Kodi Architectural Enhancement Report

Authors:

Carter Brisbois	20cbb2@queensu.ca
Dan Munteanu	20dmm14@queensu.ca
Maverick Emanuel	20mje3@queensu.ca
Nathan Gerryts	20njfg@queensu.ca
Oliver Dantzer	20ond1@queensu.ca
Trayden Boucher	trayden.boucher@queensu.ca

Abstract:

For this report, our group was tasked with proposing a new feature or enhancement for the media player software Kodi. We then had to integrate this feature into our previously created concrete architecture. Our proposed feature is integration with popular streaming services within Kodi. This would be a very challenging implementation, however, the benefits it would provide would greatly outweigh the difficulties. It would present users with a much wider range of content directly within Kodi. To implement this, we identified two different ways it could possibly be done. To implement this feature, we analyzed the two implementations and chose the better of the two. The architecture of Kodi changed drastically from doing so. In our previous investigations, we found that Kodi uses a combination of both object-oriented and layered architectural styles. This new feature requires a client-server architectural style on top of the other styles already being used. This investigation solidified our knowledge on the topics that have been discussed in class throughout the semester. It was a culmination of everything we've learned; it put our software architecture skills to the test to not only think of a feasible idea but to implement it as well.

Introduction and Overview:

With the new era of streaming entertainment, we believe that Kodi has been left behind by its competitors (Netflix, Disney+, etc.). That is why our architectural enhancement is to implement integration with popular streaming services (legally and with appropriate licensing) to allow users to access a wider range of content directly within Kodi. Kodi will now use API's provided by streaming services to access the content from that platform. The user would require a subscription to the service but would now get the customizability and perks that Kodi provides. Within Kodi, the user would then be able to stream and download content from any added streaming services. This introduces a client-server model that must be implemented within Kodi's architecture. The client is the user within Kodi that accesses the content of the respective streaming service through an API. Within the client-server connection, there will be an authentication token so that the connection between the

user and the streaming service can verify the validity of the user's account. Furthermore, Kodi should implement a GUI to seamlessly integrate the streaming service content with locally stored content. There are two ways to implement this feature within Kodi's architecture: one where the client-server model is implemented with the content-management subsystem of Kodi, or where a new top-level object is created. We decided to choose the option of implementing the model within the content-management subsystem of Kodi, due to the fact that the subsystem has access to the Kodi databases, making the implementation of its features and customization easier. Of course, as this is a suggestion for Kodi, there are going to be legality obstacles that come with this feature. If the obstacles are overcome, however, then Kodi will have an explosion of new users that would love what Kodi has to offer. Unfortunately, this implementation does not come without drawbacks. The first of which is coming to a legal agreement with the different streaming services. The second is that Kodi will now be reliant upon their competitors services, as well as their implementation of media-based content. Using S.A.A.M. analysis we will explore non-functional requirements of this enhancement, and how the stakeholders will be affected as well as their roles in the implementation process. As well, we suggest that Kodi will implement a database for user authentication. Since the content will be managed on Kodi's end, it is important for Kodi and the respective streaming services to have a partnership when it comes to authentication. That is why we believe Kodi should have a central database with profile information and an authentication token that can be accessed by the respective streaming service so that everyone can be verified with validity. One aspect of Kodi that is explored in the use cases is the ability to download content, if legality issues can be overcome, Kodi will have the ability to download multiple shows and movies from various streaming services all on one centralized location. To conclude, we believe that including a client-server connection to the most popular streaming services would be extremely beneficial for Kodi and improve the overall user experience very much. Despite the fact there are legal obstacles in the way, a software where all the user's media content is centralized from different streaming services is just what Kodi needs.

Architecture:

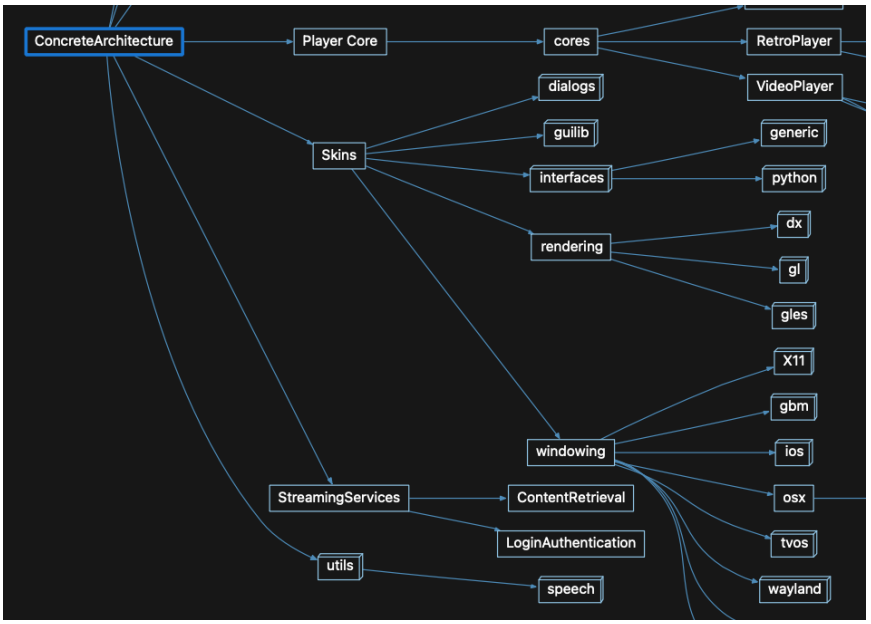
We were tasked with creating two implementations of our new proposed feature for Kodi. In both cases, a client-server architecture is necessary to provide the user with the requested content from the streaming platform. As previously stated, in order to comply with licensing laws, you require an account for the streaming service in order to use it within Kodi. Thus, account authentication is necessary to ensure the user has a subscription to that service. Furthermore, in order to stream content, Kodi must contact the server to retrieve it. In both implementations, these are the core two components that comprise the module dedicated to streaming services.

Our first implementation involved creating a new top-level object in Kodi's architecture that contains all necessary files for retrieving content and for authenticating the streaming service account. This new object would then have interactions with nearly every other top-level object. The issue with this, however, is that it leaves many dependencies between objects that could possibly be minimized by implementing it differently.

To solve this issue, in our second implementation, we created a submodule in the ContentManagement object to handle the required functions that the StreamingServices object handled. Within ContentManagement, there is a submodule dedicated to sources. Normally, this submodule handles locally saved media on the user's device. For our objectives, however, we now require a third source. The streaming service server is another source of content so it fits perfectly. By implementing this new functionality within another pre-existing object, we are minimizing dependencies between objects and keeping the code as organized as possible. This makes the code much more maintainable for the future and makes it much easier to understand.

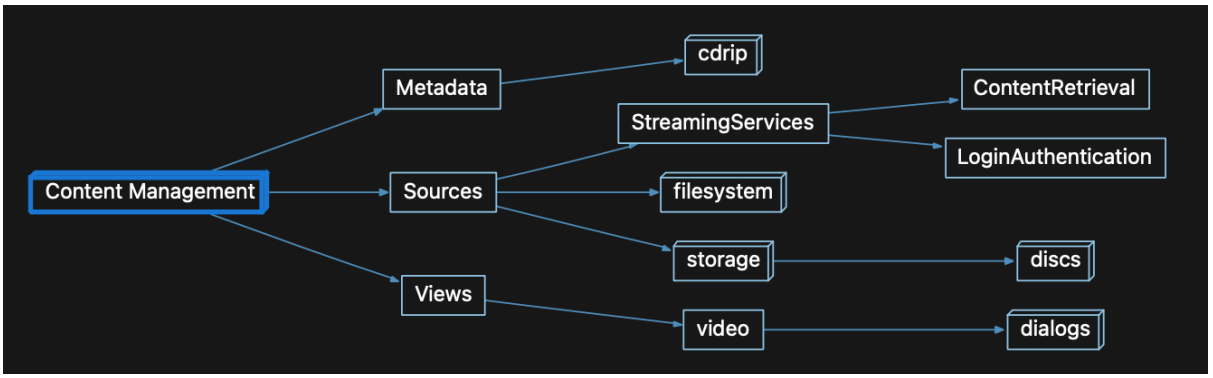
As previously mentioned, this new functionality requires a third architectural style. A client-server architecture is necessary to transfer the data from the streaming service to the user's device. For both implementations that we came up with, the streaming service server only interacts with the StreamingServices module. For the first implementation, the StreamingServices object interacts independently with the server. The object then uses the data retrieved to interact with the other objects. For the second implementation, the StreamingServices submodule interacts with the server. It can then use any other functionality from the ContentManagement object before interacting with other objects. Implementing the client-server architecture surprisingly doesn't interrupt the flow of nearly any other objects or submodules within Kodi's architecture. For both implementations, the submodules ContentRetrieval and LoginAuthentication have the same functionality. The functionalities of both submodules are expanded upon in the use-cases.

Implementation 1: New top-level object StreamingServices



Build a new top-level submodule StreamingServices. It would interact with the submodules ContentRetrieval and LoginAuthentication within itself.

Implementation 2: New submodule StreamingServices



Within ContentManagement, create a new submodule StreamingServices within Sources. It would interact with new ContentRetrieval and LoginAuthentication modules within itself.

S.A.A.M Analysis:

Non-functional Requirements

The non-functional requirements for our addition are crucial for ensuring a robust and user-friendly experience. First and foremost, authentication mechanisms must be implemented to guarantee the secure access of streaming services, prioritizing user data protection. The implementation should also prioritize ease of integration with various streaming services, allowing for the addition of new streaming services with minimal effort. Furthermore, the integrated streaming services must be accessible across a wide range of devices and platforms, promoting flexibility. Long-term compatibility is ensured by designing the system to adapt to changes in streaming service APIs. The integration should seamlessly blend with Kodi's user interface, maintaining consistency and intuitiveness for users

transitioning between locally stored content and streaming services. Lastly, strict adherence to licensing agreements with streaming service providers is essential to ensure legal compliance and a sustainable partnership between Kodi and the integrated services.

Stakeholders

The major stakeholders impacted by our proposed enhancement are Kodi developers, Kodi investors & shareholders, stakeholders of alternative streaming services, and users. For Kodi developers, the most important non-functional requirement is ensuring that the architecture is modular, making it easy to update or add additional streaming services without affecting the overall system stability. For Kodi investors and shareholders, it is important that the implementation adheres to licensing agreements with streaming service providers. They want to avoid any legal issues, following regulations and the legal agreements for the licensing deals. For the stakeholders of alternative streaming services, they as well want to ensure the enhancement follows licensing agreements. The alternative services are profitable businesses at the end of the day, and so following the legal obligations are required. Lastly, for the users of Kodi, their most important non-functional requirement is supporting easy integration with various streaming services. This would allow the addition of new services with minimal effort, and allow users to use the enhancement to its maximum capability.

Approach 1: New top-level object StreamingServices

The first approach is creating a new top-level subsystem. In this approach, the enhancement is its own subsystem. However, adding a new subsystem is complex, and the maintainability is higher than a new submodule. By adding a new submodule, this does add more modularity and flexibility for future features. However our enhancement is limited in what functionality it will need to provide - there is no need for support for any major additional features, and so implementing the enhancement via an entirely new top-level object is overcomplicated and over complex for the features and capabilities of the enhancement.

Approach 2: New submodule StreamingServices

The second approach would be to create a new submodule StreamingServices within Content Management. Compared to a new top-level subsystem, there will be less code changes, less maintainability, and the complexity of implementation will be lower. As well, this will impact the system less dramatically, as adding an entirely new subsystem creates a lot more dependencies and testing, directly impacting maintainability and costs.

Our Chosen Approach

Since the first approach is overly complex for the scope of this enhancement, leading to an increase in maintainability, costs, and implementation complexity over

approach 2, this is why we have chosen the second approach, a new submodule, as the best way to realize the enhancement.

Effects of Enhancement on the System and Risks

With our enhancement, the system now needs to maintain proper API integration with other streaming services. As time goes on, the streaming services will evolve and change their API and its documentation, requiring Kodi developers to modify. Combined with the new maintenance, the features would require new tests for the enhancement. This would include tests for all essential API integration features, such as movie requests and user authentication, as well as maintenance for these tests as the API evolves. To test the impact of interactions between the proposed enhancement and other features, tests will need to be designed to determine any bugs or changes to the system as a result of the new submodule. For example, regression tests run after every change to ensure our enhancement introduces no bugs into the system. As the system evolves, it will need to scale horizontally, as it will need to accommodate more users. This will especially be seen through network traffic, which needs to be scaled to handle the influx of API requests and calls. With this scalability comes an increase in performance requirements. With the added API requests, this increases the network traffic and usage from users, which need to be handled via the CPU. This increases the CPU utilization, meaning the performance requirements for Kodi are higher if you want to use this feature.

In terms of the conceptual architecture, our chosen approach for implementing the enhancement would add a submodule to the Content Management subsystem, with dependencies between the two that allow the new StreamingServices module to receive and send content to other parts of the system. At a low-level, this would include functionality and files for receiving content from streaming services and sending this content to the file system, where the video codecs are then able to decompress the video for video playback.

Due to the features of our enhancement, there are some potential risks. For one, the legality of proper and secure use of the content of other streaming services. With licensing agreements, Kodi developers are still in charge of proper use of the licensed content, meaning any sort of bugs or glitches that might break their system, such as ones that allow users to pirate content, will make them at fault. As well, the proper handling of user authentication information, such as passwords, is critical. The developers must account for security and use common practices, such as password encryption, for protecting the login information of its users.

External Interfaces:

- Translations, Fonts, Skins via GUI
- Media (Images, Video, Audio) via downloaded files on disk and from server
- Media via streaming services for viewing or saving locally

- Scrapers to read & extract metadata from external databases to fetch information about movies, shows, music, and images
- Streaming servers
- User information & login
- Streaming service account authentication
- Video codecs

Use Cases:

Use case 1: User authentication

For any streaming service account verification is necessary. Kodi requests an authentication token from the streaming service API using the user inputted username and password. This token would then allow Kodi to request the metadata of the movies and shows available on the streaming service from the streaming services API.

Skins:

Upon selection of the streaming service integration tab, the user will be presented with a list of streaming services to add. Upon clicking a streaming service, they will be prompted for the username and password they use, which will be sent to the Interface layer. Then upon receiving a response of error or success. the user will then be notified of the result, and given the error message if error.

Interfaces:

Receives username and password from Skins. Username and password are given to the file sharing layer, which will return either an error, where Interfaces will return the error to Skins, or an authentication token, which will be sent to the Content/Data Management layer, and Interfaces will return Success to Skins.

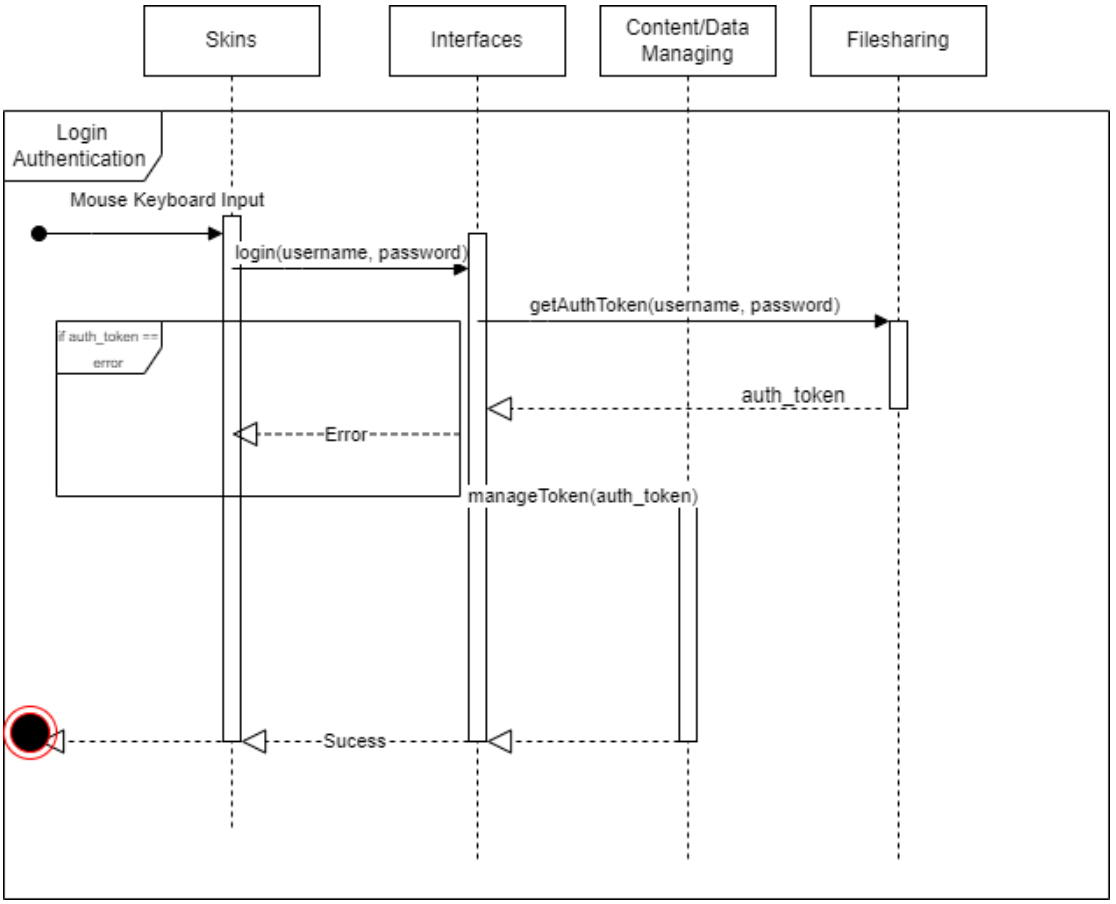
File Sharing:

Receives username and password from interface. This is where Kodi will access the internet and request a user auth token from the streaming service API using the username and password. It will return the token if the request was successful, or the error with the error message from the streaming service if it was not successful.

Content/Data Management:

SQLite database will have a table with columns profileID, streaming_service, auth_token_key. Receives user auth token from Interfaces layer, and inserts into the database the streaming service name, the token they used, and the ID of the profile they are using are inserted into the database. This will be used in the future whenever metadata, video streaming, and video files need to be requested from the streaming service’s API.

Use Case 1 Sequence Diagram:



Use Case 2: Downloading media content from the streaming service to save locally on Kodi

One of the key features of our addon would be the ability to save shows from the streaming service locally to be viewed offline or at a later time offline. This would work similarly to how streaming services allow you to download shows. Here's a breakdown of each component and how they transfer control.

Skins:

The skins subsystem handles the GUI elements. It is highly customizable to fit the user preferences. The subsystem handles the user controls and input as well as the output of the GUI.

Interface:

Once the user clicks on a control element (button, link), Skins passes control over to the interface. In this use case, The user has selected a show that they would

like to download. The interface then, would call the Content Management subsystem to access the login token.

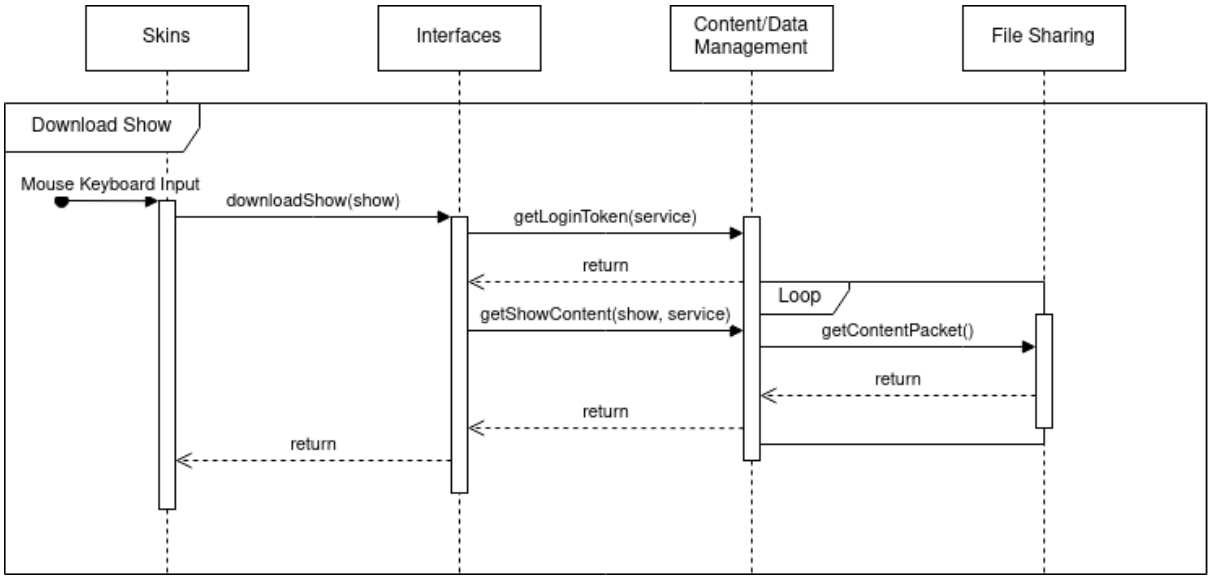
Content/Data Management:

Content Management gives the login information and token to access the streaming platform API and services, This use case assumes the token has already been generated. This subsystem also stores the content from the streaming service onto the hard drive once the download begins.

File Sharing:

This subsystem handles web communication between the Kodi and the streaming server, once the download begins, the packets will be passed from File Sharing to Content Management to be saved. This process loops until the download is completed.

Use Case 2 Sequence Diagram:



Data Dictionary:

Software: Applications that are run by a computer.

Software architecture: The design and structure of applications.

Layered architecture: A style of software architecture that involves dividing a program into layers that each perform a specific role.

Object-oriented style: A style of software architecture that involves dividing a program into self-sufficient objects that may interact with each other.

Client-server style: A style of software architecture that involves a system that hosts, delivers, and manages resources that the client requests.

Codec: A process that can compress or decompress audio or video data.

SQLite: A database engine for developers to embed in their apps.

Naming Conventions:

API: Application Programming Interface

GUI: Graphic User Interface

S.A.A.M.: Software Architecture Analysis Method

Conclusions:

The improvement of adding integrated Streaming Services to Kodi allows for both an improvement in user experience and lucrative prospects for both Streaming service and Kodi shareholders. While adhering to licensing agreements may be problematic and difficult, This improvement would be a tremendous opportunity to increase Kodi's practicality and user base. When applying a new integrated Streaming Service feature to Kodi our group decided on 2 different approaches. The approach chosen was creating new submodules in a top-level system that allowed access to a streaming service login information and content database. This approach was chosen due to its simplicity and maintainability. This approach would add a new subsystem called StreamingServices in the content management system. In this new StreamingServices folder, 2 essential files are included. The content retrieval file is paramount to accessing the streaming service content data; this is how a user would either download or stream a movie or tv show. The other file is LoginAuthentication, this file is used to make sure that a Kodi user is subscribed to a streaming service that they are trying to use. This new feature would allow Kodi to become the only necessary media service interface a user would need to interact with daily, allowing for an easy and comfortable experience navigating one's increasing mass of accessible content.

Lessons Learned:

Like all previous reports, this project gave us all more experience working in a group-oriented coding environment to further develop our communication and teamwork skills. During the creation process of this report, our group had the opportunity to apply content learned in class to a real-world application of software architecture. This gave us insight into the development process of adding new features to a real-world application. Brainstorming new additions for Kodi required critical thinking about what Kodi offers as a media player in order to determine further improvements. We had to take many different factors into account when considering. We had to consider the stakeholders, the user experience, the real-world feasibility, and many other factors. Once we had an idea for an improvement, this report allowed us to apply our prior knowledge of software

architecture to Kodi to devise a plan to implement our proposal. We had to consider how it would fit into the pre-existing architecture and the best possible way to do so. This strengthened our understanding of not just Kodi specifically, but also how an actual programmer would add to pre-existing software. In conclusion, this project consolidated what we've been learning about in class throughout the entire semester and allowed us to further develop the skills we've gained by working on the prior two reports.

References:

Corrales, L., Fonseca, M., González, G., Loría, J., & Sedó, J. (2023, July 10). *Architecture*. Official Kodi Wiki. https://kodi.wiki/view/Architecture

Dreef, K., van der Reek, M., Schaper, K., & Steinfort, M. (2015, April 23). *Architecting software to keep the lazy ones on the couch*. Delft Students On Software Architecture. http://delftswa.github.io/chapters/kodi/