ustudio Player Themes

Designing a Theme

Themes in the uStudio Player Framework are simply HTML, CSS and asset files, much like a normal website, and can be developed using existing web development tools.

The primary difference is that, instead of providing the entire HTML document, the theme provides a document fragment, which is inserted into the player iFrame when the player loads.

The Player Framework also uses data-binding to display video-specific content in the player, and to enable user interaction when theme elements are clicked.

Theme File Structure

Themes are uploaded to uStudio as a zip file containing two files and a folder:

- `template.html` should contain the document fragment with all the HTML for rendering the player theme.
- `main.css` should contain all the CSS for styling the player theme.
- `static/` should be a folder containing any static assets needed by the theme (images, etc.).

Each of these is optional: a theme containing only a CSS file will use the default HTML, but be styled with the provided CSS, a theme defining only an HTML file will be styled with the default CSS, and a static folder is not necessary if the theme has no static assets.

Important Note:

The files in a custom theme must appear at the root of the zip file, not in a sub-folder. If your custom theme does not appear to be working, please check the file structure of the custom theme zip file.

The structure of the zip file should look like this:

- `theme.zip`
- `template.html`
- `main.css`
- `static/`
    - `logo.jpg`

Player Element

In order for the Player Framework to determine where to render the video, there must be a single element with the attribute `data-player-container="player"`.

The simplest possible player theme would be:

```
html <div data-player-container="player"></div>
```

The actual video element (or Flash fallback element) will be inserted into the page as a child of that element, and will fill up the entire width and height of the container element.

Like all HTML elements, the player element will cover up any elements that appear before it in the HTML, so player controls should appear in the HTML after the player container.

Data binding

Note: In a future version of the Player Framework, the data binding attributes will be restructured and renamed to be more consistent and flexible (especially with custom video metadata). Developers will be notified and there will be a period of deprecation during which themes can be upgraded.

Data binding is used to display information about the current video inside the theme. This works by adding an attribute specifying the name of the video property you want to display, and the Player Framework will set the text content of the element to that value.

- Title: The video title can be displayed by adding the attribute `data-player-video-attribute="name"`.
- Duration: The duration, as a time-code, can be displayed by adding the attribute `data-player-text="duration"`.
- Current Time: The time-code of the viewer's current position in the video can be displayed with `data-player-text="current-time"`.

SEEK BAR

The seek bar is updated by setting the width of an element to be a percentage relative to the user's current position in the video. If they are half-way through the video, the element's width will be set to 50%, etc.

To define which element has its width set, use the attribute `data-player-width="progress"`.

You can also define an element which is updated as the video is buffered, using the attribute `data-player-width="buffer"`.

A simple progress bar implementation might look like:

```
html <div> <div data-player-width="buffer"></div> <div data-player-width="progress"></div> </div>
```

VOLUME BAR

The volume bar is managed similarly to the seek bar. The Player Framework will update the width of any element with the attribute `data-player-width="volume"` when the volume changes.

POSTER IMAGE

The poster image for the video can be displayed by adding the attribute

`data-player-container="poster"`. An `<img>` tag will be added as a child with the source set to the poster image for the video. The image will fill the entire container element, while maintaining the correct aspect ratio for the original image.

See Player States for information on only showing the poster image before the user has started the video playing.

Player States

Most players will want to display different elements at different points in time (before the user has clicked play, while playing, etc.). The Player Framework supports optionally displaying elements using a feature called player states.

To enable an element to be displayed in one or more states, use the `data-player-states` attribute, listing each state in which the element should be displayed, separated by commas.

For example: `data-player-states="state1,state2"` would cause the element to be displayed while the player is in `state1` or `state2`, and hidden in all other states.

The available states are:

- `waiting` is the state the player is in before the user plays the video for the first time, and after the video finishes playing. It is the initial state of the player when autoplay is disabled.
- `playing` is the state the player is in while the video is playing and the user is interacting with the player. It is the initial state of the player when autoplay is enabled.
- `paused` is the state the player is in while the video is paused and the user is interacting with the player.
- `idle` is the state the player is in while the video is playing or paused, after the user has stopped interacting with the player for a period of time.
- `loading` is the state the player is in while the video is loading or

buffering.

- `swapping` is the state the player is in while the video is being swapped (i.e. transitioning to or from HD mode).
- `error` is the state the player is in after the browser or framework reports an error.

Note: We are continuing to review additional states, and may add more in the future. For example, we might add an explicit `ended` state, as opposed to re-entering the `waiting` state, so that the player theme does not have to implement custom JavaScript to show a different end state versus preview state.

We invite developers to contact us with any other states which would be useful for custom themes or player interactions.

STATE TRANSITIONS

Animations can be added to state transitions using the `data-player-state-transition` attribute. The available transitions are:

- `fade` The element is faded in or out.
- `none` The element is shown or hidden immediately (default, if the attribute is not present).

User Interaction

Most themes will need to allow the user to interact with the player, playing, pausing and controlling the video. An element can be tied to one or more player actions using the attribute `data-player-actions`.

The value of the attribute should be a comma-separated list of events and the actions they trigger, of the form `event=action`.

To play the video when a user clicks an element, for example, add the attribute: `data-player-actions="click=play"`. The event name can be any DOM event, and the action can be one of the following:

- `play`

- `pause`
- `mute`
- `unmute`
- `play-pause-toggle`
- `mute-unmute-toggle`
- `hq-toggle`
- `captions-toggle`
- `fullscreen-toggle`
- `playlist-next`
- `playlist-previous`
- `playlist-select-video` (see Playlist Players for details)

TOUCH ENABLED DEVICES

On touch enabled devices, handling UI interaction, like mouse hover versus tapping a button, in CSS can involve several steps. We have simplified the process with an optional helper data attribute: `data-player-touch-classes`.

The attribute accepts a comma separated list of two classes: the first will be added to the element when the player is being viewed on a device which supports touch events, and the second is added when the device does not support touch events.

For example, to add the class `touch-enabled` to an element on touch-enabled devices, or `touch-disabled` on devices without touch support, add the following attribute to the element:

```
html data-player-touch-classes="touch-enabled,touch-disabled"
```

CONTROL STATE ATTRIBUTES

For elements bound to a "toggle" action, the Player Framework will set data attributes to indicate the current state of the parameter being controlled. Themes can use these attributes in their CSS to indicate to

the user the toggled state of the control.

The data attributes, and their values, are as follows:

- `data-player-play-state`: `playing`, `paused`
- `data-player-mute-state`: `muted`, `unmuted`
- `data-player-hq-state`: `best-fit`, `hq`
- `data-player-captions-state`: `hidden`, `showing`
- `data-player-fullscreen-state`: `fullscreen`, `normal`

For example, to change the style of a play/pause toggle button from CSS, use the following selectors:

```css [data-player-play-state=playing] { // styles when the video is playing }

[data-player-play-state=paused] { // styles when the video is paused }
```

SLIDER INTERACTIONS

Themes can define sliders used for controlling various parameters, usually in conjunction with a progress bar. To define a slider, create an element which will contain the entire slider, including the "handle" element, and add the `data-player-slider` attribute, with a value of `progress` or `volume`, depending on the parameter to control.

The Player Framework uses the jQuery UI Slider widget to provide slider functionality.

Playlist Players

Playlist Players are supported by the Player Framework, and a single theme can support both playlist and non playlist players. Elements with the attribute `data-player-playlist="true"` will only be visible in a playlist player with more than 1 video in the playlist.

PLAYLIST ITEM ELEMENTS

The Player Framework provides a facility for repeating an element once, for each video in the playlist. This can be used to provide UI for jumping to a specific video, in addition to the next/previous controls provided by the player actions (see User Interaction).

Elements with the attribute `data-player-repeat-per="video"` will be repeated once, for each video in the playlist. (The initial element will be removed, and new elements will be added as children of the same element).

For example, consider the following snippet in a playlist with 5 videos:

```
html <ul> <li data-player-repeat-per="video" data-player-video-attribute="name">
</li> </ul>
```

When the player initializes, the `<li>` element will be removed, and then 5 copies (one per video) will be inserted into the `<ul>` element. Each element's text will be set to the name of the corresponding video, because of the `data-player-video-attribute`.

An attribute of `data-player-repeat-width="percentage"` can be used to give the elements a width proportional to the number of videos in the playlist (for 5 videos, each element will be given a width of 20%).

VIDEO SELECTION

In order to allow the user to select the corresponding video by clicking on the element, several other attributes are required. When the playlist item elements are repeated, the Player Framework gives each one an `id`, prefixed by a user-controllable prefix, specified by `data-player-repeat-id-prefix`.

Given an element:

```
html <li data-player-repeat-per="video" data-player-repeat-id-prefix="video-id-">
</li>
```

The resulting element would be inserted, for a video with the ID

`V987abcDEF`:

```
html <li id="video-id-V987abcDEF" data-player-repeat-per="video" data-player-repeat-id-prefix="video-id-"></li>
```

When the element is bound to the `playlist-select-video` action (See User Interaction), this attribute and the element's `id` attribute are used to look-up the corresponding video and select it from the playlist.

The `data-player-selected-video-class` attribute can be used to specify a class to add to the element when the corresponding video is selected (and removed when another video is selected). This class can then be used in the theme's CSS to highlight the element for the currently selected video.

Managing Themes

Themes can be uploaded and managed through the uStudio API. See the API Overview for more information on accessing the API.

Themes are stored on a per-Studio basis and can be enabled on individual Destinations within that Studio. uStudio can provide command-line tools for packaging and managing themes.

Uploading a Theme

First, a theme must be zipped up following the format specified in the Theme File Structure. Next, a new theme resource must be created in the studio, by making a POST to the following URL, with the name of the theme encoded in JSON:

``` /api/v2/studios/[studio_uid]/player_themes

{ "name": "[theme_name]" } ```

This request will return JSON describing the newly created theme, including a `uid` and a URL to upload the theme zip file:

```
{ "uid": "[theme_uid]", "name": "[theme_name]", "url": "[theme_url]", "studio_url": "[studio_url]", "upload_url": "[upload_url]" }
```
```

The zip file should then be uploaded to the url specified in the `upload_url` member.

Adding a Theme to a Destination

A theme can be enabled for a destination by making a POST request to the following URL with the `uid` of the theme (as returned by creating the theme) encoded in JSON:

``` /api/v2/studios/[studio_uid]/destinations/[destination_uid]

{ “player_theme_uid”: “[theme_uid]” } ```

## Maintaining Themes

After being created, themes can be listed, modified and new theme files can be uploaded.

#### LISTING THEMES

The custom themes added to a studio can be listed by performing a GET to the player themes URL of a studio, returning the following JSON:

``` /api/v2/studios/[studio_uid]/player_themes

{ “themes”: [{...}, {...}] } ```

The JSON objects in the `themes` array will be the same as those returned when creating the themes.

MODIFYING A THEME

After a theme has been created, its parameters can be changed by performing a POST to the url specified in the `url` parameter of the theme JSON:

``` /api/v2/studios/[studio_uid]/player_themes/[theme_uid]

{ “name”: “[new name]” } ```
```

New theme files can also be uploaded to a theme by posting a new zip file to the theme's `upload_url`. All players on Destinations using that theme will automatically be updated by the uStudio system.

REMOVING A THEME FROM A DESTINATION

A custom theme can be removed from a destination, resetting it to the default theme, by setting the `player_theme_uid` to an empty string, via a POST to the destination:

``` /api/v2/studios/[studio_uid]/destinations/[destination_uid]

{ “player_theme_uid”: "" } ```
```