Discrepancy Analysis of Home Assistant

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

8 December 2024

Abstract

The report provides a comprehensive analysis of the concrete and conceptual architectures of Home Assistant, focusing on their structural organization, key components, and dependencies. Initially, the process of finalizing the concrete architecture was crucial to ensure a well-structured and efficient design. This involved referencing the architecture outlined on the Home Assistant website and comparing it with the actual codebase. The event bus, service registry, and state machine, while identified as core components in the conceptual model, were found to be integrated into a single class, Home Assistant, in the concrete implementation. This led to a more unified system, where all components interacted with the central core.py file.

To gain deeper insight into the system's dependencies and interactions, tools like Understand Software were employed, enabling the tracing of class dependencies and code changes. The analysis revealed over 86,000 dependencies within the system, highlighting the significant role of core files such as core.py, const.py, and config_entries.py. These dependencies were essential for the operation of the event-driven system, which utilizes implicit invocation through an event bus to facilitate communication between various components. In addition, the system was found to follow the principles of microservices, with loosely coupled components that can be added, removed, or replaced without affecting overall functionality.

The report also identified the presence of Model-View-Controller (MVC) architecture within the concrete design. The core.py file was found to act as the Controller, managing state changes (the Model) and responding to user interactions (the View). This MVC alignment, alongside the event-driven and microservices-based design, contributes to Home Assistant's modularity, scalability, and flexibility.

A key finding was the increased complexity of the concrete architecture compared to the conceptual design. New dependencies, such as those introduced in the auth and Helpers classes, were necessary to enhance the system’s performance and maintainability. For example, the auth class relied on a variety of components, including user and device registries, to manage user access and system security effectively. The concrete architecture also introduced refinements, such as the modularization of country codes and the removal of outdated dependencies, which improved efficiency and minimized errors.

Introduction

This report examines the concrete and conceptual architectures of the Home Assistant platform, focusing on their structures, dependencies, and key components. The purpose of this report is to explore the process of finalizing the concrete architecture of Home Assistant, which involves understanding how various components interact, identifying critical dependencies, and comparing the conceptual model with the actual implementation. By delving into the system's design, we aim to provide insights into its modularity, scalability, and flexibility, highlighting key features such as its event-driven nature, microservices principles, and the incorporation of Model-View-Controller (MVC) architecture.

The report is organized into several key sections. First, it discusses the process of finalizing the concrete architecture by mapping out the system's core components and understanding their dependencies. Tools like Understand Software were used to trace interactions between classes and identify important system components such as the event bus, service registry, and state machine. The following section explores the primary differences between the conceptual and concrete architectures, focusing on the implementation of the MVC pattern and the additional dependencies introduced in the concrete design. It also outlines how the system has evolved and the impact of these changes on performance, scalability, and maintainability. The

report concludes by summarizing how the concrete architecture supports Home Assistant's adaptability and resilience, ensuring its continued growth and functionality.

Salient conclusions drawn from this analysis include the discovery that the concrete architecture is more complex and tightly integrated than initially understood. The event-driven design and microservices principles remain central to the platform, but the concrete implementation introduces new dependencies and structural refinements to optimize system performance. Additionally, the use of MVC architecture enhances the organization of data flow, contributing to a more modular and scalable system. Ultimately, the findings emphasize the importance of understanding and organizing dependencies in order to improve system efficiency and ensure future flexibility.

Reflection Analysis Process

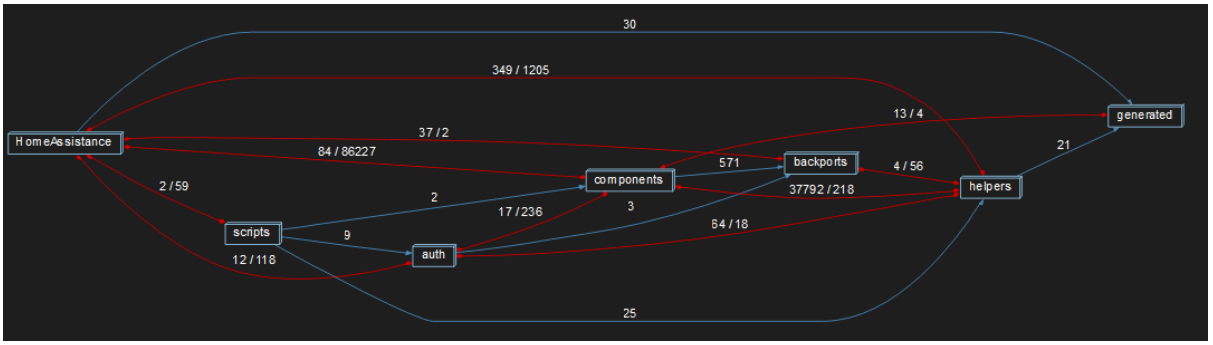
Before delving into the distinction between conceptual and concrete architecture, it was crucial to first finalize the concrete architecture and establish a thoroughly organized version. This step was essential to enhance the clarity and readability of the system, ensuring a more efficient and functional final design. To guide this process, we referenced the architecture outlined on the Home Assistant website, which identified key components such as the event bus, service registry, and state machine as the most frequently used entities in the system.

Upon examining the actual code, however, we discovered that the event bus, services registry, and state machine were not organized as separate entities but instead were integrated into the core.py file. These components were combined into a single class called Home Assistant, which served as the central point of interaction. Once this structure was clarified, we were able to establish the concrete architecture, allowing us to analyze how different classes interacted with one another.

To better understand these relationships, we utilized a tool called Understand Software, which allowed us to trace the flow of dependencies between classes. This tool enabled us to pinpoint specific lines of code that were imported into each class, providing a clear picture of how information was shared between them. This process helped us identify critical elements such as which low-level code entities were responsible for particular dependencies, who introduced or removed these dependencies, when these changes occurred, and the reasons behind these modifications.

Additionally, we were able to access the precise lines of code on GitHub that contained the dependencies. By reviewing the timestamps and associated commit messages, we could trace each change back to the individual who made it and gain insight into the rationale behind the updates. This level of detail was invaluable in understanding the evolution of the system’s design and functionality.

With this comprehensive understanding of the architecture and its dependencies, the reflection process became highly structured and efficient. The ability to trace each aspect of the system's development in such detail allowed us to draw meaningful conclusions about the design, ensuring a clear and organized approach to finalizing the concrete architecture.



Graph 1: Final dependency mockup in Understand

Architecture and Subsystems of Conceptual and Concrete

Both the conceptual and concrete architectures of Home Assistant are primarily event-driven, and they incorporate key principles of implicit invocation and microservices. At the core of Home Assistant's design is an event-driven architecture, which utilizes implicit invocation to facilitate communication between the various components of the system. This communication is achieved through an event bus, which serves as a mediator, enabling components within the subsystem to interact with the Core.py (the central Home Assistant) without making direct method calls. Instead, events are triggered through various mechanisms, including the State Machine, Timer, Service Registry, and Direct Events.

In parallel, Home Assistant also aligns with the principles of microservices architecture. Within this framework, the devices and services contained in the Components are loosely coupled. This loose coupling allows for the flexible addition, removal, or replacement of components without disrupting the overall system. As a result, the system's scalability is enhanced, as new devices or services can be integrated with minimal impact on the existing structure. This design further supports modularity, as it abstracts the interactions between different system components, minimizing direct dependencies and centralizing event distribution through the event bus.

Upon examining the new diagram from Understand Software, we observed that the concrete architecture also appears to incorporate elements of the Model-View-Controller (MVC) pattern. This observation stems from the way Home Assistant processes and manages data. In this context, Core.py takes on the role of the Controller, which manipulates the Model—represented by the states stored in an SQL database—based on requests made by the User from the View. This flow of data aligns closely with the MVC architecture, where the Controller handles user interactions and updates the Model accordingly, while the View is responsible for presenting the data to the user.

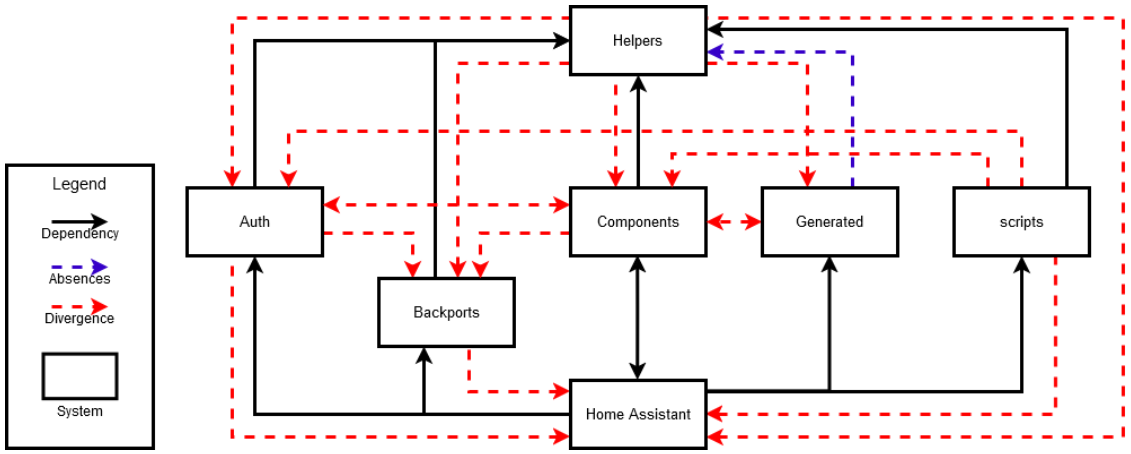
In essence, the combination of event-driven design, microservices principles, and the adoption of MVC in the concrete architecture offers a robust framework that allows Home Assistant to scale effectively, while maintaining flexibility, modularity, and efficient data management. This design also ensures that components can evolve independently, fostering a system that is both adaptable and resilient.

Comparison of Conceptual and Concrete Architecture

The primary distinction between the conceptual and concrete architectures of Home Assistant lies in the implementation of the Model-View-Controller (MVC) pattern within the newly derived concrete architecture, as discussed previously. This shift introduces new dependencies that enhance the overall structure of the subsystem, enabling a more

streamlined and efficient flow of data throughout the system. These newly established connections reinforce the architecture's MVC characteristics, as each component is responsible for providing specific services and has access to all the necessary data required to accomplish its tasks effectively. This interconnectedness between components highlights how the MVC style functions within the concrete architecture, facilitating better organization and data handling.

The concrete architecture, as a result, is considerably more complex and tightly integrated than the conceptual design. The new dependencies formed within this architecture are strategically introduced to optimize the system's performance and efficiency, rather than adding unnecessary complexity. These additions ensure that the system operates with greater precision, promoting smoother communication and data flow across the various components. In contrast, the conceptual architecture, while simpler, remains highly effective. Its modular nature—along with its reliance on the Helper component, which is integral to Home Assistant's functionality—allows for a well-organized, adaptable system that can still run smoothly if implemented as originally designed.



Graph 2: Deviations between conceptual and concrete architecture

Divergences Between Conceptual and Reflexion Model

Home Assistant

Initially, the function of Home Assistant and its components appeared to be structured in a way where Home Assistant itself called various functions, making it the primary entity dependent on multiple other functions. This conclusion was drawn from our conceptual understanding of the Core. At that stage, this structure seemed logical, as the Core's file organization resembled a pipe-and-filter approach. In this model, Home Assistant was envisioned as performing the bulk of the work, with other functions acting as helpers to handle specific tasks. Home Assistant would call these functions for particular requirements and continue its own operations, delegating certain responsibilities to the helper components.

However, upon mapping this to the concrete architecture, a much clearer picture of the actual dependencies emerged. The conceptual model had missed several crucial backward dependencies, which were much more extensive than initially thought. Notably, these backward dependencies came from areas such as Authorization, Backports, Scripts, and Helpers. Using Understand software to conduct a detailed investigation revealed that Home Assistant had a total of 86,312 dependencies. A significant portion—68% of these dependencies—stemmed from the fetching of constants found within the const.py file. The core file, core.py, accounted for 13% of all dependencies, playing a central role in the overall

architecture. Another 11% of dependencies originated from config_entries.py, which handles configurations for various connections, including Bluetooth. There was also an exceptions.py file that contributed 6% of dependencies, primarily related to error handling. The remaining 2% came from miscellaneous dependencies such as setup files, requirements, and bootstrapping processes.

core.py acts as the powerhouse of Home Assistant, being predominantly used by other components to read the event loop without blocking the process. This is accomplished through the callback function, which handles 95% of the calls, excluding those imports necessary before the calls are made. Notably, there were no lazy dependencies found in the system, although the closest example was the validate_state function, which is solely used by the template_entity.py file. A closer examination of this function, however, revealed that it plays a critical role in updating legacy configurations of entities to modern standards, further underscoring its importance in maintaining the system's integrity.

Which	Homeassistant\auth\providers\trusted_networks.py @callback def async_validate_access(self, ip_addr: IPAddress) -> None: To homeassistant\core.py def callback[_CallableT: Callable[..., Any]](func: _CallableT) -> _CallableT:
Who	Jason Hu approved by Paulus Schoutsen (Owner)
When	August 13, 2018 3:40am
Why	Callback is used by components outside of Home Assistant to call safe no-blocking reads because Home-Assistant uses multithreading. The event-loop won't be disturbed by these functions so they have the ability to read instantly without having to block others and read.

Table 1: Auth to Home Assistant

Which	homeassistant\backports\functools.py To Homeassistant__init__.py Scope -> entire home assistant directory
Who	Marc Mueller straight to main branch (Paulus approved dev?) Initial file creation by Erik Montnemery
When	April 07, 2024 1:15am
Why	Ensuring legacy code can still import old modules and use those attributes within home assistant

Table 2: Backports to Home Assistant

Which	Homeassistant\helpers\schema_config_entry_flow.py def wrapped_entity_config_entry_title (hass: HomeAssistant, entity_id_or_uuid: str) To homeassistant\core.py def split_entity_id(entity_id: str)
Who	Erik Montnemery straight to main branch (Paulus approved dev?)
When	March 30, 2022, 11:36 pm

Why	Gives the ability to the entity's title, the dependency is present because the entity cannot be retrieved from the service registry or the hass object. So it retrieves the attribute it is referred to in code within Home Assistant hence helpers depending on it.

Table 3: Helpers to Home Assistant

Which	Homeassistant\scripts\benchmark__init__.py async def valid_entity_id(hass) To homeassistant\core.py def valid_entity_id(entity_id: str)
Who	Paulus Schoutsen
When	February 24, 2020 8:35am
Why	Benchmarking to see the variations of entities, in this example light.kitchen. Home Assistant checks if <domain>.<entity> is intact with def valid_entity_id(entity_id: str). Scripts benchmark's calls that million times with async def valid_entity_id(hass). Documented as "Speed up validate_entity_id "

Table 4: Scripts to Home Assistant

Authorization

In the conceptual architecture, the authentication class (referred to as "auth" in the code) depends on the Home Assistant class, which in turn provides dependencies to the Helpers class. The auth class's primary function is to secure access to Home Assistant. When a user starts Home Assistant for the first time, an owner account is created, granting special privileges to manage user accounts and configure integrations. Auth inherits 118 dependencies from the Home Assistant class, including 62 variable imports from the const.py file. These variables are related to configuration (such as exclude and ID) and events (such as service registered and service removed). This setup is logical, as the auth class needs these variables to properly manage user access. The remaining 56 dependencies come from files like core.py, data_entryflow.py, exceptions.py, and requirements.py, bringing in methods and variables from external classes such as functools and typing. These dependencies are consistent across both the concrete and conceptual architectures, illustrating a key similarity.

In the concrete architecture, we observe additional dependencies not present in the conceptual model. The auth class now relies on about 64 dependencies from the Helpers class, spread across various files, including some from init.py and others from registry-related files such as area, category, and device, as well as function-related files like network, event, and storage. Of these, 31% come from init.py, 43% from registries, and 26% from function files. Using Understand and GitHub commit messages, we see that dependencies imported from device and entity registries into auth_store.py were added to improve the code accessing registries by refining the call to async_get_registry. These changes make the code cleaner and more efficient, enhancing readability and maintainability, though they do not introduce new functionality. Additionally, 17 lines of code are imported from typing.py in components, ensuring compatibility with Python 3.10. This enhances the clarity of type hints, improving the correctness of the code and helping developers better understand variable usage. Finally, three dependencies from backports are introduced, adding the partial variable from functools.py to improve performance in handling revoke token callbacks.

Which	homeassistant.helpers imports device_registry to auth_store.py

Who	Franck Nijhof
When	May 17, 2022 10:28 am
Why	Cleaning up accessing entity_registry.async_get_registry helper via hass

Table 5: Helpers to Auth

Which	typing.py imports Any, cast to _init_.py
Who	Marc Mueller and Franck Nijhof
When	Jan 23, 2023 9: 45 pm
Why	Refining or correcting the use of Optional typing in the code.

Table 6: Components to Auth

Helpers

The Helpers class provides utility modules with reusable functions and service-related support, making it dependent on many other classes to ensure all components have access to the necessary utilities. In the conceptual architecture, Helpers connects to auth, backports, components, and scripts. The largest dependency comes from Home Assistant (1,205 dependencies), followed by auth (18), backports (56), and components (218). Home Assistant's dependencies include async call functions and configuration variables like config_error and config_icon. Auth provides function-type variables from types.py, while backports contributes enum and functools.py, and components supply core variables for subcomponents like vacuum or water heater.

Although the Helpers class maintains consistent dependencies between the conceptual and concrete architectures, a key difference between the two is the lack of generated dependencies in the concrete architecture. Upon closer inspection, it was discovered that in the config_validation.py file, a large number of country codes defining currency were moved into their own dedicated file within the Helpers class. This change eliminated the previous dependencies that Helpers relied on for generation. To further enhance efficiency, a script was created to allow for seamless updating of country codes. This adjustment addressed issues where outdated currency codes might cause redundancies or errors when a country code was modified, removed, or added. This deviation in the concrete architecture contributes to improved efficiency and eliminates potential errors, ultimately streamlining the process.

Which	Removed currency = vol.In from config_validation.py
Who	Erik Montnemery
When	Nov 8, 2022 5: 12 pm
Why	Create repairs issue if an outdated currency code is configured

Table 7: Removed generated dependency from Helpers

Backports

The conceptual purpose of the Backport component remains the same: ensuring compatibility by bridging the gap between older Python environments and the latest Home Assistant features. It provides compatibility for older Python versions without requiring user

upgrades, and modular backporting for specific Python libraries (enum, functools). This high-level abstraction allows Home Assistant to maintain uniform functionality across different setups while simplifying developer workflows. Number of Dependencies is limited to Python's built-in libraries and possibly other Home Assistant modules. Number of Components is three modules (__init__.py, enum.py, functools.py). The rationale is that backports reduce fragmentation, simplify development, and increase accessibility for diverse environments.

Which	#pylint: disable-next=hass-deprecated-import (homeassistant/backports/functools.py)
Who	Bdraco, J. Nick Koston
When	Oct. 3, 2024, 6:53 PM EDT
Why	Add pylint rule for cached_property

Table 8: Backports to Home Assistant

Which	homeassistant.helpers.deprecation (homeassistant/backports/enum.py)
Who	cdce8p, Marc Mueller
When	Apr. 6, 2024, 7:15 PM EDT
Why	Deprecated old backports and typing aliases

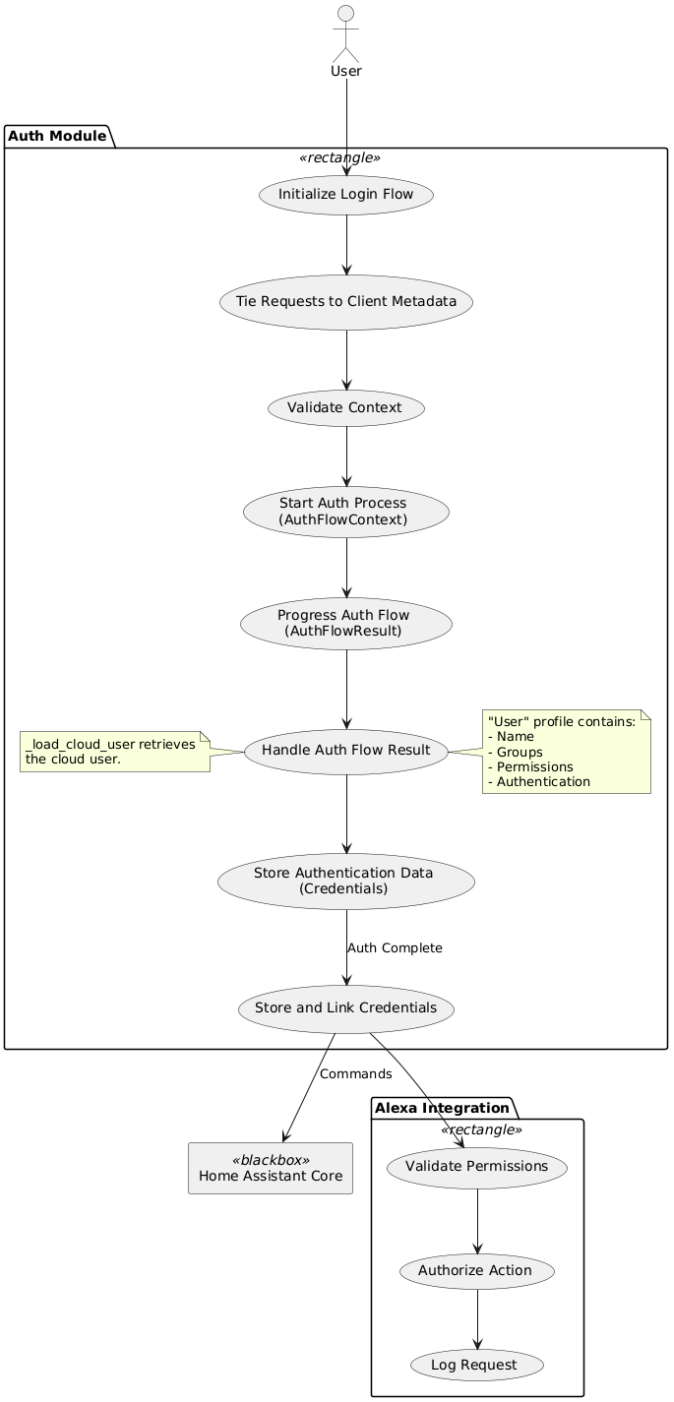
Table 9: Backports to Helpers

Components

The auth subsystem has 111 dependencies on the components subsystem, with the majority of these dependencies coming from init.py (42.3%), models.py (40.5%), and const.py (14.4%). One example of how these dependencies are implemented is the import of "User" from auth\models.py into components\cloud\prefs.py. The "User" object represents a user's profile within the system, containing data such as their name, groups, permissions, and authentication details. This is utilized in the _load_cloud_user function, where it retrieves the cloud user associated with the system. The commit that references the User import was made to improve code practices, specifically to adhere to the PEP8 standard for imports, utilizing the isort tool for better organization.

Which	homeassistant\auth\models.User To: homeassitant\components\cloud\prefs.py
Who	Basnijholt authored and Fabaff committed
When	December 8 2019 12:01 EST
Why	Use isort to sort imports according to PEP8 for cloud

Table 10: Auth to Components



Graph 3: Use case of Alexa: Auth Component

The backports subsystem has 555 dependencies on the components subsystem, with these dependencies split between enum.py (34.4%) and functools.py (65.6%). For example, functools is imported from homeassistant\backports\functools.py into homeassistant\components\arcam_fmj\mediaplayer.py. In this code, functools is used within the async_setup_platform function to wrap coroutine functions with decorators such as @functools.wraps. This ensures that metadata is preserved, improving introspection, debugging, and integration with the Home Assistant framework.

Which	homeassistant\backports\functools.py To: homeassistant\components\arcam_fmj\mediaplayer.py
Who	elupus
When	August 14 2023 12:03 PM EDT
Why	Avoid leaking backtrace on connection lost in arcam amd to Correct ruff error after rebase

Table 11: Backports to Components

There are 13 dependencies from generated to components, the fewest compared to other subsystems. Upon examining the codebase, we observe that supported engines, regions, and voices are imported from amazon_polly.py in generated. These dependencies are utilized in various parts of the code. For instance, SUPPORTED_REGIONS is a predefined list or set of valid values representing the acceptable regions for the service being configured. It is used with vol.In(SUPPORTED_REGIONS) to validate the input for the CONF_REGION configuration option, which supports Amazon Polly's text-to-speech services.

Which	Homeassistant\generated\amazon_polly To: homeassistant\components\amazon_polly\tts.py
Who	Robbiet480 authored and pvizeli committed
When	January 26 2017 5:22 PM EST
Why	Add an Amazon Polly TTS platform * Remove SPEED_MED from fan * Add an Amazon Polly TTS platform * Update boto library version for notify.aws_* platforms to match the tts.amazon_polly req * Improve log line and add docstring to function * Simplify config logic * Remove duplicate logic * Don't know how this got in here... * initial options work * Remove stale config option and only allow supported languages * Make requested changes * Polly is only supported in some regions * Allow filename to contain underscores (for amazon_polly platform name), remove unnecessary default_lang, other small things * Add options dict to service description

Table 12: Generated to Components

In the concrete architecture, we observe dependencies that are not present in the conceptual model. The components subsystem relies on 37,858 dependencies from helpers, which are spread across various file paths within the helpers module. The majority of these dependencies originate from init.py, entity_platform.py, and device_registry, accounting for 28%, 13.7%, and 11.4% respectively. To simplify, let’s focus on device.py, which contains

the async_remove_stale_device_links_keep_entity_device function. This function removes outdated device links related to a given configuration entry while keeping the link to a specified device. The device to retain is determined using the source_entity_id_or_uuid parameter, which is resolved to the corresponding device ID via the async_entity_id_to_device_id function. The actual removal process is handled by the async_remove_stale_devices_links_keep_current_device function. This change improved code efficiency and system maintainability as the codebase evolves.

Which	homeassistant\helpers\device.py To: homeassistant\components\derivative\init.py line 9
Who	dougiteixeira
When	June 22 2024 9:03 AM EDT
Why	Add the ability to change the source entity of the Derivative helper

Table 13: Helpers to Components

Generated

The Generated component acts as a central data host for dynamically created configuration data used by other components. In the conceptual architecture, Home Assistant is the only subsystem that directly depends on Generated. Home Assistant dynamically parses and generates configuration data through Generated, utilizing this data to support critical features essential for its operation. The conceptual architecture also shows that Generated depends on the Helpers class. Additionally, the architecture of Generated suggests a microservices design, where communication occurs with components like Home Assistant, Components, and Helpers through an event bus.

In the concrete architecture, further analysis using Understand revealed a convergence between Components and Generated. These two components are interconnected through several function calls. Specifically, Components makes four calls to the Final module within typing.py, making Generated dependent on Components. This interaction ensures that the values within Generated are explicitly type-checked and immutable, reducing the likelihood of coding errors and improving type safety.

Which	Generated/bluetooth.py from typing import Final To components/simplisafe/typing.py Scope -> Generated imports typing directly from Components
Who	Author: Jakob Schlyter Approver: Epenet
When	May 22, 2024
Why	Ensures values are immutable and explicitly type-checked. This improves type safety and reduces potential errors in code

Table 14: Generated to Components

Scripts

In the Conceptual architecture, the Script component is used by the Home Assistant sub-system. Script depends on the Helper component to complete its tasks. The relationship that Scripts has to the other components and its role in the sub-system as a benchmarking toolkit, strengthens the fact that the Script component is an important part of the architecture and supports our claim that the architecture has aspects of both the microservice and MVC styles.

The script component has new connections in the new concrete architecture, as it now shows dependencies with Components and Auth. It depends on these components to call specific data which it uses to complete its tasks. The relationship between these components is like the relationship Scripts has with the Helper component. Scripts have a total of 125 calls to other components and this number shows just how much Script relies on them. The Home Assistance sub-system is the only component that calls Scripts and only calls it twice, it calls it to run all its functions for a benchmark test. Scripts on the other hand calls Home Assistance the most,47.2% more than the other components. Most calls are imports from Home Assistance that allow it to do its tasks. Though not as vast as Home Assistant, in order to continue acting as a benchmark it calls Component and Auth components to collect the necessary data to complete its role. Whether it's managing users via Auth or checking files via Component, the tasks done by Scripts are crucial to the Home Assistant system.

The Script component is mostly the same in both architectures as its main connection is with Home Assistant and Helper. This connection is expressed with 47.2% of Scripts dependencies on Home Assistant and 20% with Helpers. The new dependencies that are present in the new concrete architecture are with Auth and Component, which only make up 7.2% and 1.6% of Scripts dependencies. The new dependencies are not as significant as the main ones as they are only there to help with a small number of functions. Other than those small number of functions and their connections, Script has the same role in both architectures.

Which	core/homeassistant/__main__.py
Who	KapJI/ Ruslan Sayfutdinov & Balloob/ Paulus Schoutsen
When	Dec 23, 2021, 2:14 pm EST & Jul 3, 2016, 2:38 pm EDT
Why	Fix pylint plugin which checks relative imports & Make scripts available via CLI (* Rename sqlalchemy migrate script * Add script support to CLI

Table 15: Home Assistant depending on Script

Which	core/homeassistant/scripts /auth.py
Who	Balloob/ Paulus Schoutsen
When	Jul 13, 2018, 9:31 am EDT
Why	*User management

	* Lint * Fix dict * Reuse data instance * OrderedDict all the way

Table 16: Script depending on Auth

Which	core/homeassistant/scripts /auth.py
Who	Epenet
When	Aug 18, 2024, 9:43 am EDT
Why	Improve type hints in scripts/auth

Table 17: Script depending on Component

Which	core/homeassistant/scripts /auth.py
Who	Bdraco, J. Nick Koston
When	Jan 20, 2024, 9:16 pm EST
Why	Always load auth storage at startup

Table 18: Script depending on Helper

Conclusion

In conclusion, both the conceptual and concrete architectures of Home Assistant share fundamental similarities in structure and design principles. At their core, both architectures incorporate elements of implicit invocation and microservices, which are key to ensuring the modularity and scalability of the system. These architectural styles facilitate smooth data flow between the various components, promoting efficient communication and reducing direct dependencies. As a result, both versions of the architecture support the overall flexibility and adaptability of Home Assistant, allowing it to seamlessly integrate with a wide variety of devices and services.

Despite these similarities, the concrete architecture introduces new dependencies that significantly enhance the system's efficiency. These additional dependencies not only support the maintenance of an MVC (Model-View-Controller) design but are also integral to the proper functioning of the individual components. By enabling the system to better organize and manage the flow of data, these new dependencies optimize the interactions between components, leading to improved performance and scalability. The concrete architecture, therefore, represents a more refined and efficient version of the conceptual architecture, one that has evolved through continuous development and refinement.

Furthermore, by examining the new reflection model in the concrete architecture and comparing it to the conceptual design, we gained a deeper understanding of how software architectures are formed, modified, and optimized over time. This comparative analysis provided valuable insights into the reasoning behind architectural changes, helping us better

appreciate the underlying principles that drive these modifications. It also highlighted the importance of adaptability and ongoing improvement in creating robust, scalable systems. Overall, this process has enhanced our awareness of the complexities involved in designing and maintaining software architectures, reinforcing the significance of both theoretical understanding and practical implementation.

Lessons Learned

One of the most important lessons we learned throughout our research for this report is the critical role of proper documentation in software development. The absence of comprehensive README files meant that we had to rely heavily on the comments within the code and the commit messages to understand its functionality. Unfortunately, many of the comments were minimal and only offered basic descriptions, such as file titles or brief explanations of individual lines of code. This limited commentary provided very little insight into the overall design or purpose of the code, forcing us to examine each line of code carefully and make inferences about how the functionality was being achieved.

This lack of sufficient documentation significantly slowed down our process and introduced considerable difficulty as we attempted to piece together the underlying logic and reasoning behind various components. Without clear, explanatory documentation, it became a time-consuming task to decipher how the different parts of the code interacted and worked together. Moreover, this challenge could have been easily avoided if more detailed documentation had been provided, particularly in the form of higher-level explanations and an organized structure for understanding the architecture.

Additionally, proper documentation would not only have made our task easier but would also be invaluable for future developers working on the code. Well-documented code offers a clear map of the system's architecture, enabling developers to quickly grasp how different modules and components are designed to function. It serves as a crucial resource for troubleshooting, maintaining, and enhancing the system over time. By providing context, explanations, and usage examples, documentation can also save developers from having to spend excessive time trying to decode the inner workings of the system. Ultimately, clear and comprehensive documentation is a key factor in ensuring the longevity, maintainability, and ease of understanding of any software project.

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**: 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.
- **Model-View-Controller**: An architectural style that separates an application into three interconnected components: Model, View, and Controller. This separation enables modularity, scalability, and testability by clearly defining the roles of each component
- **README**: A common way to document the contents and structure of a folder and/or a dataset so that a researcher can locate the information they need
- **Benchmark:** A performance evaluation method or standard used to measure, compare, and optimize systems, software, or processes against predefined criteria. It

provides a quantitative basis for assessing efficiency, effectiveness, and improvements over time.

- **Reflexion Models:** A technique in software architecture used to compare a high-level planned architecture (intended model) with the actual implementation (source model) of a system. It provides a structured way to understand and address discrepancies, enabling improved alignment between design and implementation.
- **Metadata**: refers to "data about data." It provides information that describes, explains, or gives context to other data, enabling better organization, understanding, and usability. Metadata can exist at various levels, from files and documents to databases, systems, or web content.
- **Four W's**: Which, Who, When, Where. They're questions that help understand code files similar to commits in Git. It's the format for the StickyNote recovery method.
- **StickyNote:** Helps map code changes to entities and dependencies instead of lines. Is used with Reflexion models for data recovery and is used to keep track of Git commits and the rationale behind them.

References:

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