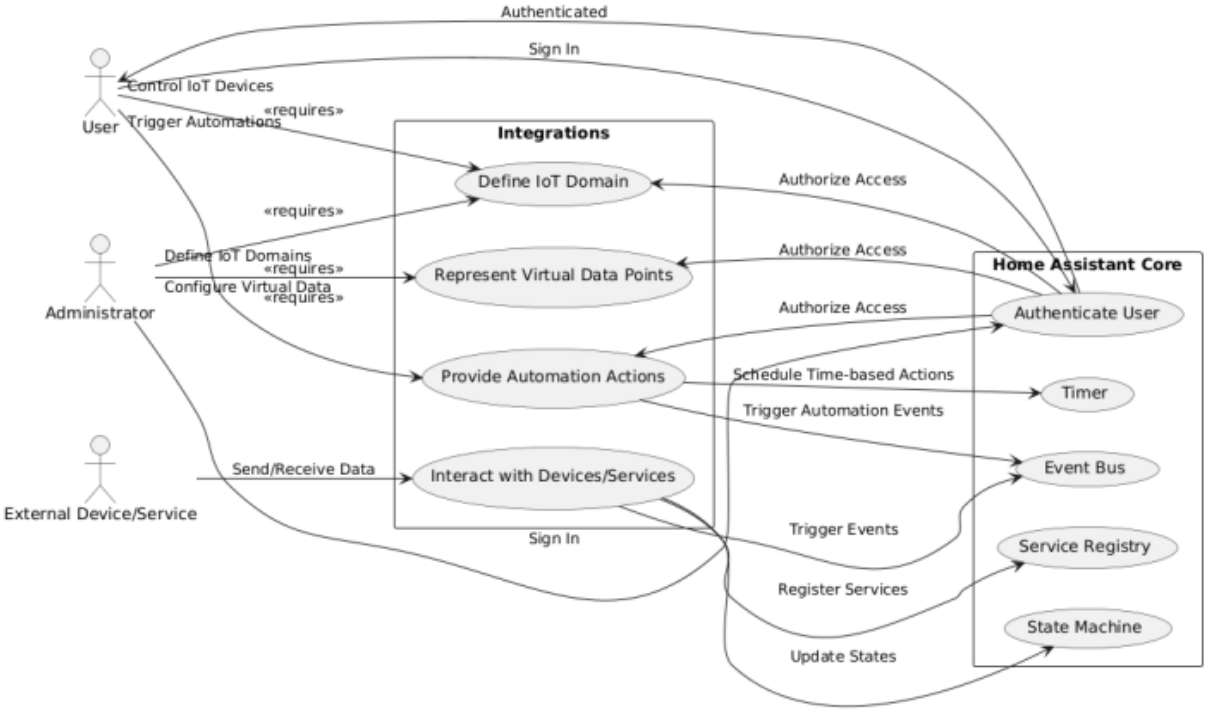

Figure 5: Different interactions amongst the various actors of the Home Assistant Core

The diagram above highlights the primary use cases for managing IoT devices, integrations, and automations within Home Assistant. Users interact with devices like lights or thermostats through the Home Assistant interface, allowing them to control devices or trigger automation sequences. Meanwhile, the Administrator oversees system-level configurations, including defining IoT domains, setting up integrations, and creating virtual entities to support more advanced workflows. External devices and services, such as smart bulbs or third-party APIs, communicate bidirectionally with Home Assistant, facilitating smooth data exchange and state updates.

The graph illustrates several core use cases, including authentication processes to regulate system access, the definition of IoT domains for organizing devices, and interactions with devices or services to send commands and receive status updates. It also enables automation tasks, such as scheduling the turning off of lights at a specific time, as well as the use of virtual data points to support software-based controls and more complex workflows. Key system components that support these functionalities include the Timer, which manages scheduled tasks, the Event Bus, which facilitates communication between system events, and the Service Registry, which handles service calls. Authentication is critical for securing user access to the system, while the State Machine ensures that device states remain consistent across the platform.

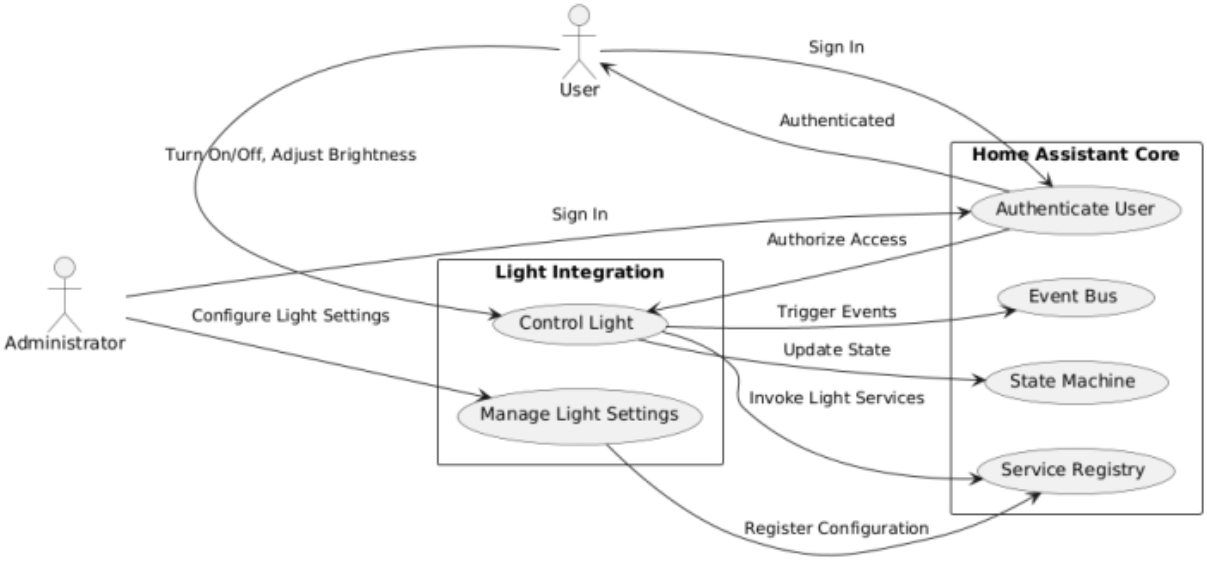

Figure 6: Smart Bulb Example

For instance, when a user wishes to turn on a smart light, the process begins with the user logging into the Home Assistant system via the Authenticate User function. This ensures that only authorized individuals have access to control devices, maintaining the security of the system. After successful authentication, the user sends a command to turn on the light. This request is then routed through the Light Integration, which is responsible for managing interactions with the smart bulb. Upon receiving the command, the Light Integration triggers the Event Bus in the Home Assistant Core, initiating the processing of the command.

As the request is processed, the State Machine updates the light's state to "on," ensuring that the system reflects the actual status of the device. This real-time update is crucial for maintaining accurate device state information. Simultaneously, the Service Registry activates the appropriate service to execute the command on the smart bulb, ensuring that the light responds to the user's request. A state diagram for this interaction would show various transitions, such as "off," "on," and "adjust brightness," all managed smoothly by the Home Assistant Core.

This example highlights how Home Assistant efficiently handles device-specific interactions, such as controlling a smart light. It demonstrates the system's modularity and scalability, enabling users to integrate and control a wide variety of devices within broader automation workflows. For simplicity, it's important to note that in this diagram, the Timer component has been collapsed into the Event Bus, streamlining the illustration of the process. This approach exemplifies the flexibility and power of the Home Assistant Core architecture, ensuring a seamless user experience while supporting extensive automation capabilities.

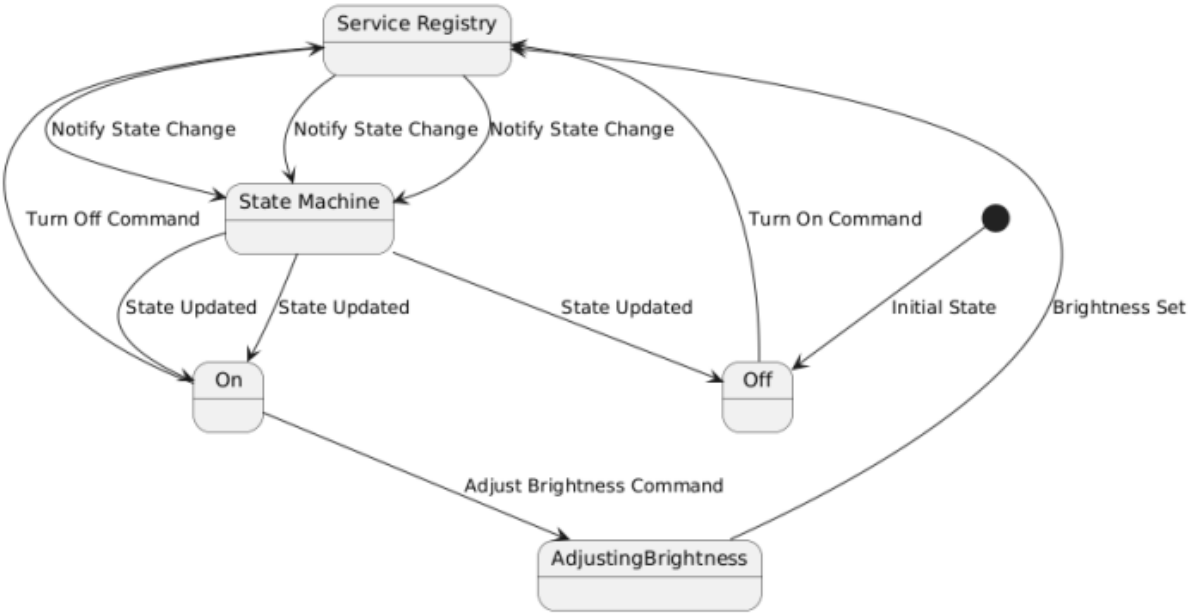

Figure 7: Service registry and state machine interaction

Conclusions

In conclusion, the conceptual architecture of Home Assistant is designed using the Implicit Invocation and Microservice architectural styles. These design approaches form the foundation of the application by incorporating an Event Bus for communication between connected devices and services. Additionally, the Microservice style supports modularity, enabling the various services within Home Assistant to operate independently and interact with one another. While the conceptual and concrete architectures differ in their level of complexity and the relationships between components, the core principles remain consistent. Both architectures utilize events to facilitate communication between components, with one component generating events that others respond to.

This report has provided us with an in-depth understanding of how these architectural styles play a crucial role in the development process. It also highlights the interconnections between the sub-systems of Home Assistant, which work together to enable a seamless flow of data across components and systems. The comparison between the concrete and conceptual architectures effectively demonstrates how the overall development process is structured and how the system's design supports its functionality.

Lessons Learned

The most important lesson is that concrete and conceptual architecture often differ from each other. This is true for both the conceptual made by our team and the one made by the developers on their main page. Based on our analysis, the difference is due to changes that occur during the implementation stage. These changes may be modifications, new functionality, new requirements, or optimizations. This lesson allows us to understand the development process and things that can pop up in the process which alter the direction of the project.

Our analysis of the architecture of Home Assistant deepened our understanding and appreciation of the styles involved. We gained first-hand experience with how architectural styles are used and how they influence how projects are developed. We learned how the Event Bus is used to maintain a proper flow between many different functions and how a system can be centred around it. This new viewpoint is necessary to properly understand the development process and it prepares us for when we have our own projects to develop.

Glossary

- **Implicit Invocation:** Architectural style where components and subsystems communicate by triggering events rather than communicating directly. This allows asynchronous event handling by decoupling the components.
- **Microservices Architecture:** Architectural style that enables a system to be loosely coupled and independently deployable, with components and subsystems each handling a specific function. Subsystems can be outsourced.
- **Event Bus:** The main component of event-driven architecture like Implicit-Invocation. Enables communication between different parts of the system by broadcasting events to subscribed listeners.
- **Service-Oriented Architecture (SOA):** Architectural style where the system is structured around independent components and subsystems.
- **Observer Pattern:** Behavioural design pattern, where an object notifies all its subscribers (which can be other objects) of its change in state.
- **Abstract Factory Pattern:** Creational design pattern, providing an interface to create objects belonging to a certain group without specifying their concrete structure.
- **Event Loop:** A programming construct that continuously handles tasks or events asynchronously.
- **Thread Pool:** A group of reusable threads that work together to complete tasks more quickly.
- **Backward Compatibility:** The ability of newer software to function with older systems or versions.
- **IoT Domains:** Functionality-based groups of IoT devices and services that make management simpler.

References

1. Schoutsen, P. Github. (2013). Home Assistant. https://developers.home-assistant.io/docs/architecture/core
2. Schoutsen, P. Github. (2013). Home Assistant. https://developers.home-assistant.io/docs/architecture_index/

3. Schoutsen, P. Github. (2013). Home assistant. https://developers.home-assistant.io/docs/dev_101_states/

4. Schoutsen, P. Github. (2013). Home assistant. https://www.home-assistant.io/

5. Schoutsen, P. Github. (2013). Home assistant. https://developers.home-assistant.io/docs/core/entity

6. Schoutsen, P. Github. (2013). Home assistant. https://developers.home-assistant.io/docs/architecture/devices-and-services

7. Schoutsen, P. Github. (2013). Home assistant. https://www.home-assistant.io/docs/configuration/

8. Schoutsen, P. Github. (2013). Home assistant. https://developers.home-assistant.io/blog/2024/04/08/deprecated-backports-and-typing-aliases/

9. Schoutsen, P. Github. (2013). Home assistant. https://www.home-assistant.io/integrations/timer/?utm_source=chatgpt.com

10. Home Assistant (2013) Core [Source code]. https://github.com/home-assistant/core