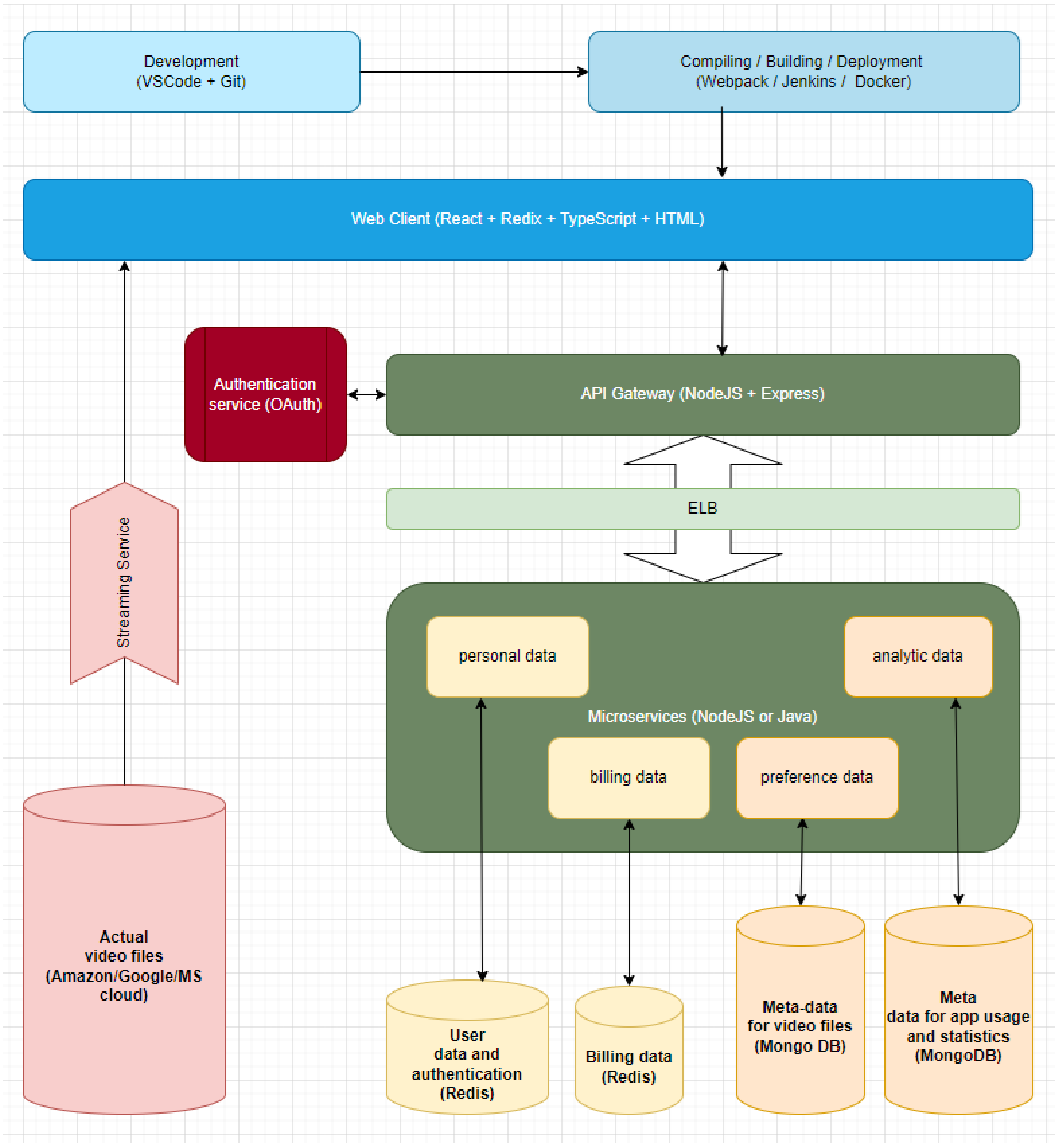
Streaming Platform Architecture

Purpose

This document presents a high-level solution diagram. Some detailed explanations for the most important layer are provided as well.

High level diagram


Development (VSCode + Git)
Compiling / Building / Deployment (Webpack / Jenkins / Docker)
Web Client (React + Redix + TypeScript + HTML)
Authentication service (OAuth)
API Gateway (NodeJS + Express)
ELB
Streaming Service
personal data
analytic data
Microservices (NodeJS or Java)
billing data
preference data
Actual video files (Amazon/Google/MS cloud)
User data and authentication (Redis)
Billing data (Redis)
Meta-data for video files (Mongo DB)
Meta data for app usage and statistics (MongoDB)

Detailed explanations

This includes mostly the layers related to UI, communication and storage.

Client-side layer:

This layer is responsible for generating the entire UI, giving the user ability to interact by trigering API requests. Technologies used here need to easily support following:

- Component oriented design. This is critical for a maintainable application. This architecture will allow easy extensions without breaking existing functionality. This "composition" pattern should be preferred over the "object-oriented" approach. Uni-directional dataflow should be preferred due to its simplicity and easily predictable behavior. The best choice here will be React, due to its well know component-based approach, popularity, well established community and a lot of extension libraries.
- State management system. This should allow easy storage and centralized access of any data used across the application. State updates should be standardized with action dispatching system. State handling should be centralized to specific set of functions only. State changes need to trigger events that could be easily observed in order to make UI updates. This centralized state, mirroring the look of the application UI, can be also stored on the server, if needed, for quick server rendering or just to easily do "continue-from-where-you-left" functionality. There is out of the box solution that provides all of these requirements – Redux. An easy to integrate implementation of Redux for React is also available. Consider using Ract-Redux with its toolkit.
- Code should be written in Typescript so it can benefit from interfaces and enforce clear contract between components. This is currently the front-end development industry standard.
- CSS – preprocessor needs to be used, like LESS or SASS. This will ensure easy extractions of mixins and tokens that will guarantee consistency across the application and alignment with UX visuals.
- Webpack bundling. This is critical for the performance on the front-end, especially the loading phase. This will allow separation of modules into different bundles which can be then served on demand.
- Device recognition – Performance in real time for streaming application is critical. It is very important to optimize the data stream for the device user uses. This information should be kept on the client side and passed further to the Elastic Load Balancer (ELB)

Important note – for more detailed explanation of the main aspects of component separation and other must have items, please refer to the "Movie Library Project" document.

Authentication layer:

This is critical for security. The best choice will be dedicating this to a 3rd party. There are many good solutions out there, like *Auth0* or a cloud-based *OneLogin*. These will hide all the complexity of managing passwords, recovery and multifactor authentication. They are also reliable in terms of prompt security patches.

API Gateway

This is the main API distribution point that will be hit directly by the web client. Following consideration should be made before choosing a technology. Here they are:

- This is real-time application. Many users are expected to be logged in and interact with the app – search for movies, write reviews, check details, etc. It is important to use an event driven, non-blocking model here.
- Microservices architecture is already part of the overall design approach. This layer should be operated by a tier that plays well and easy with microservices.
- This will be a single page application. On the front-end, the proposed technologies embrace the component-based approach with easy extendibility for state management. The solution for this layer should play nice with these.
- The number of users is expected to be around 50K daily. This is not too much, but scalability here is critical – replacing one of the main layers of the architecture in order to expand growing users demand is unacceptable. We need a solution that can handle a large number of concurrent requests.
- Programing language on the front-end is JavaScript/TypeScript. Ideally, if same is used here, that will lead to a lot smaller learning curve, which means, more resources form pure management perspective, easier maintenance and future improvements.
- There is no machine learning involved, so we don’t need to consider a system or a language that has this as an advantage.
- This is a simple API Gateway, which will be used to distribute requests across microservices. For example, an authentication could be encapsulated in a single endpoint in this layer and be exposed to the front-end client. However, behind the scenes, this layer will take care of few calls to different microservices – like: calling Oath to check credentials, then call Redis to get authorized services for that user, then create token for any further transactions. We need to support all this in an easy and scalable way.
- RESTful APIs with an easy option for plugging different middleware will be also a desirable option. Such middleware for example can ensure easy implementation of extra logging or collecting data about users or export data for further analyzing or creating report files.

The perfect fit for all of the above is NodeJS.

ELB (Elastic Load Balancer)

This is not critical for small streaming applications. However, the architecture should provide an empty hook that can be easily pointed to a load balancer. If user demand grows and such ELB becomes a need, following needs to happen: this balancer should receive 2 main inputs – the device type, which is initially send from the client-side and the authorization object, which is extracted form user’s record (Redis). These will be used from the ELB to decide which set of microservices and data stores user should be redirected to – in case of 2 data storage centers (one in Nord America and another one in Europe), the closer or less loaded one should be assigned to that user. In case of particular streaming requirements (for example low vs high resolution viewing device) similar redirection should be considered.

Microservices

This layer contains all microservices needed for the API Gateway to operate. The separation of concern and the scalability here is critical. These should be open for extensions and closed for modifications. This will ensure high level of backwards compatibility – existing APIs triggered from the API Gateway can rely on existing and stable microservices. In the same time, new APIs can connect to other set of microservices and profit form new functionalities and updates. All this should be controlled by clean interfaces, where both sides should send and receive always the bare minimum of data needed for a successful operation. This is also known as Interface segregation.

Data sources

Few types of data sources must be considered.

User data needed for authentication and billing.

This is critical and highly sensitive data. Storing this should be done in a separate database instance/cluster that will be highly monitored for unauthorized access as well as regularly checked for outdated 3rd party software that is marked as vulnerable at OWASP - https://owasp.org/ . Such updates should be scheduled immediately. All data on this server should be encrypted and a copy of it should be maintained on entirely differed data storage system to defeat ransomware attacks. The nature of this data is more or less static and its schema changes are minimal and rare. Any RDBMS could be used, but better fit will be a quick “key-value” caching solution, like Redis.

User data for authorization

What kind of services user is allowed to use? For example, streaming in HD, 4K or 8K? Number of devices allowed or what exactly user can do, like uploading or streaming time limits, options to add reviews, mark favorites, create watch lists, get AI proposed content, etc… The nature of this data is very dynamic. When the platform grows, the need of collecting and analyzing more data will also grow. Data schemas here need to be highly customizable and dynamic. Presenting them in a form of JSON objects will open the possibility for easy handling on NodeJS (API Gateway layer) or transferring them into both client or administrative solutions which will visualize them, create reports, generate suggestion, etc. No SQL, document-based DB here will be a good choice – for example MongoDB or Casandra.

The actual data – the video files

This requires a simple high-capacity storage. Databases, regardless of their type are not appropriate for such needs. Pure cloud storage solution should be used here. The important aspects when choosing, should following:

- Reliability – This is not critical, neither has a security risk. However, service interruption might lead to clients' frustration, followed by un-subscriptions.
- Access frequency and data transfer. Depending on the daily usage either "pay per view" or "pay unlimited" approach should be considered. This decision might have significant financial impact on the platform profitability. An option to change this "on the fly", based on regularly collected usage statistics should be considered.

Disclaimer

Deployment layer is only briefly shown in the diagram. IDE tools and version control systems are also not part of this scope.

Highly specific needs, like particular streaming services for real time video processing and transferring are not considered in this document. Supported audio/video formats, data compression and noise-reduction encoding are also not part of this scope.