Avid5 Software Design

# Introduction

Avid5 runs on a Media PC. It:

* plays many different forms of audio and visual content using the J River Media Center (JRMC) player,
* streams music from Spotify, and
* streams video content through Roku or Chromecast devices.

Output is via an HDMI-connected screen and a multi-zone AV receiver.

The component hardware and player software applications, together with necessary audio/visual hardware, are all controlled and coordinated by the Avid5 software, which runs under Windows (10 or 11) or Linux KDE. The source for this is organized as a single Visual Studio web application project.

This document describes the architecture and design of that bespoke software. Note that all the Avid software described here is freely available as open source under the MIT License.

The description in this document is at a relatively high level, showing how each part fits in the context of the complete system and how each relates to other parts. Detailed descriptions as to how each part works can be obtained from the source code which is commented.

The Avid5 software module is a single web application **Avid5.Net**. This presents the Avid UI to any connected touch-capable browser (e.g. phone or tablet), accessing all the devices and players via its “middleware” classes.

In addition, there is a publicly hosted web application **Avid5.Auth**, which runs somewhere on the public internet to provide the necessary authentication handshake for Spotify, and to hold the necessary secret key to confirm the identity of the accessing software.

All software modules use managed code with the .Net 6 execution environment.

The Avid5.Net web application follows the Model-View-Controller (MVC) structure. There are no explicitly implemented models. Instead the model presented is handled by a set of middleware classes each of which interfaces with a particular player application or hardware component.

The software is designed with the expectation that there are multiple distinct player applications, each handling different content formats. And this was the case in earlier implementations:

* TV viewing and recording was handled by **DVBViewer**
* Video playback (including of recorded TV) was handled by **Zoom Player**
* Music and Photos were handled by **JRMC**

In earlier implementations, this was following a “best of breed” strategy, where no one application could handle everything well. Nowadays, JRMC can be (and is) used as the sole media player, though aspects of the internal design will allow for multiple player applications.

Each set of related views for a running application is implemented as a single MVC controller. Each controller has a set of web views. These sets of views all have two variants: wide and narrow. The wide views are intended for larger screen devices, such as tablets. The narrow views are intended for smaller screen devices such as smart phones. Buttons which navigate between views are constrained to link within the same variant: wide to wide and narrow to narrow. This preserves the overall user experience appropriate for each device. However, the software will automatically change between wide and narrow views as necessary (e.g. when rotating a screen).

The views are arranged as panes, each of which is independently scrollable. The narrow views normally have only a single pane. The wide views can have multiple views, often three. In a wide view, library browser type panes are generally positioned at the left of the view, and player transport buttons are generally positioned at the bottom right. This layout tends to suit a right-handed user best.

Each controller has an associated JavaScript file included in each of its views. That file implements the response of the UI elements (lists, buttons etc) to user interactions (tap, hold, drag, swipe etc). This response could be one or more of:

* Acting purely within the UI with no effect on Avid (e.g. scrolling).
* Invoking some Avid action through an AJAX request to a Controller URL (e.g. pausing a player).
* Replacing the contents of a single pane (partial view) within the view displayed in the browser (e.g. browsing the albums of a selected artist).
* Totally replacing the complete view within the browser, probably with a complete change of the JavaScript (e.g. launching a totally new player).

## MVC Controllers and Views

### Home and Top Bar

In the narrow variant, the Home view allows selection of a player application. It contains buttons for “TV”, “Radio”, “Video”, “Music”, “Spotify”, “Photos”, “Stream”, “Guide” and “Security”. Each of these launches an appropriate player and switches to the default view for the appropriate controller.

In the wide variant, the Home view is blank.

All views, including the home views, contain a “top bar” partial view of a fixed set of controls that appear at the top of all views. The colour of this top bar is styled to indicate the current player application.

These controls include:

* A “Settings” button for special actions and non-default behaviour.
* A Volume display
* Volume buttons (increase, decrease, and mute toggle)
* A Home/Back/Off button, which either displays the Home view or (when on the Home view) turns off all player applications.

In addition, in the wide variant, the top bar contains the player application buttons, which are then accessible from all views without needing to access Home first.

The top bar (and therefore every view) has a JavaScript handler function (SwitchPanelAfterWake) that determines if it is being viewed for the first time in the last minute. This case will occur when (for example) a new controlling device is awoken from its sleeping state. When this case occurs, the view is automatically switched to a suitable default view for the currently running player application, as the view displayed when the device last updated (i.e. before sleeping) may no longer be appropriate.

The “Settings” button available on all screens displays an overlaid “menu” of less-needed actions:

* Screen off
* Screen on
* Sound on TV
* Sound in rooms
* Change view
* Load new media
* Spotify login
* Restart Avid
* Reboot Systems

Turning the screen on and off is normally handled automatically according to the selected player. But it can be controlled manually as well.

The routing of the output sound is normally handled automatically to suit the player application. If the TV screen is used, the sound uses the TV speakers. When no screen is used (e.g. Radio, Music and Spotify), the higher quality music speakers around multiple rooms are used instead. This default choice can also be overridden manually. And for the 5.1 speakers, there is a choice of audio treatment.

Avid’s media database must be manually reloaded when new music or photos are added externally, although this also happens automatically at 03:00 every morning.

For access to Spotify, credentials for a register use are normally required once. Thereafter the user’s identity is use for subsequent browsing and playing.

The Avid software can always be manually restarted in the event of unexpected failure. This is rarely needed. Even more rare is the need to forcibly reboot the AV receiver and media PC.

The rendering of the top bar checks if the client IP address is on the local LAN, and if not will redirect to a “Go Away” page (/Home/GoAway). The Guide (EPG) views also detect if the client IP address is on the local LAN, and if not will **omit** the top bar (and its “GoAway” check), thus allowing remote recording, but disallowing any further navigation.

### Action

The ActionController implements web methods for actions which are not interacting with any particular player application. These methods include those for launching and existing player applications, controlling volume, and the actions of the common top bar Tools menu. The ActionController URLS are:

* /Action/GetRunning [AJAX query]
* /Action/VolumeUp [Action]
* /Action/VolumeDown [Action]
* /Action/VolumeMute [Action]
* /Action/VolumeGet [AJAX query]
* /Action/Launch [Action]
* /Action/StartSky [Action]
* /Action/AllOff [Action]
* /Action/ScreenOff [Action]
* /Action/ScreenOn [Action]
* /Action/StartStream [Action]
* /Action/GoRoku [Action]
* /Action/GoChromecast [Action]
* /Action/GoChromecastAudio [Action]
* /Action/GoSmart [Action]
* /Action/GoPC [Action]
* /Action/SoundTV [Action]
* /Action/SoundRooms [Action]
* /Action/RebuildMediaDb [Action]
* /Action/RecycleApp [Action]

### TV

The TvController implements views and action methods for live terrestrial TV and Radio played through the J River Media Center player application. These URLS are generally used as actions by UI functions in tv.js. The TvController URLS are:

* /Tv/Watch [Narrow View]
* /Tv/ControlPane [Updatable Pane]
* /Tv/Channels [Narrow View]
* /Tv/ChannelsPane [Updatable Pane]
* /Tv/Radio [Narrow View]
* /Tv/RadioPane [Updatable Pane]
* /Tv/NowAndNext [AJAX query]
* /Tv/ChangeChannel [Action]
* /Tv/Action [Action]
* /Tv/Buttons [Narrow View]
* /Tv/All [Wide View]

### Video

The VideoController implements views and action methods for recorded TV and ripped DVDs played through the J River Media Center player application. These URLS are generally used as actions by UI functions in video.js. The VideoController URLS are:

* /Video/Watch [Narrow View]
* /Video/WatchPane [Updatable Pane]
* /Video/All [Wide View]
* /Video/Recordings [Narrow View]
* /Video/Recording [Narrow View]
* /Video/DVDs [Narrow View]
* /Video/RecordingsPane [Updatable Pane]
* /Video/DVDsPane [Updatable Pane]
* /Video/PlayRecording [Action]
* /Video/DeleteRecording [Action]
* /Video/PlayDvdDisk [Action]
* /Video/PlayDvdDirectory [Action]
* /Video/PlayVideoFile [Action]
* /Video/GetPlayingInfo [AJAX query]
* /Video/SendMCWS [Action]

### Music

The MusicController implements views and action methods for stored music and music streamed from BBC iPlayer Radio played through the J River Media Center player application. These URLS are generally used as actions by UI functions in music.js. The MusicController URLs are:

* /Music/All [Wide View]
* /Music/Playing [Narrow View]
* /Music/Queue [Narrow View]
* /Music/QueuePane [Updatable Pane]
* /Music/Browser [Narrow View]
* /Music/BrowserPane [Updatable Pane]
* /Music/GetPlayingInfo [AJAX query]
* /Music/SendMCWS [Action]
* /Music/RemoveQueuedTrack [Action]
* /Music/GetAlbumImage [AJAX query]

### Photos

The PhotosController implements views and action methods for stored photographs viewed through J River Media Center player application. These URLS are generally used as actions by UI functions in photos.js. The PhotosController URLS are:

* /Photos/Display [Narrow View]
* /Photos/Browse [Narrow View]
* /Photos/Images [Narrow View]
* /Photos/ImagesPane [Updatable Pane]
* /Photos/All [Wide View]
* /Music/GetThumbnail [AJAX query]

### Spotify

The SpotifyController implements views and action methods for streamed music played through the Avid.Spotify player application. These URLS are generally used as actions by UI functions in spotify.js. The SpotifyController URLS are:

* /Spotify/All [Wide View]
* /Spotify/Playing [Narrow View]
* /Spotify/Queue [Narrow View]
* /Spotify/QueuePane [Updatable Pane]
* /Spotify/Browser [Narrow View]
* /Spotify/BrowserPane [Updatable Pane]
* /Spotify/GetPlayingInfo [AJAX query]
* /Spotify/PlayAlbum [Action]
* /Spotify/PlayTrack [Action]
* /Spotify/PlayPlaylist [Action]
* /Spotify/SkipToQueuedTrack [Action]
* /Spotify/RemoveQueuedTrack [Action]
* /Spotify/PlayPause [Action]
* /Spotify/Skip [Action]
* /Spotify/Back [Action]
* /Spotify/Plus10 [Action]
* /Spotify/Minus10 [Action]
* /Spotify/SetPosition [Action]
* /Spotify/GetAlbumImage [AJAX query]
* /Spotify/AddTrackToPlayList [Action]
* /Spotify/AddAlbumToPlayList [Action]
* /Spotify/RemoveTrackFromPlayList [Action]
* /Spotify/RemoveAlbumFromPlayList [Action]

### Streaming

The StreamingController implements views and action methods for streaming devices and services. These devices include a Roku 3 box and a Chromecast dongle both plugged into the Yamaha Receiver as distinct HDMI inputs. These URLS are generally used as actions by UI functions in streaming.js. The StreamingController URLS are:

* /Streaming/Browser [Narrow View]
* /Streaming/Controls [Narrow View]
* /Streaming/All [Wide View]
* /Streaming/RokuLaunch [Action]
* /Streaming/RokuGetPlayingInfo [AJAX query]
* /Streaming/KeyDown [Action]
* /Streaming/KeyUp [Action]
* /Streaming/KeyPress [Action]
* /Streaming/SendText [Action]

The Roku 4 box offers many streaming services accessible through the Avid UI, including Sky’s Now TV, BBC iPlayer etc.

### Guide

The GuideController implements views and action methods for viewing the Electronic Programme Guide (EPG) and setting recording for terrestrial TV and Radio programmed though the JRMC Recording service and its MCWS API. These URLS are generally used as actions by UI functions in guide.js. The GuideController URLS are:

* /Guide/Browser [Narrow View]
* /Guide/BrowserWide [Wide View]
* /Guide/BrowserPane [Updatable Pane]
* /Guide/ListingsPane [Updatable Pane]
* /Guide/Description [AJAX query]
* /Guide/Record [Action]
* /Guide/Cancel [Action]
* /Guide/RecordSeries [Action]
* /Guide/CancelSeries [Action]

The Guide (EPG) views detect if the client IP address is on the local LAN, and if not will **omit** the top bar, this allowing remote recoding, but disallowing any further navigation.

## Middleware Classes

The “Model” aspect of the MVC pattern is provided by a set of “middleware classes”, each of which is designed to offer a useful API into a single player application, external service or hardware component. The controller and view classes implement the necessary UI views and actions though these middleware classes.

### JRMC

The JRMC class encapsulates those uses of the J River Media Center player application for cataloguing and playing all stored music and for viewing photos.

The main interaction with the JRMC running as a service is its “MCWS” (media center web service) service running as an HTTP web service on port 52119 on the local host. This web service API is fully documented within the JRMC software.

While that API is very rich, its very generality does not efficiently handle the access patterns used within Avid.

The structuring of the Avid music catalogue splits music into “classical” and “non-classical”. Classical music is primarily indexed by composer, and then within composers by album. Non-classical music has two primary indexes – by “album artist” and by album name. When indexed by “album artist”, music is further keyed by album name. Both the “album artist” and “album name” primary indexes are also indexed by the initial letter. Finally the most expected style of music playing is that generally complete albums are played as a whole.

To support this indexing pattern, a custom structure is maintained by the JRMC class (of classes TrackData, AlbumData and AlbumCollection). This internal structure is constructed as an object hierarchy though a tree-walk of the J River data through the MCWS interface. But as the cost of constructing the object structure is high (tens of seconds), the complete structure is serialized out to an XML cache file. On startup, this file is loaded in preference to the tree-walk – a sub-second activity. When the J River data changes (e.g. when a new album is purchased), the LoadAndIndexAllAlbums method rebuilds the binary cache. This is run either manually or automatically at 03:00 each morning.

The structure and purposes of the JRMC class and its members should be fairly clear from the commenting. However, there a few aspects worth noting:

* In addition to the AlbumList (and related collections) there is a PhotoAlbumList used by the PhotoController and its views. This is constructed from MCWS in the same manner.
* There are some “hard-wired” queries that suit the album-oriented usage. These include GetLuckyDipAlbums and GetRecentAlbums, both of which return data on twenty albums.
* An album is determined as “classical” if its path contains the directory “Classical”. This determination then imposes no constraints on the genre or other meta-data tags.
* JRMC has various screen modes and there are methods to set the appropriate mode for the running player application.

### VideoTV

The Video class encapsulates those uses of the J River Media Center player application for watching and recording live TV, watching recorded terrestrial TV programmes, ripped and physical DVDs and other videos.

The class has contained classes:

* Channel – a TV or Radio channel
* Recording – a recorded TV programme stored in a file
* Programme – a scheduled programme in the EPG
* Timer – a scheduled event to record a programme when broadcast
* Series – a rule to determine whether to record future programmes broadcast on the same channel with matching titles at or about the same time on the same day of the week

### Spotify

The Spotify middleware class is a .Net class wrapper around the Avid.Spotify WebAPI methods. These are organized in six groups:

* Browsing and Searching  
  These navigate the Spotify data model and hierarchy
* Playlists and My Music  
  These access the saved albums and playlists for the authenticated user
* Player Queue Management
* Player currently playing track operations  
  The actual audio player is within the Yamaha AVR, though all control is via this current class
* Constructors of SpotifyData classes from Web API model  
  These are internal mechanisms
* Authentication  
  Interacts with the authentication service to login a specific Spotify user. This is generallu only needed once and the credentials are securely preserved and re-used using OAUTH2 mechanisms

### Roku

The Roku class accesses a Roku streaming device through its web API, emulating keystrokes issued on its remote control, and listing and launching Roku “Apps”.

### Running

The Running middleware class is responsible for launching and exiting player applications, for ensuring that two players are not running at the same time, and for each application ensuring that the screen is turned on or off as appropriate, and that the sound is routed to the correct set of speakers by default. It keeps track of the currently running player application.

### Receiver

The Receiver middleware class provides methods for querying and controlling a Yamaha AV receiver. The only models with which this has been used are the RX-V671 and RX-V6A, but I strongly suspect that it will work with other models in the same range. The significant aspect of the Yamaha receiver is that it can be controlled via an easy to understand HTTP protocol.

The receiver (at least those particular models) has two separate zones. One can route digital inputs (HDMI or SPDIF) to a 5.1 speaker array. The other zone can use the same input and can route inputs to stereo speakers and auxiliary stereo amplifiers in other rooms.

The Receiver middleware class has methods to:

* Turn on and off the two “output zones” (i.e. two sets of speakers) independently. Normally only one zone will operate at any one time.
* Control the volume and muting of the “current” zone.
* Specify the input source, which can be the PC (and its JRMC players), a Roku streaming device, a Chromecast streaming device, or the audio from the smart TV itself.

### Screen

The display screen for Avid is a high quality LG 4K Smart TV, connected as the HDMI output of the Receiver. However, most of the capability of this TV is wasted as it is simply used as a 3840x2160 output screen for any media PC player application (or streaming device) that needs a visual display. None of its TV tuner, audio or “smart” networked capability is normally used (though this can be routed as a “Stream” option).

The only control for the screen is to be able to turn it on or off.

Unfortunately, the LG cannot be controlled by its network interface. The network control service running on the TV firmware is encrypted, and restricted to the proprietary remote app.

The TV can be turned on with Wake-on-Lan. The power status can be determined by an IP “echo” packet. But no network mechanism will turn the TV off.

The power can be controlled via the HDMI-CEC protocol, using only two CEC commands – one to turn on the screen, and one to turn it off. The HDMI-CEC codes can be generated by an external [Pulse-Eight USB-CEC Adapter](https://www.pulse-eight.com/p/104/usb-hdmi-cec-adapter) accessed through the “cec-client” command-line program.

### Config

The Config middleware class reads a single XML file from the Avid5Config.xml file in the directory specified in the invocation command-line parameter. The values in that XML files can be used to configure paths, IP addresses etc. The properties of the Config class are documented in the “Avid5 Installation and Configuration” document.

### Render

The Render middleware class is used as a utility class in MVC views to construct the HTML <script> tag with the URL of a JavaScript file, such that the file is cached only until the source file changes. This copes with the development cycle when the JavaScript file may change at any time, while still benefiting from the caching of an unchanged file in normal operation. It works by appending an unused argument to the generated script URL whose value is the timestamp of the real source file for the JavaScript, so that changes in the source file result in different (separately cached) URLs.

### IpAddress

The IpAddress middleware class is utility class to determine if the client IP address is on the local LAN and therefore to allow full access. External clients are restricted to using the EPG only for remote recording. For simplicity, this determination is based on a textual representation of IPV4 or IPV6 addresses. In the future, this could perhaps be determined with more of an understanding of the address formats.

### Button

The Button class abstracts the display of on-screen buttons. In the past, these were PNG images of various shapes. Nowadays the buttons are styled in CSS. Button contents can be either text or icons. The icons used are selected from Google’s “Material” symbols accessed remotely. The content of the button is the text specified, but with embedded Material symbol names surrounded by double square brackets.

### TP\_Link

The TP\_Link class encapsulates the network access to “TP-Link” brand smart light bulbs and switches. This is used by the Security class to manage scheduled and random changes of lighting for home security.

The only public methods are “TurnOn()” and “TurnOff()”. Devices are identified via their IP network address.

### Security

The Security class is configured by the file “Security.xml” and manages scheduled and random changes of lighting for home security. An example Security.xml is provided.

Individual smart switches and lights can be identified and controlled manually.

Several “Profiles” can be configured to specify the probability that each light or switch will be on or off in any period of time. The actual switching is random within these profile rules. And the UI allows the selection of the current profile.

Additionally, the current profile can be set through external HTTP access, and this is used by a Raspberry Pi computer to select the profile based on a key pressed on a simple numeric keypad.

## User Interface

The user interface for controlling Avid is purely via HTML in a web browser on a touch device (smartphone or tablet). The HTML is generated as MVC views using the “Razor” view engine notation, which combines C# code and HTML in an easily understood manner. The C# code accesses data to display from the appropriate middleware classes.

The views come in two variants: wide and narrow. The wide views are intended for larger screen devices, such as tablets. The narrow views are intended for smaller screen devices such as smartphones. Buttons which navigate between views are constrained to link within the same variant: wide to wide and narrow to narrow. This preserves the overall user experience appropriate for each device.

The variant views are then assembled from included “pane” partial views. A narrow view will normally have a set of buttons to switch panes and then display only a single selected pane. A wide view will normally display all the panes appropriate for the current player application – usually three. By convention, in a wide view:

* The complete left-hand-side is a “browsing” or “selecting” list of things to play.
* The lower right-hand-side contains the play controls.
* And the upper right-hand-side contains either a list of things playing or some additional controls.

The top of every view is the common “top bar” discussed earlier – identical (apart from colour) in every view for every player application.

The Razor view engine generates “pure” HTML with no embedded styling or JavaScript. Objects in the HTML all have class and id names as necessary to label the content for CSS styling or handling touch actions. A single CSS file handles all the styling of all views in a consistent manner. Each player application has a single JavaScript script file included to handle the user interaction behaviour for that player. In addition, a common JavaScript file (Functions.js) contains some functions and behaviours common to multiple players.

The JavaScript which implements the user interactions does **not** use a touch-centric platform, such as JQuery.Touch or JQuery.Mobile. Experience has shown that these are nowhere near responsive enough for the requirements of Avid.

The JavaScript, does however make extensive use of JQuery, which significantly enhances the maintainability of the script code. All the touch handling (touch, tap, doubletap, hold, drag, swipe, pinch) is handled by Hammer.js (<http://eightmedia.github.io/hammer.js/>). Each pane has a Hammer object with all necessary actions bound to appropriate buttons or list items. This has proven very effective in constructing a consistent set of responsive behaviours.

In addition, Functions.js implements two common touch behaviours:

* EnableDragScroll, when added to a Hammer object for a pane of list items (e.g. album titles), makes that list touch-scrollable within a fixed area of the browser.
* EnableMouseBehaviour, when added to a Hammer object for a blank rectangular area, makes that area act like a mouse touch-pad, sending MouseMove and MouseClick actions as the area is touched, dragged and tapped.

Additional third-party JavaScript used are:

* noUiSlider (<http://refreshless.com/nouislider/>) – a lightweight and minimal on-screen slider control.
* Jquery.easing (<http://gsgd.co.uk/sandbox/jquery/easing/>) – which adds attractive deceleration to the touch drag and swipe behaviours.

## Free and Open Source Software

### JQuery

The JavaScript code for the UI makes extensive use of the JQuery library (<http://jquery.com/>). It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.

### Hammer.js

All the touch handling (touch, tap, doubletap, hold, drag, swipe, pinch) is handled by Hammer.js version 1.x (<http://eightmedia.github.io/hammer.js/>). This lightweight and efficient JavaScript adds a rich touch capability of a browser-based UI, without the complexity and significant performance costs of adopting a library such as JQuery Mobile.

* Note that Hammer.js version 2.x is very different and not used.

### noUiSlider

The sliders used for progress bars etc are implemented using noUiSlider (<http://refreshless.com/nouislider/>) – a lightweight and minimal on-screen slider control. This takes the form of a JavaScript library for the slider behaviour and a CSS style sheet.

### SpotifyAPI-Net

The Spotify Web API can most easily be accessed from .Net code by a wrapper library. The use used in Avid5 within the Spotify middleware class is SpotifyAPI-NET found at <https://github.com/JohnnyCrazy/SpotifyAPI-NET>, developed under the LGPL open source licence.

### NLog

NLog (<http://nlog-project.org/>) is a free open source logging platform for .NET and other platforms. It is used in Avid5.Net, whose nlog.config file configures logging to files in the sub-directory “Logs” within the web application directory. A new log file is started each day, named by the date.