Contents

0.1 King Kiosk User Guide

Your Complete Guide to the Most Powerful Digital Signage Platform

Version 2.3 | Beta Tester Edition

0.1.1 Table of Contents

0.1.2 1. Welcome to King Kiosk

0.1.2.1 What is King Kiosk?

King Kiosk transforms any screen into a powerful, remotely-controlled digital signage display. Whether you're managing a single tablet on your kitchen wall or coordinating hundreds of displays across multiple locations, King Kiosk gives you complete control from anywhere in the world.

Imagine walking into a room and having your display instantly show your calendar, weather, and security camera feeds. Picture a retail environment where product information updates automatically across all displays. Envision a smart home hub that responds to your presence, your voice, and your automation rules.

That's King Kiosk.

0.1.2.2 Why King Kiosk is Different

The digital signage industry is stuck in the past. Most solutions are either too simple (just showing a slideshow) or too complex (requiring dedicated IT teams). King Kiosk breaks that pattern.

Here's what sets us apart:

True Multi-Window Tiling Unlike traditional kiosk software that shows one thing at a time, King Kiosk lets you display multiple widgets simultaneously. Show a clock, weather, calendar, and live camera feed all on the same screen, each in its own resizable tile.

Real-Time Remote Control Through MQTT (a lightweight messaging system - think of it as a walkie-talkie for your devices), you can control every aspect of your displays instantly. Change content, adjust layouts, send notifications, trigger alerts - all in real-time from anywhere.

Home Assistant Native King Kiosk speaks the language of smart homes. It auto-discovers itself to Home Assistant, publishes sensor data, and responds to automations. Your displays become part of your smart home ecosystem.

AI-Powered Voice Control Connect to Home Assistant's conversation agents, Anthropic's Claude, OpenAI, Google Gemini, or even local Ollama instances. Your displays can listen, understand, and respond.

Cross-Platform Consistency The same powerful experience on Android, iOS, macOS, Windows, Linux, and even Apple TV. One interface, one command structure, everywhere.

Secure by Design Optional command signing ensures only authorized commands reach your displays - like a secret handshake that verifies messages are genuine. Perfect for enterprise deployments.

0.1.2.3 Platform Availability

King Kiosk runs on: - **Android** - Phones, tablets, and dedicated kiosk hardware - **iOS** - iPhones and iPads - **macOS** - Mac desktops and laptops - **Windows** - Desktop and kiosk PCs - **Linux** - Including Raspberry Pi - **tvOS** - Apple TV (with remote browser streaming) - **Web** - Browser-based deployment (limited features)

0.1.3 2. Getting Started

0.1.3.1 Initial Setup

When you first launch King Kiosk, you'll find a clean, modern interface ready for customization. Before diving into remote control, take a moment to explore the app:

1. **Access Settings** - Tap or click the lock icon to unlock your display. You'll be prompted to enter a PIN - **the default PIN is 1234**.

2. **Configure MQTT Settings** - Set up your MQTT connection for remote control and **Set Your Device Name** - Give your display a memorable name like “Kitchen Display” or “Lobby Screen 1”
3. **Change Your PIN** - For security, navigate to **Settings -> Security** and change the default PIN to something unique for your deployment

0.1.3.2 Android Kiosk Mode (Lockdown)

On Android, King Kiosk can lock a device so it stays dedicated to the display—great for wall tablets, public screens, and smart-home dashboards.

There are **two practical levels** of kiosk lockdown on Android:

- **Standard kiosk (no Device Owner)**: easy to enable in Settings, but a determined user can often escape depending on the device/OEM.
- **Full kiosk (Device Owner)**: strongest lockdown; King Kiosk can enforce itself as HOME and apply device-owner kiosk policies.

0.1.3.2.1 Standard Setup (No Device Owner) In **Settings -> App Settings -> Kiosk Mode**: 1. Enable kiosk mode and approve the requested permissions (Device Admin / overlays / battery optimization prompts). 2. When prompted, set **King Kiosk** as the **Home/Launcher** app and choose **Always**.

What to expect: the device stays in King Kiosk most of the time, but some devices still allow exit via certain system gestures/shortcuts.

0.1.3.2.2 Full Lockdown (Device Owner Setup) Device Owner setup is intended for **dedicated kiosk devices** and usually requires a **freshly reset** device. This is a more advanced setup typically done by IT administrators.

1. Factory reset the device.
2. Before adding any Google account, connect the device to a computer via USB and run a special command using Android development tools. (If you’re not familiar with this process, consult your IT administrator or search for “Android Device Owner provisioning” tutorials.)
3. Launch King Kiosk and enable **Kiosk Mode**.

What to expect: The strongest lockdown available - users cannot exit the app using normal means. To recover from a misconfiguration, you may need physical access to the device, and in worst cases, a factory reset.

0.1.3.3 iOS/iPadOS Kiosk Mode (Guided Access)

iOS and iPadOS are much more locked down than Android. For “true kiosk mode” you typically use:

- **Guided Access** (recommended for personal devices): enabled by the user in iOS Settings, then started from the device.
- **Single App Mode** (enterprise/education): requires a **supervised** device and device management software (MDM - Mobile Device Management, used by businesses and schools).

In King Kiosk, enabling **Kiosk Mode** on iOS is **best-effort**: - The app can hide system UI and lock orientation. - The app can attempt to start Guided Access if it’s enabled, otherwise it will show instructions.

To enable Guided Access: 1. On the iPhone/iPad, go to **Settings -> Accessibility -> Guided Access** and turn it on. 2. Open King Kiosk. 3. Triple-click the **Side/Home button** and choose **Guided Access**, then tap **Start**.

0.1.3.4 macOS Kiosk Mode (Best-Effort)

macOS does not support a fully locked-down kiosk mode without device management and system policy.

King Kiosk can still provide a solid “kiosk-like” experience on macOS: - Fullscreen + Always on Top - Hide Dock and menu bar (best-effort) - Reduce app switching and some system UI access (best-effort)

For **true kiosks** on macOS, use a **dedicated macOS user account** with parental controls or restrictions, and/or device management software that limits system access.

0.1.3.5 Connecting to Your MQTT Server

MQTT is the communication backbone of King Kiosk - it’s how your displays receive commands and send back status updates. Think of an MQTT server (also called a “broker”) as a central message hub that routes commands to your devices.

You’ll need access to an MQTT server - this could be: - A local server like Mosquitto running on a Raspberry Pi or home computer - Your Home Assistant’s built-in MQTT server - A cloud MQTT service like HiveMQ or CloudMQTT

In Settings, navigate to the MQTT section and enter: - **Server Address**: The IP address or web address of your MQTT server (e.g., 192.168.1.100 or mqtt.example.com) - **Port**: Usually 1883 for standard

connections, 8883 for encrypted connections - **Username & Password**: If your server requires login credentials - **Enable SSL**: For encrypted, secure connections

Once connected, King Kiosk automatically listens for commands and begins sending status updates.

0.1.3.6 Understanding Your Device ID

Every King Kiosk installation has a unique device ID. This ID is used in all MQTT topics to ensure commands reach the right display. Your device ID is based on your configured device name, sanitized for MQTT compatibility.

For example, if you name your device "Living Room Display", your device ID might be `living_room_display` or similar.

You can find your device ID in Settings under the Device Information section.

0.1.3.7 Home Assistant Auto-Discovery

If you use Home Assistant, King Kiosk can automatically appear as a device in your smart home dashboard. When enabled, King Kiosk announces itself using Home Assistant's MQTT Discovery protocol.

To enable auto-discovery: 1. Ensure your Home Assistant is using the same MQTT broker as King Kiosk 2. In King Kiosk Settings, navigate to **MQTT > Home Assistant Discovery** 3. Toggle the feature on

Once connected, your display will appear in Home Assistant with: - **Sensors** for device status, battery level, screen state, and more - **Buttons** for common actions like refresh, screenshot, and restart - **Camera** entity that displays periodic screenshots of your display

This means you can monitor and control your King Kiosk displays directly from the Home Assistant UI, include them in automations, and view their status on your HA dashboards - all without any manual YAML configuration.

0.1.4 3. The MQTT Architecture

0.1.4.1 How King Kiosk Communicates

King Kiosk uses a elegant two-level communication model:

System Commands control the device as a whole - things like volume, brightness, creating windows, taking screenshots, and managing layouts.

Element Commands control individual widgets - like telling a specific clock to switch from analog to digital, or a carousel to advance to the next slide.

This separation means you can manage displays at whatever level of detail you need.

0.1.4.2 Message Addresses (Topics) Explained

Every MQTT message has a "topic" - think of it as the mailing address that tells the system where to deliver the message. King Kiosk uses a consistent, easy-to-follow address structure:

Sending Commands TO King Kiosk:

```
kingkiosk/{your-device-id}/system/cmd          -> System-wide commands
kingkiosk/{your-device-id}/element/{widget-id}/cmd  -> Widget-specific commands
```

Receiving Responses FROM King Kiosk:

```
kingkiosk/{your-device-id}/system/response     -> System command results
kingkiosk/{your-device-id}/element/{widget-id}/response  -> Widget command results
kingkiosk/{your-device-id}/element/{widget-id}/state     -> Widget current state
kingkiosk/{your-device-id}/element/{widget-id}/event     -> Widget events
kingkiosk/{your-device-id}/info                -> Device capabilities
kingkiosk/{your-device-id}/status              -> Online/offline status
```

0.1.4.3 Command and Response Flow

Commands are sent as structured text messages in a format called JSON (JavaScript Object Notation) - basically name-value pairs wrapped in curly braces. Here's the basic format:

```
{
  "command": "command_name",
  "parameter1": "value1",
  "parameter2": "value2"
}
```

For example, to set the volume to 50%:

```
{
  "command": "set_volume",
  "value": 0.5
}
```

King Kiosk responds with a JSON result:

```
{
  "success": true,
  "command": "set_volume",
  "volume": 0.5,
```

```
  "timestamp": "2024-12-19T10:30:00.000Z"
}
```

0.1.4.4 Tracking Your Commands

If you send many commands, responses can arrive out of order. To make it easy to match a response to the command that triggered it, you can include a `correlation_id` field (a tracking number of your choice) in the request. King Kiosk will include that same value in the response.

Why it helps: - Lets your automations match responses to the right command - Helps avoid confusion when multiple devices or automations are active at once - Makes troubleshooting easier because you can search by a single ID

How to use it: - Pick any short, unique ID (for example `ui-42` or `req-2024-12-19-01`) - Include it in your command, and the response will have the same ID - If you don't include a tracking ID, everything still works - you'll just match responses by timing or content

Example with `correlation_id`:

```
{
  "command": "set_volume",
  "value": 0.5,
  "correlation_id": "req-0007"
}
```

0.1.4.5 Secure Command Signing (Optional)

For deployments requiring extra security, King Kiosk supports signed commands. When enabled, commands must include a special code that proves they came from an authorized source - like a digital wax seal that verifies authenticity.

Signed commands wrap your regular command in a protective envelope:

```
{
  "ts": 1703001234,
  "msg": "{\"command\":\"play\"}",
  "sig": "a1b2c3d4e5f6..."
}
```

The verification code ("sig") is calculated using a secret password known only to you and your devices. This prevents unauthorized people from sending commands to your displays.

0.1.5 4. The Windowing System

0.1.5.1 Understanding Tiles and Windows

King Kiosk's windowing system is what truly sets it apart. Instead of a single full-screen view, you can create multiple "tiles" - each containing a different widget.

Window Properties: - **Position (x, y)** - Where on screen the window appears - **Size (width, height)** - How large the window is - **Movement Animation (`animate`)** - MQTT move/update commands animate by default; set `"animate": false` to snap instantly - **Opacity** - Transparency level (0.0 = invisible, 1.0 = fully visible) - **Stacking Order (Z-Index)** - Which window appears on top when they overlap (higher numbers appear on top, like stacking papers) - **Background Mode** - Many widgets support transparent backgrounds

Setting Opacity When Creating Windows:

Most widget creation commands accept an `opacity` parameter:

```
{
  "command": "open_clock",
  "window_id": "my-clock",
  "opacity": 0.8,
  "x": 100,
  "y": 100
}
```

Many widgets also support transparent backgrounds with `background_mode` and `background_opacity`:

```
{
  "command": "open_clock",
  "window_id": "transparent-clock",
  "background_mode": "transparent",
  "background_opacity": 0.6,
  "theme": "dark"
}
```

Background modes include: `transparent`, `color`, `gradient`, and `image`.

Controlling Z-Index (Window Stacking Order):

New windows are automatically placed on top. To change stacking order for existing windows, use the `bring_to_front` and `send_to_back` commands:

```
{
  "command": "bring_to_front",
  "window_id": "important-widget"
}
```

```
{
  "command": "send_to_back",
  "window_id": "background-widget"
```

```
}
```

The z-index is automatically managed - `bring_to_front` assigns the highest z-index, while `send_to_back` assigns the lowest. Window stacking order is preserved when you save and load screen states.

Windows can be in different states: - **Normal** - Regular display - **Minimized** - Hidden but not closed - **Maximized** - Full screen

0.1.5.2 Controlling Windows via Touch and Mouse

King Kiosk windows respond to both touch and mouse input, giving you intuitive direct control regardless of your device type.

Touch Controls (Tablets, Phones, Touch Screens): - **Tap and drag the title bar** - Move the window - **Drag the edges or corners** - Resize the window - **Double-tap the title bar** - Toggle maximize - **Swipe down on title bar** - Minimize - **Tap the X button** - Close the window

Mouse Controls (Desktop, Laptop): - **Click and drag the title bar** - Move the window - **Drag the edges or corners** - Resize the window - **Double-click the title bar** - Toggle maximize - **Click the minimize button** - Minimize the window - **Click the X button** - Close the window - **Right-click the title bar** - Access window context menu - **Mouse wheel** - Scroll content within the window

Apple TV Remote: - **D-pad** - Navigate between windows using spatial focus - **Select button** - Interact with focused window - **Play/Pause** - Enter Move Mode to reposition windows - **Menu button** - Exit modes, go back, close menus

See Section 11: Apple TV Experience for the complete guide to King Kiosk's revolutionary multi-window system on tvOS.

0.1.5.3 Controlling Windows via MQTT

Creating, moving, and managing windows remotely is where King Kiosk shines.

Creating a Clock Window:

```
{
  "command": "open_clock",
  "window_id": "lobby-clock",
  "mode": "analog",
  "theme": "dark"
}
```

Creating a Web Browser Window:

```
{
  "command": "open_browser",
  "url": "https://example.com",
  "title": "My Browser"
}
```

Moving a Window:

```
{
  "command": "move_window",
  "window_id": "lobby-clock",
  "x": 500,
  "y": 200
}
```

Window movement is smoothly animated by default (including Apple TV). To snap immediately instead of animating:

```
{
  "command": "move_window",
  "window_id": "lobby-clock",
  "x": 500,
  "y": 200,
  "animate": false
}
```

Resizing a Window:

```
{
  "command": "resize_window",
  "window_id": "lobby-clock",
  "width": 400,
  "height": 400
}
```

Closing a Window:

```
{
  "command": "close_window",
  "window_id": "lobby-clock"
}
```

Closing All Windows:

```
{
  "command": "close_all_windows"
}
```

Bringing a Window to Front:

```
{
  "command": "bring_to_front",
  "window_id": "lobby-clock"
}
```

Sending a Window to Back:

```
{
  "command": "send_to_back",
  "window_id": "lobby-clock"
}
```

Maximizing a Window:

```
{
  "command": "maximize_window",
  "window_id": "lobby-clock"
}
```

Minimizing a Window:

```
{
  "command": "minimize_window",
  "window_id": "lobby-clock"
}
```

0.1.5.4 Window Layouts and Arrangements

Switch between different layout modes:

Set Window Mode:

```
{
  "command": "window_mode",
  "mode": "tiling"
}
```

Available modes: - `tiling` or `tile` - Windows automatically arrange in a grid - `floating` or **`float`** - Windows can overlap freely - `toggle` - Switch between tiling and floating

---

0.1.6 5. Saving and Managing Screen States

0.1.6.1 What is a Screen State?

A screen state is a snapshot of your current display layout - all the windows, their positions, sizes, and configurations. Think of it as a "preset" you can recall instantly.

This is incredibly powerful for scenarios like: - **Morning Mode** - Calendar, weather, and news - **Movie Mode** - Single full-screen media player - **Dashboard Mode** - Multiple gauges and charts for monitoring - **Presentation Mode** - Specific content for meetings

0.1.6.2 Saving Your Layout

Once you've arranged your display how you like it:

```
{
  "command": "save_screen_state",
  "name": "Morning Dashboard"
}
```

If a state with that name already exists and you want to overwrite it:

```
{
  "command": "save_screen_state",
  "name": "Morning Dashboard",
  "overwrite": true
}
```

0.1.6.3 Loading Saved Layouts

Instantly switch to a saved layout:

```
{
  "command": "load_screen_state",
  "name": "Morning Dashboard"
}
```

All current windows will close, and the saved layout will be restored exactly as you saved it.

0.1.6.4 Listing Your Saved States

See all available layouts:

```
{
  "command": "list_screen_states"
}
```

Response includes the name, window count, and when each was saved.

0.1.6.5 Deleting a Saved State

Remove a layout you no longer need:

```
{
  "command": "delete_screen_state",
  "name": "Old Layout"
}
```

0.1.6.6 Exporting and Importing for Backup

Export a layout for backup or transfer:

```
{
  "command": "export_screen_state",
  "name": "Morning Dashboard"
}
```

The response includes the complete screen state data that you can save externally.

Import a previously exported layout:

```
{
  "command": "import_screen_state",
  "name": "Imported Dashboard",
  "screen_state": { ...exported data... },
  "overwrite": false
}
```

Tip: You can include `response_topic` and `correlation_id` on these commands to make automation workflows easier to track.

0.1.6.7 Full Device Backup and Restore (Settings + Layouts)

If you want a complete portable snapshot (not just one layout), use `get_config`.

Step 1: Export a full backup

```
{
  "command": "get_config",
  "include_secrets": true,
  "include_layouts": true,
  "correlation_id": "backup-2026-02-06",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

This returns: - `config` (all current settings) - `screen_states` (all saved named layouts) - `current_layout` (the live layout snapshot) - `screen_state_count`

If you plan to share the backup outside your trusted environment, set `include_secrets` to **`false`** so sensitive fields are masked.

Step 2: Restore (or clone to another tablet)

```
{
  "command": "provision",
  "settings": { ...config from backup... },
  "screen_states": [ ...screen_states from backup... ],
  "current_layout": { ...current_layout from backup... },
  "overwrite": true,
  "correlation_id": "restore-2026-02-06"
}
```

To clone multiple devices, publish the same payload to each target device's `kingkiosk/{device_id}/system/cmd` topic.

Provision responses include what succeeded and failed: - `applied_settings` - `failed_settings` - `screen_states_imported` - `screen_states_failed` - `current_layout_applied`

0.1.6.8 Remote Feature Server Provisioning

You can configure Feature Server settings remotely over MQTT, so you do not need to open the Settings screen on each device.

In plain English: - `featureServerEnabled`: turns Feature Server on or off. - `featureServerAutoConnect` : if on, the app will reconnect automatically after app restart/network recovery. - `featureServerUrl` : the server host/IP to connect to (for best cross-platform compatibility, use just host/IP, for example `192.168.1.50`). - `featureServerUseHttps`: use secure WebSocket (`wss`) if your server is configured for TLS. - `featureServerProduceAudio`: whether microphone audio should be produced to the server. - `intercomEnabled`: whether the device should participate in intercom/broadcast.

Example command (publish to `kingkiosk/{device_id}/system/cmd`):

```
{
  "command": "provision",
  "settings": {
    "featureServerEnabled": true,
    "featureServerAutoConnect": true,
    "featureServerUrl": "192.168.1.50",
    "featureServerUseHttps": false,
    "featureServerProduceAudio": true,
    "intercomEnabled": true
  },
  "correlation_id": "feature-server-setup-001"
}
```

How to verify it worked: - Subscribe to `kingkiosk/{device_id}/feature_server/state`. - This topic is retained, so you get the latest status immediately. - Watch fields like `enabled`, `connected`, `state`, and `last_error`.

Important behavior: - If `featureServerEnabled` is **`false`**, state stays `disabled` (gray in UI) and auto-reconnect does not run.

0.1.6.9 Scheduling Screen States

Automatically switch layouts based on time of day:

Set up a schedule:

```
{
  "command": "set_screen_schedule",
  "entries": [
    {
      "id": "morning",
      "screen_state": "Morning Dashboard",
      "at": "07:00",
      "days": [1, 2, 3, 4, 5],
      "enabled": true
    },
    {
      "id": "evening",
      "screen_state": "Evening Mode",
      "at": "18:00",
      "days": [1, 2, 3, 4, 5, 6, 7],
      "enabled": true
    }
  ],
  "enabled": true
}
```

Days are: 1=Monday, 2=Tuesday, … 7=Sunday

View current schedule:

```
{
  "command": "list_screen_schedule"
}
```

Enable/Disable scheduling:

```
{
  "command": "enable_screen_schedule"
}
```

```
{
  "command": "disable_screen_schedule"
}
```

Manually trigger a scheduled state:

```
{
  "command": "trigger_screen_schedule",
  "screen_state": "Morning Dashboard"
}
```

0.1.6.10 Fleet Layout Replication

For multi-display deployments, you can replicate layouts across all devices in a “fleet”:

Publish your current layout to all fleet members:

```
{
  "command": "replicate_layout",
  "fleet_id": "lobby-displays"
}
```

Subscribe a device to receive fleet layouts:

```
{
  "command": "subscribe_fleet",
  "fleet_id": "lobby-displays",
  "auto_apply": true
}
```

Stop receiving fleet layouts:

```
{
  "command": "unsubscribe_fleet",
  "fleet_id": "lobby-displays"
}
```

0.1.7 6. System Commands

0.1.7.1 Device Control

List all active windows:

```
{
  "command": "list_windows"
}
```

Returns detailed information about every window including ID, type, position, size, and configuration.

Get device information: Subscribe to `kingkiosk/{device_id}/info` to receive: - Device ID and version - Platform (Android, iOS, macOS, etc.) - Supported capabilities - Available widget types - Currently active widgets - `timestamp` (when this info message was published) - `app_start_timestamp` (when the app process started)

0.1.7.2 Volume and Brightness

Set Volume (0.0 to 1.0):

```
{
  "command": "set_volume",
  "value": 0.5
}
```

Mute/Unmute:

```
{
  "command": "mute"
}
```

```
{
  "command": "unmute"
}
```

Set Brightness (0.0 to 1.0):

```
{
  "command": "set_brightness",
  "value": 0.8
}
```

Get Current Brightness:

```
{
  "command": "get_brightness"
}
```

Restore Default Brightness:

```
{
  "command": "restore_brightness"
}
```

0.1.7.3 Notifications and Alerts

Show a Toast Notification:

```
{
  "command": "notify",
  "title": "Welcome",
  "message": "Good morning!",
  "duration": 5
}
```

Priority levels: `low`, `normal`, `high`

Show a rich notification with formatting:

```
{
  "command": "notify",
  "title": "Update Available",
  "message": "**Version 2.0** is now available with *exciting new features*!",
  "format": "markdown",
  "thumbnail": "https://example.com/icon.png",
  "priority": "high"
}
```

Show an Alert Dialog:

```
{
  "command": "alert",
  "title": "Attention Required",
  "message": "Please check the system status.",
  "type": "warning",
  "position": "center",
```

```
  "auto_dismiss_seconds": 30
}
```

Alert options: - `type`: `info`, `success`, `warning`, `error` - `position`: Where on screen to show - `show_border`: true/false - `border_color`: Hex color code - `auto_dismiss_seconds`: 1-300 seconds

0.1.7.4 Visual Effects

Halo Effect (Border Glow):

Create an attention-grabbing glowing border around the screen or a specific window:

```
{
  "command": "halo_effect",
  "enabled": true,
  "color": "#FF0000",
  "width": 20,
  "intensity": 0.8,
  "pulse_mode": "gentle"
}
```

Pulse modes: `none`, `gentle`, `moderate`, `alert`

Apply to a specific window:

```
{
  "command": "halo_effect",
  "window_id": "important-alert",
  "enabled": true,
  "color": "#FFD700",
  "pulse_mode": "alert"
}
```

Disable the effect:

```
{
  "command": "halo_effect",
  "enabled": false
}
```

Screensaver (Bouncing Content):

Display a full-screen screensaver with bouncing items like clocks, images, or text. Each item moves independently around the screen, bouncing off the edges. Tap anywhere to dismiss.

Turn on a simple clock screensaver:

```
{
  "command": "screensaver",
  "action": "enable",
  "items": [
    {
      "type": "clock",
```

```
      "config": { "show_seconds": true, "text_color": "#00FF00" },
      "width": 250,
      "height": 100,
      "speed": 1.0,
      "scale": 1.5
    }
  ],
  "background_color": "#000000",
  "background_opacity": 0.9
}
```

Add multiple bouncing items (each moves independently):

```
{
  "command": "screensaver",
  "action": "enable",
  "items": [
    {
      "id": "clock_1",
      "type": "clock",
      "config": { "show_seconds": false },
      "speed": 0.8
    },
    {
      "id": "logo_1",
      "type": "image",
      "config": { "url": "https://example.com/logo.png" },
      "width": 150,
      "height": 150,
      "speed": 1.2
    },
    {
      "id": "welcome",
      "type": "text",
      "config": { "text": "Welcome!", "font_size": 48 },
      "speed": 1.0
    }
  ]
}
```

Item types you can use: - `clock` - Digital clock (options: `show_seconds`, `show_date`, `text_color`, `font_size`) - `image` or `logo` - Image from URL (options: `url`, `fit`) - `text` - Custom text (options: `text`, `font_size`, `font_weight`, `text_color`) - `icon` - System icon (options: `icon`, `size`, `color`) - icons include: star, heart, home, music, sun, moon

Item properties: - `speed` - How fast it bounces (0.5 = slow, 1.0 = normal, 2.0 = fast) - `scale` - Size multiplier (1.5 = 50% bigger) - `width`, `height` - Base size in pixels

Turn off the screensaver:

```
{
  "command": "screensaver",
  "action": "disable"
}
```

Add an item to a running screensaver:

```
{
```

```
  "command": "screensaver",
  "action": "add_item",
  "item": {
    "id": "new_icon",
    "type": "icon",
    "config": { "icon": "star", "color": "#FFD700" },
    "speed": 1.5
  }
}
```

Remove an item:

```
{
  "command": "screensaver",
  "action": "remove_item",
  "item_id": "new_icon"
}
```

0.1.7.5 Text-to-Speech

Make your display speak. When the Feature Server is connected, TTS automatically uses the high-quality Piper engine. When disconnected, it falls back to on-device TTS. The same commands work in both modes.

Basic speech:

```
{
  "command": "tts",
  "text": "Welcome to King Kiosk!"
}
```

With voice customization:

```
{
  "command": "tts",
  "text": "The current temperature is 72 degrees.",
  "language": "en-US",
  "volume": 0.8,
  "rate": 0.9,
  "pitch": 1.0
}
```

With a specific Piper voice (Feature Server):

```
{
  "command": "tts",
  "text": "Good morning! The front door is unlocked.",
  "voice": "en_US-lessac-medium",
  "volume": 0.9,
  "rate": 0.5
}
```

Control playback:

```
{
```

```
  "command": "tts",
  "action": "stop"
}
```

Actions: `speak`, `stop`, `pause`, `resume`

Get available voices:

```
{
  "command": "tts",
  "action": "getVoices",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

When the Feature Server is connected, this returns Piper voices with details like `voiceId`, `languageCode`, `quality`, and `installed` status. You can filter results:

```
{
  "command": "tts",
  "action": "getVoices",
  "language": "en_US",
  "quality": "medium",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

Install a voice on the Feature Server:

```
{
  "command": "tts",
  "action": "voice_pull",
  "voice": "en_US-lessac-medium",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

Check TTS status:

```
{
  "command": "tts",
  "action": "status",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

The response includes `feature_server_connected`, `engine` (either `piper` or the on-device engine), and current settings.

0.1.7.6 Speech-to-Text

Enable voice input:

Start listening:

```
{
  "command": "stt",
  "action": "start"
}
```

Stop listening and get result:

```
{
  "command": "stt",
  "action": "stop"
}
```

Configure language:

```
{
  "command": "stt",
  "action": "set_language",
  "language": "en"
}
```

Send transcriptions to AI agent:

```
{
  "command": "stt",
  "action": "send_to_ai_agent",
  "enabled": true
}
```

Audio Input Device (Microphone)

The microphone used for Speech-to-Text is the one selected in **Settings -> Speech & AI -> Audio Input Device**.

You can also query/change it via MQTT:

```
{
  "command": "unified_audio",
  "action": "status",
  "response_topic": "kingkiosk/{device_id}/system/response"
}
```

```
{
  "command": "unified_audio",
  "action": "list_devices",
  "response_topic": "kingkiosk/{device_id}/system/response"
}
```

```
{
  "command": "unified_audio",
  "action": "set_device",
  "device_id": "1",
  "response_topic": "kingkiosk/{device_id}/system/response"
}
```

0.1.7.7 Screenshots and Screen Capture

Take a screenshot:

```
{
  "command": "screenshot",
```

```
  "notify": true
}
```

The screenshot is published as image data to `kingkiosk/{device_id}/screenshot`. Home Assistant can use this as a camera feed to view your display remotely.

Enable periodic screenshot capture:

```
{
  "command": "screenshot_camera",
  "action": "enable",
  "interval": 30
}
```

This captures and publishes a screenshot every 30 seconds.

Disable screenshot capture:

```
{
  "command": "screenshot_camera",
  "action": "disable"
}
```

0.1.7.8 Background Settings

Set a solid color background:

```
{
  "command": "set_background",
  "type": "default"
}
```

Set an image background:

```
{
  "command": "set_background",
  "type": "image",
  "image_url": "https://example.com/wallpaper.jpg"
}
```

Set a web page as background:

```
{
  "command": "set_background",
  "type": "webview",
  "web_url": "https://example.com/animated-background"
}
```

Get current background:

```
{
  "command": "get_background"
}
```

0.1.7.9 Screensaver

King Kiosk includes a built-in screensaver that activates after a period of inactivity. This helps prevent screen burn-in and can create a more polished appearance when the display isn't being actively used.

Configuring the Screensaver (In-App):

Go to **Settings → App Settings** and find the **Screensaver** option. You can choose from three modes:

Mode	Description
Off	Screensaver is disabled
Dim	Screen goes black after the timeout period
Screensaver	A bouncing clock appears after the timeout period

You can also set the **Timeout** (1-60 minutes) - how long the device must be idle before the screensaver activates.

Waking the Screensaver:

- **Touch/Remote:** Tap anywhere on the screen or press any button on the remote to wake the display
- **MQTT Command:** Send a wake command to dismiss the screensaver remotely

Wake from screensaver via MQTT:

```
{
  "command": "screensaver",
  "action": "wake"
}
```

Alternative actions that also wake the screensaver: `wake_up`, `deactivate`

Get screensaver state:

```
{
  "command": "screensaver",
  "action": "get_state"
}
```

> **Note:** The idle screensaver is separate from the MQTT-controlled bouncing screensaver. The idle screensaver activates automatically based on inactivity, while the MQTT screensaver can be triggered remotely with custom items like images, text, and clocks.

0.1.7.10 AI Integration

King Kiosk can connect to various AI services for voice conversations and intelligent responses.

Enable AI agent:

```
{
  "command": "ai_agent",
  "action": "enable"
}
```

Configure Home Assistant conversation:

```
{
  "command": "ai_agent",
  "action": "configure_home_assistant",
  "base_url": "http://homeassistant.local:8123",
  "access_token": "your_long_lived_token"
}
```

Configure a chat bot:

```
{
  "command": "provision_ai_chatbot",
  "provider": "anthropic",
  "api_key": "your_api_key",
  "model": "claude-3-sonnet",
  "enable_speech": true,
  "enable_tts": true
}
```

Supported providers: `anthropic`, `openai`, `gemini`, `ollama`

Send a message to the AI:

```
{
  "command": "ai_agent",
  "action": "send_message",
  "message": "What's the weather like today?"
}
```

0.1.7.11 Batch Commands and Scripting

Batch scripting lets you send a single MQTT command that runs a whole sequence of steps on the kiosk — one after another, with optional pauses between them. Instead of publishing five separate MQTT messages from Home Assistant and hoping the timing works out, you send one `batch` command and King Kiosk handles the rest.

Any command you can send individually over MQTT works inside a batch. Open windows, load a saved layout, play audio, speak text, change the background, adjust brightness — mix and match however you want.

0.1.7.11.1 Why Use Batches?

- **Morning routines** — Set the volume, load a dashboard, announce the weather, all from one automation trigger
- **Scene changes** — Close everything, set a background, open specific windows in the right positions, with timed delays so the transitions look smooth
- **Presentations and demos** — Script a walkthrough that steps through screens automatically
- **Provisioning a new device** — Configure a kiosk from scratch: set the background, add windows, enable features, all in one shot

0.1.7.11.2 How It Works Send a `batch` command with a `commands` array. Each item is a normal King Kiosk MQTT command. They execute in order, top to bottom. Use `wait` steps to pause between commands.

Only one batch can run at a time. If you send a new batch while one is already running, it will be rejected.

If any individual command fails, the batch logs the error and keeps going with the next step — it won't stop the whole sequence.

0.1.7.11.3 Basic Example

```
{
  "command": "batch",
  "commands": [
    { "command": "set_volume", "value": 0.3 },
    { "command": "wait", "seconds": 2 },
    { "command": "notify", "title": "Starting", "message": "Morning routine beginning" },
    { "command": "load_screen_state", "name": "Morning Dashboard" },
    { "command": "tts", "text": "Good morning! Here's your dashboard." }
  ]
}
```

0.1.7.11.4 Provisioning a Fresh Device Set up a brand new kiosk with one command:

```
{
  "command": "batch",
  "commands": [
    { "command": "close_all_windows" },
    { "command": "set_background", "type": "image", "image_url": "https://example.com/
        wallpaper.jpg" },
    { "command": "wait", "seconds": 1 },
    { "command": "open_browser", "window_id": "dashboard", "url": "http://homeassistant.
        local:8123", "auto_connect": true },
    { "command": "set_volume", "value": 0.5 },
    { "command": "set_brightness", "value": 0.8 },
    { "command": "save_screen_state", "name": "Default Layout" }
  ]
}
```

0.1.7.11.5 Scene Transitions

Create smooth scene changes with timed delays:

```
{
  "command": "batch",
  "commands": [
    { "command": "close_all_windows" },
    { "command": "wait", "seconds": 0.5 },
    { "command": "set_background", "type": "image", "image_url": "https://example.com/
        movie-night.jpg" },
    { "command": "wait", "seconds": 0.5 },
    { "command": "open_browser", "window_id": "media", "url": "https://youtube.com", "
        auto_connect": true },
    { "command": "set_brightness", "value": 0.3 },
    { "command": "set_volume", "value": 0.7 },
    { "command": "tts", "text": "Movie night is ready." }
  ]
}
```

0.1.7.11.6 The Wait Command

Use `wait` to pause between steps. The `seconds` value can range from 0 to 300 (5 minutes max). Fractional values like `0.5` work fine for short pauses.

```
{ "command": "wait", "seconds": 5 }
```

Waits are interruptible — if you cancel the batch during a wait, it stops immediately rather than waiting for the full duration.

0.1.7.11.7 Saving and Loading Scenes with Screen States

You can save the current layout as a named screen state, then restore it later — either manually or as part of a batch. This is the easiest way to build reusable "scenes."

Save a scene:

```
{ "command": "save_screen_state", "name": "Movie Night" }
```

Load a saved scene:

```
{ "command": "load_screen_state", "name": "Movie Night" }
```

List all saved scenes:

```
{
  "command": "list_screen_states",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

Delete a scene:

```
{ "command": "delete_screen_state", "name": "Old Layout" }
```

Export a scene as JSON (for backup or sharing to other devices):

```
{
```

```
  "command": "export_screen_state",
  "name": "Movie Night",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

Import a scene from JSON:

```
{
  "command": "import_screen_state",
  "name": "Movie Night",
  "screen_state": { ... },
  "overwrite": true
}
```

0.1.7.11.8 Monitoring and Canceling Check batch status:

```
{
  "command": "batch_status",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

The response tells you whether a batch is running, how far along it is, and the total number of steps.

Cancel a running batch:

```
{
  "command": "kill_batch_script"
}
```

The batch stops after the current command finishes (or immediately if it's in a `wait` step).

0.1.7.11.9 Home Assistant Automation Example

```
automation:
  - alias: "Good Morning Routine"
    trigger:
      - platform: time
        at: "07:00:00"
    action:
      - service: mqtt.publish
        data:
          topic: "kingkiosk/kitchen_tablet/system/cmd"
          payload: >
            {
              "command": "batch",
              "commands": [
                { "command": "set_brightness", "value": 0.6 },
                { "command": "set_volume", "value": 0.4 },
                { "command": "load_screen_state", "name": "Morning Dashboard" },
                { "command": "wait", "seconds": 1 },
                { "command": "tts", "text": "Good morning! Here's your schedule for today.
                    " }
              ]
            }
```

0.1.8 7. Widget Reference

0.1.8.1 Web Browsers

Display any web page in a window.

Open a web browser:

```
{
  "command": "open_browser",
  "url": "https://www.google.com",
  "title": "Search",
  "window_id": "google-browser",
  "x": 100,
  "y": 100,
  "width": 800,
  "height": 600
}
```

Alternative commands: `open_web`, `open_simple_web`

Apple TV Convenience: On tvOS (Apple TV), these browser commands (`open_browser`, `open_web`, `open_simple_web`) automatically map to `create_remote_browser`. This means you can use the same familiar API across all platforms - King Kiosk handles the platform-specific implementation for you. Your automations work everywhere without modification.

0.1.8.2 Remote Browser (Apple TV & iOS)

For Apple TV and iOS devices where local browser rendering is limited, King Kiosk offers a remote browser experience. A server renders the web page and streams it to your device in real-time.

Create a remote browser:

```
{
  "command": "create_remote_browser",
  "window_id": "browser_1",
  "initial_url": "https://www.google.com",
  "auto_connect": true
}
```

Optional parameters: `name`, `video_profile` (`auto`, `720p30`, `1080p30`, `1080p60`), `show_overlay`

Navigate to a URL:

```
{
  "command": "navigate_remote_browser",
  "window_id": "browser_1",
  "url": "https://www.example.com"
}
```

Browser navigation:

```
{
  "command": "remote_browser_back",
  "window_id": "browser_1"
}
```

```
{
  "command": "remote_browser_forward",
  "window_id": "browser_1"
}
```

```
{
  "command": "remote_browser_reload",
  "window_id": "browser_1"
}
```

Simulate interaction:

```
{
  "command": "remote_browser_click",
  "window_id": "browser_1",
  "x": 400,
  "y": 300
}
```

```
{
  "command": "remote_browser_scroll",
  "window_id": "browser_1",
  "delta_x": 0,
  "delta_y": -100
}
```

```
{
  "command": "remote_browser_key",
  "window_id": "browser_1",
  "key": "Enter"
}
```

```
{
  "command": "remote_browser_text",
  "window_id": "browser_1",
  "text": "search query"
}
```

Apple TV Remote mapping: - D-pad: Move pointer (with acceleration) - Select/Enter: Click - Menu/Escape: Navigate back - Play/Pause: Toggle Pointer/Scroll mode - Touch swipe: Pointer mode = move cursor, Scroll mode = scroll - Long press: Right-click

Browser Session Persistence:

Remote browser sessions automatically remember your logins, just like a regular browser. When you log into Netflix, YouTube, or any website, you'll stay logged in - even after closing the app or restarting your device.

How it works: - Your login sessions and site data are tied to the `window_id` you specify - Use the same `window_id` = same login sessions preserved - Use a different `window_id` = fresh browser with

no saved logins

Example: If you create a browser with `"window_id": "netflix"`, log into Netflix, then close it - the next time you create a browser with `"window_id": "netflix"`, you'll still be logged in. But `"window_id": "youtube"` would be a separate browser that doesn't share Netflix's login.

Key points: - **Automatic**: No configuration needed - just works - **Isolated**: Each `window_id` has its own separate logins and data - **Persistent**: Survives app restarts and device reboots

Clear browser data (logout/reset):

```
{
  "command": "remote_browser_clear_data",
  "window_id": "browser_1"
}
```

This clears all cookies, localStorage, and cached data for the browser tile, then restarts the session. Use this to implement "logout" functionality for web apps.

Debug status (helpful for double-audio / missing-audio issues):

```
{
  "command": "remote_browser_status",
  "window_id": "browser_1"
}
```

0.1.8.3 Clock Widget

Display beautiful analog or digital clocks.

Create a clock:

```
{
  "command": "open_clock",
  "window_id": "main-clock",
  "mode": "analog",
  "theme": "dark",
  "show_numbers": true,
  "show_second_hand": true,
  "x": 50,
  "y": 50,
  "width": 300,
  "height": 300
}
```

Change clock mode via element command: Topic: `kingkiosk/{device_id}/element/main-clock/cmd`

```
{
  "command": "set_mode",
  "mode": "digital"
}
```

Toggle between analog and digital:

```
{
  "command": "toggle_mode"
}
```

Configure appearance:

```
{
  "command": "configure",
  "theme": "light",
  "background_mode": "gradient",
  "background_color": "#1a1a2e"
}
```

Background modes: `transparent`, `gradient`, `color`, `image`

0.1.8.4 Weather Widget

Display current weather and forecasts from OpenWeather.

Create a weather widget:

```
{
  "command": "open_weather_client",
  "window_id": "weather-1",
  "api_key": "your_openweather_api_key",
  "location": "New York",
  "units": "imperial",
  "show_forecast": true,
  "auto_refresh": true,
  "refresh_interval": 1800,
  "x": 400,
  "y": 50,
  "width": 350,
  "height": 400
}
```

Units: `imperial` (Fahrenheit), `metric` (Celsius), `standard` (Kelvin)

You can also use coordinates instead of city name:

```
{
  "latitude": 40.7128,
  "longitude": -74.0060
}
```

0.1.8.5 Media Player

Play video, audio, and images.

Play a video:

```
{
  "command": "play_media",
  "type": "video",
  "url": "https://example.com/video.mp4",
  "window_id": "video-player",
  "title": "My Video",
  "loop": true,
  "x": 100,
  "y": 100,
  "width": 640,
  "height": 360
}
```

Play audio:

```
{
  "command": "play_media",
  "type": "audio",
  "url": "https://example.com/music.mp3",
  "style": "visualizer"
}
```

Audio styles: `window` (basic player), `visualizer` (spectrum analyzer)

Visualizer options:

```
{
  "command": "play_media",
  "type": "audio",
  "url": "https://example.com/music.mp3",
  "style": "visualizer",
  "visualizer_type": "fft",
  "bars": 64,
  "color_scheme": "rainbow",
  "show_peaks": true
}
```

Display an image:

```
{
  "command": "play_media",
  "type": "image",
  "url": "https://example.com/photo.jpg",
  "window_id": "image-display"
}
```

Control playback:

```
{
  "command": "play",
  "window_id": "video-player"
}
```

```
{
  "command": "pause",
  "window_id": "video-player"
}
```

```
{
```

```
  "command": "close",
  "window_id": "video-player"
}
```

Background audio control:

```
{
  "command": "play_audio"
}
```

```
{
  "command": "pause_audio"
}
```

```
{
  "command": "stop_audio"
}
```

```
{
  "command": "seek_audio",
  "position": 30
}
```

0.1.8.6 YouTube Player

Play YouTube videos directly.

Open YouTube:

```
{
  "command": "youtube",
  "url": "https://www.youtube.com/watch?v=dQw4w9WgXcQ",
  "title": "Music Video",
  "window_id": "youtube-player",
  "x": 100,
  "y": 100,
  "width": 800,
  "height": 450
}
```

0.1.8.7 Image Carousel

Create beautiful slideshows.

Create a carousel:

```
{
  "command": "create_carousel",
  "window_id": "photo-carousel",
  "title": "Family Photos",
  "items": [
    { "type": "image", "url": "https://example.com/photo1.jpg" },
    { "type": "image", "url": "https://example.com/photo2.jpg" },
    { "type": "image", "url": "https://example.com/photo3.jpg" }
```

```
  ],
  "config": {
    "auto_play": true,
    "interval": 5,
    "infinite_scroll": true,
    "show_indicator": true
  }
}
```

Video carousel:

```
{
  "command": "create_video_carousel",
  "window_id": "video-carousel",
  "items": [
    { "type": "video", "url": "https://example.com/video1.mp4" },
    { "type": "video", "url": "https://example.com/video2.mp4" }
  ]
}
```

Add items to existing carousel:

```
{
  "command": "add_carousel_item",
  "window_id": "photo-carousel",
  "item": { "type": "image", "url": "https://example.com/photo4.jpg" }
}
```

Navigate carousel:

```
{
  "command": "navigate_carousel",
  "window_id": "photo-carousel",
  "index": 5
}
```

Or use `next` or `previous` as index values.

Configure carousel:

```
{
  "command": "set_carousel_config",
  "window_id": "photo-carousel",
  "auto_play": false,
  "interval": 10,
  "scroll_direction": "vertical"
}
```

0.1.8.8 Gauges and Thermostats

Display values visually with beautiful gauges.

Create a simple gauge:

```
{
  "command": "create_gauge",
  "gauge_id": "cpu-usage",
```

```
  "title": "CPU Usage",
  "gauge_type": "circular",
  "min": 0,
  "max": 100,
  "value": 45,
  "unit": "%",
  "x": 100,
  "y": 100,
  "width": 200,
  "height": 200
}
```

Gauge types: `linear`, `circular`, `radial`, `semicircular`, `thermostat`

Create a thermostat gauge with zones:

```
{
  "command": "create_gauge",
  "gauge_id": "living-room-thermostat",
  "title": "Living Room",
  "gauge_type": "thermostat",
  "min": 50,
  "max": 90,
  "unit": "°F",
  "step_size": 1,
  "interactive": true,
  "color_mode": "zones",
  "config": {
    "zones": [
      { "min": 50, "max": 65, "color": "#3498db", "label": "Cold" },
      { "min": 65, "max": 75, "color": "#2ecc71", "label": "Comfort" },
      { "min": 75, "max": 90, "color": "#e74c3c", "label": "Hot" }
    ],
    "pointers": [
      {
        "id": "current",
        "label": "Now",
        "color": "#FFFFFF",
        "locked": true,
        "style": "needle"
      },
      {
        "id": "target",
        "label": "Target",
        "color": "#00BFFF",
        "locked": false,
        "style": "target"
      }
    ]
  }
}
```

Color modes: `solid`, `gradient`, `thresholds`, `zones`

Pointer styles: `needle`, `dot`, `triangle`, `line`, `target`

Update gauge value:

```
{
  "command": "set_gauge_value",
  "gauge_id": "cpu-usage",
  "value": 72
```

```
}
```

Update a specific pointer:

```
{
  "command": "set_pointer_value",
  "gauge_id": "living-room-thermostat",
  "pointer_id": "current",
  "value": 72
}
```

Lock/unlock gauge:

```
{
  "command": "lock_gauge",
  "gauge_id": "living-room-thermostat"
}
```

```
{
  "command": "unlock_gauge",
  "gauge_id": "living-room-thermostat"
}
```

Configure gauge:

```
{
  "command": "configure_gauge",
  "gauge_id": "cpu-usage",
  "min": 0,
  "max": 100,
  "decimals": 1,
  "show_min_max": true
}
```

Platform controls: - tvOS: Use Siri Remote D-pad to select pointers and adjust values - iOS: Touch/swipe on gauge to adjust, tap pointers to select - macOS: Keyboard arrows to adjust, mouse click to select

0.1.8.9 Charts

Visualize data with various chart types.

Create a line chart:

```
{
  "command": "create_chart",
  "chart_id": "temperature-chart",
  "chart_type": "line",
  "title": "Temperature History",
  "max_points": 60,
  "x": 100,
  "y": 100,
  "width": 400,
  "height": 300
}
```

Chart types: `line`, `bar`, `pie`

Add data to chart:

```
{
  "command": "append_chart_data",
  "chart_id": "temperature-chart",
  "value": 72.5
}
```

Replace all chart data:

```
{
  "command": "replace_chart_data",
  "chart_id": "temperature-chart",
  "values": [68, 70, 72, 75, 73, 71, 69]
}
```

Update pie chart:

```
{
  "command": "update_pie_chart",
  "chart_id": "sales-pie",
  "slices": [
    { "value": 40, "label": "Product A", "color": "#3498db" },
    { "value": 30, "label": "Product B", "color": "#2ecc71" },
    { "value": 30, "label": "Product C", "color": "#e74c3c" }
  ]
}
```

Configure chart:

```
{
  "command": "configure_chart",
  "chart_id": "temperature-chart",
  "config": {
    "primaryColor": "#3498db"
  }
}
```

Reset chart data:

```
{
  "command": "reset_chart",
  "chart_id": "temperature-chart"
}
```

0.1.8.10 Maps

Display interactive maps with custom pins and overlays.

Create a map:

```
{
  "command": "open_map",
  "window_id": "location-map",
```

```
  "title": "Store Locations",
  "initial_camera": {
    "lat": 40.7128,
    "lon": -74.0060,
    "zoom": 12
  },
  "interaction": {
    "touch_enabled": true,
    "remote_enabled": true,
    "allow_user_drop_pins": false
  },
  "x": 100,
  "y": 100,
  "width": 600,
  "height": 400
}
```

Custom tile provider (e.g., for satellite imagery):

```
{
  "provider": {
    "url_template": "https://your-tile-server/{z}/{x}/{y}.png",
    "attribution": "Map data © Your Provider"
  }
}
```

Add pins to map: Topic: `kingkiosk/{device_id}/element/location-map/cmd`

```
{
  "command": "add_pins",
  "pins": [
    {
      "pin_id": "store-1",
      "lat": 40.7128,
      "lon": -74.0060,
      "icon": { "type": "default" }
    },
    {
      "pin_id": "store-2",
      "lat": 40.7580,
      "lon": -73.9855,
      "icon": { "type": "url", "value": "https://example.com/marker.png" }
    }
  ]
}
```

Move the camera:

```
{
  "command": "set_camera",
  "camera": {
    "lat": 40.7580,
    "lon": -73.9855,
    "zoom": 15
  }
}
```

Add text overlays:

```
{
  "command": "add_text",
```

```
  "texts": [
    {
      "text_id": "label-1",
      "text": "Headquarters",
      "anchor_type": "geo",
      "geo": { "lat": 40.7128, "lon": -74.0060 },
      "style": {
        "font_size_px": 16,
        "color": "#FFFFFF",
        "background_color": "#333333"
      }
    }
  ]
}
```

Clear all pins:

```
{
  "command": "clear_pins"
}
```

0.1.8.11 Canvas (Visual Diagrams)

Create custom visual diagrams with objects and connections.

Create a canvas:

```
{
  "command": "open_canvas",
  "window_id": "network-diagram",
  "title": "Network Topology",
  "doc": {
    "objects": [
      {
        "object_id": "router",
        "type": "node",
        "label": "Main Router",
        "x": 200,
        "y": 100,
        "width": 100,
        "height": 60
      },
      {
        "object_id": "server",
        "type": "node",
        "label": "Server",
        "x": 200,
        "y": 250,
        "width": 100,
        "height": 60
      }
    ],
    "connections": [
      {
        "connection_id": "link-1",
        "from": { "object_id": "router" },
        "to": { "object_id": "server" },
        "style": {
          "color": "#00FF00",
```

```
            "arrow": "end"
          }
        }
      ]
    },
    "interaction": {
      "touch_enabled": true,
      "edit_mode": false
    }
}
```

Object types: `node`, `text`, `shape`, `image`, `embed`, `group`

Shape types (for `type: shape`): `rect`, `round_rect`, `circle`, `line`

Add objects: Topic: `kingkiosk/{device_id}/element/network-diagram/cmd`

```
{
  "command": "add_objects",
  "objects": [
    {
      "object_id": "workstation-1",
      "type": "node",
      "label": "Workstation",
      "x": 50,
      "y": 250,
      "width": 80,
      "height": 50
    }
  ]
}
```

Add connections:

```
{
  "command": "add_connections",
  "connections": [
    {
      "connection_id": "link-2",
      "from": { "object_id": "router" },
      "to": { "object_id": "workstation-1" },
      "style": { "animated": true }
    }
  ]
}
```

Embed widgets in canvas:

```
{
  "object_id": "gauge-embed",
  "type": "embed",
  "x": 350,
  "y": 100,
  "width": 120,
  "height": 120,
  "embed": {
    "widget_type": "gauge",
    "id": "bandwidth-gauge",
    "config": {
      "min": 0,
      "max": 100,
```

```
      "unit": "Mbps"
    }
  }
}
```

0.1.8.12 Animated Text

Display text with stunning animations.

Create animated text:

```
{
  "command": "open_animated_text",
  "window_id": "welcome-text",
  "text": "Welcome to King Kiosk!",
  "preset": "typewriterShimmer",
  "style": {
    "fontSize": 48,
    "color": "#FFFFFF"
  },
  "x": 100,
  "y": 100,
  "width": 600,
  "height": 200
}
```

Available presets: - `typewriterShimmer` - Characters appear one by one with shimmer - `neonPulse` - Glowing neon effect - `bounceCascade` - Characters bounce in - `alertFlash` - Attention-grabbing flash - `tickerMarquee` - Scrolling ticker tape

Custom effects:

```
{
  "command": "open_animated_text",
  "window_id": "custom-text",
  "text": "Breaking News!",
  "effects": [
    {
      "type": "fade",
      "durationMs": 500,
      "stagger": { "eachMs": 50 }
    },
    {
      "type": "slide",
      "from": { "x": 0, "y": -50 },
      "to": { "x": 0, "y": 0 },
      "durationMs": 800,
      "easing": "easeOutBack"
    }
  ],
  "timeline": {
    "mode": "loop"
  }
}
```

Effect types: `fade`, `slide`, `scale`, `colorize`, `marquee`, `typewriter`

Easing options: `linear`, `easeOutCubic`, `easeInOutCubic`, `easeOutBack`

Update text content: Topic: `kingkiosk/{device_id}/element/welcome-text/cmd`

```
{
  "command": "set_text",
  "text": "New message here!"
}
```

Trigger animation replay:

```
{
  "command": "trigger"
}
```

0.1.8.13 MQTT Image

Display images that update automatically from MQTT messages.

Create an MQTT image tile:

```
{
  "command": "mqtt_image",
  "action": "open",
  "window_id": "camera-feed",
  "window_name": "Security Camera",
  "mqtt_topic": "home/camera/living_room/image",
  "is_base64": true,
  "x": 100,
  "y": 100,
  "width": 640,
  "height": 480
}
```

The image updates automatically whenever a new message arrives on the subscribed topic.

For JSON payloads:

```
{
  "mqtt_topic": "home/camera/snapshot",
  "json_field": "data.image"
}
```

This extracts the image from `{ "data": { "image": "base64data..."} }`.

Update subscription topic:

```
{
  "command": "mqtt_image",
  "action": "update_topic",
  "window_id": "camera-feed",
  "mqtt_topic": "home/camera/backyard/image"
}
```

0.1.8.14 MQTT Button

Create buttons that publish MQTT messages when pressed.

Create an MQTT button:

```
{
  "command": "mqtt_button",
  "action": "configure",
  "window_id": "light-toggle",
  "label": "Living Room Light",
  "mode": "toggle",
  "publish_topic": "home/lights/living_room/set",
  "publish_payload": { "state": "TOGGLE" },
  "subscription_topic": "home/lights/living_room/state",
  "icon_on": "lightbulb",
  "icon_off": "lightbulb_outline",
  "color_on": "#FFD700",
  "color_off": "#808080"
}
```

Modes: - `toggle` or **`switch`** - Two-state button - `icon_button` or `button` - Momentary button

Trigger the button via MQTT:

```
{
  "command": "mqtt_button",
  "action": "trigger",
  "window_id": "light-toggle"
}
```

0.1.8.15 Calendar (with Bidirectional Sync)

The King Kiosk calendar is more than just a display widget - it features **full bidirectional MQTT synchronization**. This means Home Assistant or any external system can add, update, and delete calendar events, and King Kiosk will broadcast changes made locally so external systems stay in sync.

The calendar sync is **always-on** - it works even when no calendar widget is visible on screen, making it perfect for automation scenarios.

Create a calendar widget:

```
{
  "command": "calendar",
  "action": "create",
  "window_id": "main-calendar",
  "name": "Family Calendar",
  "x": 100,
  "y": 100,
  "width": 400,
  "height": 350
}
```

Add an event (via system command):

```
{
  "command": "calendar",
  "action": "add_event",
  "window_id": "main-calendar",
  "event": {
    "title": "Team Meeting",
    "start": "2024-12-20T10:00:00",
    "end": "2024-12-20T11:00:00"
  }
}
```

Navigate to a specific date:

```
{
  "command": "calendar",
  "action": "go_to_date",
  "window_id": "main-calendar",
  "date": "2024-12-25"
}
```

Clear all events:

```
{
  "command": "calendar",
  "action": "clear_events",
  "window_id": "main-calendar"
}
```

0.1.8.15.1 Bidirectional MQTT Sync

For advanced integration with Home Assistant or other systems, use the element-level commands on the calendar's dedicated MQTT topic.

Topic: `kingkiosk/{device_id}/element/calendar/cmd`

Create or Update an Event (Upsert):

```
{
  "command": "calendar_upsert",
  "correlation_id": "ha-req-123",
  "event": {
    "uid": "doctor-appt-001",
    "title": "Doctor Appointment",
    "start": "2025-01-15T00:00:00Z",
    "end": "2025-01-15T23:59:59Z",
    "allDay": true,
    "description": "Bring insurance card",
    "color": "#4CAF50"
  }
}
```

The `uid` (or `id`) uniquely identifies the event. If an event with that ID exists, it's updated. Otherwise, a new event is created. Multiple events per day are fully supported.

Delete an Event:

```
{
  "command": "calendar_delete",
```

```
  "correlation_id": "ha-req-124",
  "uid": "doctor-appt-001"
}
```

Bulk Replace All Events:

Replace all calendar events at once - perfect for syncing from an external calendar source:

```
{
  "command": "calendar_bulk_replace",
  "correlation_id": "ha-req-125",
  "events": [
    {
      "uid": "meeting-001",
      "title": "Team Standup",
      "start": "2025-01-15T09:00:00Z",
      "allDay": false
    },
    {
      "uid": "meeting-002",
      "title": "Project Review",
      "start": "2025-01-15T14:00:00Z",
      "allDay": false
    },
    {
      "uid": "holiday-001",
      "title": "Company Holiday",
      "start": "2025-01-20T00:00:00Z",
      "allDay": true,
      "color": "#FF5722"
    }
  ]
}
```

Get Calendar Status:

```
{
  "command": "calendar_status"
}
```

0.1.8.15.2 Receiving Calendar Events

When events are added, updated, or deleted **locally** on the King Kiosk device (via touch interaction), the changes are broadcast so external systems can stay synchronized.

Subscribe to: `kingkiosk/{device_id}/element/calendar/event`

Event Added/Updated Locally:

```
{
  "event": "calendar_event_upserted",
  "origin": "kingkiosk",
  "timestamp": "2025-01-15T10:30:00Z",
  "event_data": {
    "uid": "new-event-001",
    "title": "Lunch with Client",
    "start": "2025-01-16T12:00:00Z",
    "allDay": false,
    "description": "Downtown restaurant"
```

```
  }
}
```

Event Deleted Locally:

```
{
  "event": "calendar_event_deleted",
  "origin": "kingkiosk",
  "timestamp": "2025-01-15T10:35:00Z",
  "uid": "new-event-001"
}
```

Command Acknowledgements:

When your system sends a command, King Kiosk acknowledges successful processing: - `calendar_upsert_appli` - Event was created/updated - `calendar_delete_applied` - Event was deleted - `calendar_bulk_applied` - Bulk replace completed

0.1.8.15.3 Smart Loop Prevention The calendar sync includes intelligent loop prevention. When King Kiosk applies a change received via MQTT, it suppresses broadcasting that same change back out. The `correlation_id` field helps track and deduplicate commands, so you can safely have bidirectional sync without echo loops.

0.1.8.16 Timers and Stopwatch

Create a stopwatch:

```
{
  "command": "stopwatch",
  "window_id": "main-stopwatch",
  "name": "Workout Timer",
  "x": 100,
  "y": 100,
  "width": 250,
  "height": 150
}
```

Create a countdown timer:

```
{
  "command": "timer_widget",
  "window_id": "countdown-1",
  "name": "Cooking Timer",
  "config": {
    "duration": 600
  }
}
```

Control timers:

```
{
  "command": "timer_control",
```

```
  "timer_id": "countdown-1",
  "action": "start"
}
```

Actions: `start`, `stop`, `pause`, `reset`

0.1.8.17 Alarmo (Security System)

Integrate with the Alarmo Home Assistant add-on. Full MQTT parity: per-mode PIN requirements, force arm (bypass open sensors), and skip exit delay.

Create an Alarmo panel:

```
{
  "command": "alarmo",
  "window_id": "alarm-panel",
  "name": "Home Security",
  "mqtt_base_topic": "alarmo",
  "entity": "alarm_control_panel.alarmo",
  "require_code_to_arm": true,
  "require_code_to_disarm": true,
  "code_length": 4,
  "available_modes": ["armed_away", "armed_home", "armed_night"],
  "x": 100,
  "y": 100,
  "width": 350,
  "height": 500
}
```

Per-mode PIN configuration — require a code for Away but not Home:

```
{
  "command": "alarmo",
  "window_id": "alarm-panel",
  "require_code_to_arm": true,
  "require_code_to_disarm": true,
  "code_required_modes": { "away": true, "home": false, "night": true }
}
```

Force arm & skip delay — the panel shows toggle chips for these options. You can also set defaults:

```
{
  "command": "alarmo",
  "window_id": "alarm-panel",
  "force": true,
  "skip_delay": true
}
```

Key parameters: `require_code` (legacy, sets both arm/disarm), `require_code_to_arm`, `require_code_to_disarm`, `code_required_modes`, `force`, `skip_delay`, `mqtt_base_topic`, `area`, `available_modes`, `code_length`.

0.1.8.18 DLNA Player

Display and control DLNA/UPnP media.

Create a DLNA player:

```
{
  "command": "dlna_player",
  "window_id": "dlna-1",
  "name": "DLNA Player",
  "x": 100,
  "y": 100,
  "width": 640,
  "height": 480
}
```

The DLNA player shows content pushed from DLNA controllers on your network (phones, computers, media servers).

0.1.8.19 PDF Viewer

Display PDF documents.

Open a PDF:

```
{
  "command": "open_pdf",
  "url": "https://example.com/document.pdf",
  "title": "User Manual",
  "window_id": "manual-pdf",
  "x": 100,
  "y": 100,
  "width": 600,
  "height": 800
}
```

0.1.8.20 Games

Because even kiosks need a break sometimes.

Launch Missile Command:

```
{
  "command": "stop_the_missiles",
  "title": "Missile Command",
  "window_id": "game-1",
  "x": 100,
  "y": 100,
  "width": 600,
  "height": 400
}
```

Control game:

```
{
  "command": "game_control",
  "window_id": "game-1",
  "action": "start"
}
```

Actions: `start`, `restart`, `stop`, `pause`, `resume`, `toggle_sound`, `set_transparent`, `set_background_mode`, `set_background_opacity`

0.1.9 8. Building Your Own Widgets

One of King Kiosk's most powerful features is the ability to create your own custom widgets using standard web technologies (HTML, CSS, and JavaScript). If you can build a web page, you can build a King Kiosk widget.

0.1.9.1 Why Build Custom Widgets?

The built-in widgets cover most common use cases, but sometimes you need something unique:

- **Branded Displays** - Create widgets that match your company's visual identity
- **Custom Data Visualization** - Display data from your specific systems in exactly the format you need
- **Interactive Kiosks** - Build check-in systems, surveys, or product configurators
- **Specialized Integrations** - Connect to APIs and services that King Kiosk doesn't support natively
- **Internal Tools** - Create dashboards for your specific workflows

Custom widgets run inside an embedded web browser window and communicate with King Kiosk through a special connection, giving you full access to send and receive messages, save data that persists across restarts, and respond to remote commands.

0.1.9.2 Quick Start

Creating a custom widget is as simple as creating an HTML file:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>My Widget</title>
```

```
  <style>
    body {
      margin: 0;
      padding: 20px;
      background: #1a1a2e;
      color: white;
      font-family: sans-serif;
      display: flex;
      align-items: center;
      justify-content: center;
      min-height: 100vh;
    }
    .value { font-size: 72px; font-weight: bold; }
  </style>
</head>
<body>
  <div class="value" id="display">--</div>

  <script>
    // Wait for the KingKiosk bridge to be ready
    window.addEventListener('kingkiosk-ready', function() {
      console.log('KingKiosk bridge ready!');

      // Listen for commands
      window.KingKiosk.onCommand(function(command, payload) {
        if (command === 'set_value') {
          document.getElementById('display').textContent = payload.value;
        }
      });
    });
  </script>
</body>
</html>
```

0.1.9.3 Adding Your Widget

You have three ways to add your custom widget:

Option 1: Host on a Web Server (Recommended for complex widgets)

Host your widget files on any web server, then add it using `open_browser`:

```
{
  "command": "open_browser",
  "url": "https://your-server.com/widgets/my-widget/",
  "title": "My Custom Widget"
}
```

Option 2: For MQTT-controllable widgets

Use the specific widget commands for each type:

```
{
  "command": "create_gauge",
  "gauge_id": "temperature",
  "gauge_type": "radial",
  "min": 0,
  "max": 100,
  "unit": "°F"
```

```
}
```

```
{
  "command": "create_chart",
  "chart_id": "metrics",
  "chart_type": "line",
  "max_points": 50
}
```

```
{
  "command": "mqtt_button",
  "action": "configure",
  "window_id": "light-switch",
  "label": "Living Room Light",
  "mode": "toggle",
  "publish_topic": "home/lights/living/set",
  "subscription_topic": "home/lights/living/state"
}
```

Option 3: Carousel for multiple items

```
{
  "command": "create_carousel",
  "window_id": "lobby-display",
  "items": [
    {"type": "image", "url": "https://example.com/slide1.jpg"},
    {"type": "image", "url": "https://example.com/slide2.jpg"}
  ],
  "config": {
    "auto_play": true,
    "interval": 10
  }
}
```

0.1.9.4 Connecting Your Widget to King Kiosk

King Kiosk provides a special `window.KingKiosk` object in your widget's web page that lets your code communicate with the app. Always wait for this connection to be ready before using it:

```
window.addEventListener('kingkiosk-ready', function() {
  // Bridge is now available - start using the API
  console.log('KingKiosk API ready!');
});
```

Available Methods:

Method	Purpose
`KingKiosk.onCommand(callback)`	Receive commands from MQTT or King Kiosk
`KingKiosk.sendCommand(cmd, payload)`	Send a command (published to MQTT)

Method	Purpose
`KingKiosk.publishTelemetry(data)`	Send sensor readings or status data to other systems
`KingKiosk.storage.get(key)`	Retrieve a stored value
`KingKiosk.storage.set(key, value)`	Store a value persistently
`KingKiosk.storage.getAll()`	Get all stored values
`KingKiosk.getWidgetInfo()`	Get widget ID and platform info

0.1.9.5 Receiving Commands

Register a callback to receive commands sent from MQTT or King Kiosk:

```
window.KingKiosk.onCommand(function(command, payload) {
  console.log('Received:', command, payload);

  switch (command) {
    case 'set_value':
      updateDisplay(payload.value);
      break;
    case 'set_color':
      document.body.style.backgroundColor = payload.color;
      break;
    case 'refresh':
      fetchLatestData();
      break;
  }
});
```

Send commands to your widget via MQTT:

Topic: `kingkiosk/{device_id}/element/{widget_id}/cmd`

```
{
  "command": "set_value",
  "payload": { "value": 42 }
}
```

0.1.9.6 Sending Data Back From Your Widget

Your widget can send commands back to King Kiosk or share data (like sensor readings or user actions) with other systems like Home Assistant.

Send a Command:

```
// Notify that a button was pressed
window.KingKiosk.sendCommand('button_pressed', { buttonId: 'start' });

// Send form data
window.KingKiosk.sendCommand('form_submitted', {
  name: 'John Doe',
  email: 'john@example.com'
});
```

Commands are published to: `kingkiosk/{device_id}/widget/{widget_id}/event`

Share Sensor Data or Status Updates:

```
// Share sensor readings with Home Assistant or other systems
window.KingKiosk.publishTelemetry({ temperature: 72.5 });

// Publish multiple metrics
window.KingKiosk.publishTelemetry({
  cpu_usage: 45.2,
  memory_used: 8192,
  uptime_seconds: 86400
});

// Periodic heartbeat
setInterval(function() {
  window.KingKiosk.publishTelemetry({
    heartbeat: true,
    timestamp: Date.now()
  });
}, 30000);
```

Telemetry is published to: `kingkiosk/{device_id}/widget/{widget_id}/telemetry`

0.1.9.7 Persistent Storage

Store data that persists across widget reloads:

```
// Store values
window.KingKiosk.storage.set('theme', 'dark');
window.KingKiosk.storage.set('lastUpdate', Date.now());
window.KingKiosk.storage.set('settings', { volume: 80, muted: false });

// Retrieve values (async)
const theme = await window.KingKiosk.storage.get('theme');
console.log('Current theme:', theme); // 'dark'

// Get all stored values
const allData = await window.KingKiosk.storage.getAll();
```

0.1.9.8 Platform Considerations

Custom widgets work across all King Kiosk platforms, but there are some considerations:

Platform	URL Loading	Inline HTML	Storage	Telemetry	JS Bridge
macOS	Yes	Yes	Yes	Yes	Full
iOS	Yes	Yes	Yes	Yes	Full
Android	Yes	Yes	Yes	Yes	Full
Windows	Yes	Yes	Yes	Yes	Full
Linux	Yes	Yes	Yes	Yes	Full
Web	Yes	Yes	Yes	Yes	Full
tvOS	Yes*	Yes*	Yes*	Yes*	Full*

tvOS (Apple TV) Special Notes:

Apple TV uses a Remote Browser server to render web content (since Apple TV cannot display web pages directly). For tvOS widgets: - Provide a `server_url` in metadata pointing to your Remote Browser server - The **full JavaScript bridge API is available** - `KingKiosk.onCommand()`, `sendCommand()`, `publishTelemetry()`, and `storage.*` all work through the Remote Browser server - Storage and telemetry operate via the Remote Browser signaling channel - Inline HTML is limited to ~2KB due to URL length restrictions (use URL loading for larger widgets)

Responsive Design Tips:

```
const info = await window.KingKiosk.getWidgetInfo();

if (info.platform === 'tvos') {
  // Larger fonts, focus-based navigation
  document.body.classList.add('tv-mode');
} else if (info.platform === 'ios' || info.platform === 'android') {
  // Touch-optimized interface
  document.body.classList.add('touch-mode');
}
```

0.1.9.9 Complete Example: Smart Thermostat

Here's a fully-featured custom widget demonstrating all API features:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Smart Thermostat</title>
  <style>
    body {
      font-family: -apple-system, sans-serif;
      background: linear-gradient(135deg, #1a1a2e, #16213e);
      color: white;
```

```
      min-height: 100vh;
      display: flex;
      flex-direction: column;
      align-items: center;
      justify-content: center;
      margin: 0;
      padding: 20px;
    }
    .temperature { font-size: 96px; font-weight: 200; }
    .unit { font-size: 36px; vertical-align: top; }
    .label { font-size: 14px; text-transform: uppercase; opacity: 0.7; }
    .controls { display: flex; gap: 20px; margin-top: 40px; }
    .btn {
      width: 60px; height: 60px; border-radius: 50%;
      border: 2px solid rgba(255,255,255,0.3);
      background: rgba(255,255,255,0.1);
      color: white; font-size: 24px; cursor: pointer;
    }
    .btn:hover { background: rgba(255,255,255,0.2); }
    .mode {
      margin-top: 20px; padding: 8px 16px;
      background: rgba(255,255,255,0.1); border-radius: 20px;
    }
    .mode.heating { background: rgba(255,100,100,0.3); }
    .mode.cooling { background: rgba(100,100,255,0.3); }
  </style>
</head>
<body>
  <div class="temperature">
    <span id="temp">72</span><span class="unit">°F</span>
  </div>
  <div class="label">Target Temperature</div>
  <div class="controls">
    <button class="btn" id="btnDown">-</button>
    <button class="btn" id="btnUp">+</button>
  </div>
  <div class="mode" id="mode">OFF</div>

  <script>
    let targetTemp = 72;
    let mode = 'off';

    window.addEventListener('kingkiosk-ready', async function() {
      // Load saved state
      const saved = await window.KingKiosk.storage.get('targetTemp');
      if (saved) targetTemp = saved;
      updateDisplay();

      // Handle incoming commands
      window.KingKiosk.onCommand(function(cmd, payload) {
        if (cmd === 'set_temperature') {
          targetTemp = payload.temperature;
          window.KingKiosk.storage.set('targetTemp', targetTemp);
          updateDisplay();
          sendTelemetry();
        }
        if (cmd === 'set_mode') {
          mode = payload.mode;
          updateDisplay();
        }
      });

      // Send initial telemetry
      sendTelemetry();
```

```
    });

    function updateDisplay() {
      document.getElementById('temp').textContent = targetTemp;
      document.getElementById('mode').textContent = mode.toUpperCase();
      document.getElementById('mode').className = 'mode ' + mode;
    }

    function sendTelemetry() {
      window.KingKiosk.publishTelemetry({
        target_temperature: targetTemp,
        mode: mode
      });
    }

    document.getElementById('btnUp').onclick = function() {
      targetTemp++;
      window.KingKiosk.storage.set('targetTemp', targetTemp);
      updateDisplay();
      sendTelemetry();
      window.KingKiosk.sendCommand('temperature_changed', {
        temperature: targetTemp, direction: 'up'
      });
    };

    document.getElementById('btnDown').onclick = function() {
      targetTemp--;
      window.KingKiosk.storage.set('targetTemp', targetTemp);
      updateDisplay();
      sendTelemetry();
      window.KingKiosk.sendCommand('temperature_changed', {
        temperature: targetTemp, direction: 'down'
      });
    };
  </script>
</body>
</html>
```

This widget: - Loads its saved state on startup - Responds to `set_temperature` and `set_mode` commands via MQTT - Publishes telemetry when values change - Sends events when the user interacts with buttons - Persists its state between reloads

0.1.9.10 Native Widget Alternatives

While building custom widgets with web technologies (HTML/CSS/JavaScript) offers maximum flexibility, King Kiosk also provides **built-in alternatives** that don't require any coding. These are perfect when you want high-performance displays without the complexity of web development, or when you're working in environments where custom coding isn't practical.

Three Native Approaches:

Approach	Best For	Complexity	Performance
Canvas Widget	Complex dashboards, diagrams, network maps	Medium	Excellent
MQTT Image	External rendering, Python/Node.js integration	Low	Good
Dynamic Native Widgets	Simple displays using built-in components	Low	Excellent

0.1.9.11 Canvas: Declarative Native Graphics

The Canvas widget is King Kiosk's most powerful native rendering system. Instead of writing code, you describe what to draw using JSON - King Kiosk handles all the rendering natively using a high-performance graphics engine.

Why Canvas? - **No web overhead** - Pure native rendering, no WebView memory or CPU cost - **Declarative** - Describe what you want, not how to draw it - **Composable** - Embed gauges, charts, and buttons inside your canvas - **Interactive** - Objects can publish MQTT messages when tapped - **Real-time updates** - Modify any object via MQTT commands

Create a Canvas:

```
{
  "command": "open_canvas",
  "window_id": "my-dashboard",
  "title": "Server Status",
  "x": 50, "y": 50,
  "width": 800, "height": 600,
  "doc": {
    "background": {
      "color": "#1a1a2e"
    },
    "objects": [],
    "connections": []
  }
}
```

0.1.9.11.1 Canvas Objects

Every visual element on a canvas is an "object" with a unique ID, position, size, and type-specific properties.

Common Object Properties:

Property	Type	Description
`object_id`	string	Unique identifier (required)
`type`	string	`node`, `text`, `shape`, `image`, `embed`, `group`
`x`, `y`	number	Position (top-left corner)
`width`, `height`	number	Size
`z_index`	int	Render order (higher = on top)
`visible`	bool	Show/hide without removing
`locked`	bool	Prevent local dragging
`rotation_deg`	number	Rotation around center
`opacity`	number	0.0 to 1.0
`style`	object	Colors, borders, etc.
`mqtt`	object	Tap-to-publish behavior

Text Objects:

```
{
  "object_id": "title",
  "type": "text",
  "text": "System Status",
  "x": 50, "y": 30,
  "style": {
    "color": "#ffffff",
    "font_size_px": 28,
    "font_weight": "bold"
  }
}
```

Shape Objects:

```
{
  "object_id": "status-bg",
  "type": "shape",
  "shape": "round_rect",
  "x": 50, "y": 80,
  "width": 200, "height": 100,
  "style": {
    "fill_color": "#2d3436",
    "stroke_color": "#00ff00",
    "stroke_width_px": 2,
    "corner_radius_px": 12
  }
}
```

Shape types: `rect`, `round_rect`, `circle`, `line`

Image Objects:

```
{
  "object_id": "server-icon",
  "type": "image",
  "x": 70, "y": 100,
  "width": 64, "height": 64,
  "source": {
    "url": "https://example.com/icons/server.png"
  },
  "fit": "contain",
  "opacity": 0.9
}
```

Node Objects (for diagrams):

```
{
  "object_id": "server-1",
  "type": "node",
  "x": 100, "y": 200,
  "width": 120, "height": 80,
  "label": "Web Server",
  "icon": { "set": "material", "name": "dns" },
  "style": {
    "color": "#3498db",
    "border_color": "#2980b9"
  },
  "ports": [
    { "port_id": "out", "pos": { "edge": "east", "t": 0.5 } }
  ]
}
```

Embedded Widgets:

This is where Canvas becomes incredibly powerful - embed fully functional native widgets inside your canvas:

```
{
  "object_id": "cpu-gauge",
  "type": "embed",
  "x": 300, "y": 150,
  "width": 150, "height": 150,
  "embed": {
    "widget_type": "gauge",
    "id": "cpu-gauge-widget",
    "config": {
      "gauge_type": "radial",
      "min": 0,
      "max": 100,
      "value": 45,
      "unit": "%",
      "title": "CPU",
      "thresholds": [
        { "value": 70, "color": "#f1c40f" },
        { "value": 90, "color": "#e74c3c" }
      ]
    }
  }
}
```

Embeddable widgets: `gauge`, `chart`, `mqtt_button`, `mqtt_action_status`

0.1.9.11.2 Connections (Lines Between Objects)

Connect objects with lines - perfect for network diagrams, flowcharts, and system maps:

```
{
  "connections": [
    {
      "connection_id": "link-1",
      "from": { "object_id": "server-1", "port_id": "out" },
      "to": { "object_id": "database-1", "port_id": "in" },
      "style": {
        "color": "#00ff00",
        "width_px": 2,
        "arrow": "end"
      }
    }
  ]
}
```

Connection styles: - `arrow`: `none`, `end`, `both` - `dash`: Array of dash/gap lengths, e.g., `[5, 3]` - `animated`: **`true`** for flowing animation effect

0.1.9.11.3 Interactive Canvas Objects

Make objects respond to taps by publishing MQTT messages:

```
{
  "object_id": "restart-btn",
  "type": "shape",
  "shape": "round_rect",
  "x": 600, "y": 500,
  "width": 150, "height": 50,
  "style": {
    "fill_color": "#e74c3c",
    "corner_radius_px": 8
  },
  "mqtt": {
    "publish": {
      "topic": "home/server/command",
      "payload": { "action": "restart" }
    }
  }
}
```

0.1.9.11.4 Updating Canvas Objects via MQTT

After creating a canvas, update individual objects without recreating the entire canvas:

Update an object:

```
{
  "command": "update_objects",
  "objects": [
    {
```

```
      "object_id": "cpu-gauge",
      "embed": {
        "config": {
          "value": 78
        }
      }
    }
  ]
}
```

Change text:

```
{
  "command": "update_objects",
  "objects": [
    {
      "object_id": "status-text",
      "text": "All Systems Operational",
      "style": {
        "color": "#00ff00"
      }
    }
  ]
}
```

Add new objects:

```
{
  "command": "add_objects",
  "objects": [
    {
      "object_id": "alert-banner",
      "type": "text",
      "text": "WARNING: High CPU Usage",
      "x": 50, "y": 550,
      "style": { "color": "#e74c3c", "font_size_px": 18 }
    }
  ]
}
```

Remove objects:

```
{
  "command": "remove_objects",
  "object_ids": ["alert-banner"]
}
```

0.1.9.11.5 Complete Canvas Example: Server Dashboard

Here's a full example creating a server monitoring dashboard:

```
{
  "command": "open_canvas",
  "window_id": "server-dashboard",
  "title": "Infrastructure Status",
  "width": 900, "height": 700,
  "doc": {
    "background": {
      "color": "#0f0f23"
    },
```

```
    "objects": [
      {
        "object_id": "header",
        "type": "text",
        "text": "Infrastructure Monitor",
        "x": 30, "y": 20,
        "style": { "color": "#ffffff", "font_size_px": 32, "font_weight": "bold" }
      },
      {
        "object_id": "web-server",
        "type": "node",
        "x": 100, "y": 150,
        "width": 140, "height": 90,
        "label": "Web Server",
        "icon": { "set": "material", "name": "public" },
        "style": { "color": "#3498db" },
        "ports": [{ "port_id": "out", "pos": { "edge": "east", "t": 0.5 } }]
      },
      {
        "object_id": "api-server",
        "type": "node",
        "x": 380, "y": 150,
        "width": 140, "height": 90,
        "label": "API Server",
        "icon": { "set": "material", "name": "dns" },
        "style": { "color": "#9b59b6" },
        "ports": [
          { "port_id": "in", "pos": { "edge": "west", "t": 0.5 } },
          { "port_id": "out", "pos": { "edge": "east", "t": 0.5 } }
        ]
      },
      {
        "object_id": "database",
        "type": "node",
        "x": 660, "y": 150,
        "width": 140, "height": 90,
        "label": "Database",
        "icon": { "set": "material", "name": "storage" },
        "style": { "color": "#27ae60" },
        "ports": [{ "port_id": "in", "pos": { "edge": "west", "t": 0.5 } }]
      },
      {
        "object_id": "cpu-gauge",
        "type": "embed",
        "x": 100, "y": 350,
        "width": 180, "height": 180,
        "embed": {
          "widget_type": "gauge",
          "id": "cpu",
          "config": {
            "gauge_type": "radial",
            "min": 0, "max": 100,
            "value": 42,
            "unit": "%",
            "title": "CPU Usage",
            "thresholds": [
              { "value": 70, "color": "#f39c12" },
              { "value": 90, "color": "#e74c3c" }
            ]
          }
        }
      },
      {
        "object_id": "memory-gauge",
```

```
        "type": "embed",
        "x": 360, "y": 350,
        "width": 180, "height": 180,
        "embed": {
          "widget_type": "gauge",
          "id": "memory",
          "config": {
            "gauge_type": "radial",
            "min": 0, "max": 64,
            "value": 28,
            "unit": "GB",
            "title": "Memory",
            "thresholds": [
              { "value": 48, "color": "#f39c12" },
              { "value": 58, "color": "#e74c3c" }
            ]
          }
        }
      },
      {
        "object_id": "requests-chart",
        "type": "embed",
        "x": 620, "y": 350,
        "width": 250, "height": 180,
        "embed": {
          "widget_type": "chart",
          "id": "requests",
          "config": {
            "chart_type": "line",
            "title": "Requests/sec",
            "max_points": 30
          }
        }
      },
      {
        "object_id": "status-text",
        "type": "text",
        "text": "All Systems Operational",
        "x": 30, "y": 600,
        "style": { "color": "#2ecc71", "font_size_px": 24 }
      },
      {
        "object_id": "restart-btn",
        "type": "embed",
        "x": 700, "y": 580,
        "width": 160, "height": 60,
        "embed": {
          "widget_type": "mqtt_button",
          "id": "restart-all",
          "config": {
            "label": "Restart All",
            "publish_topic": "infrastructure/command",
            "publish_payload": { "action": "restart_all" },
            "style": "danger"
          }
        }
      }
    ],
    "connections": [
      {
        "connection_id": "web-to-api",
        "from": { "object_id": "web-server", "port_id": "out" },
        "to": { "object_id": "api-server", "port_id": "in" },
        "style": { "color": "#2ecc71", "width_px": 3, "animated": true }
```

```
      },
      {
        "connection_id": "api-to-db",
        "from": { "object_id": "api-server", "port_id": "out" },
        "to": { "object_id": "database", "port_id": "in" },
        "style": { "color": "#2ecc71", "width_px": 3, "animated": true }
      }
    ]
  }
}
```

0.1.9.12 MQTT Image: External Rendering

If you have existing tools that generate images (Python scripts, Node.js apps, image processing pipelines), the MQTT Image widget lets you stream rendered frames directly to King Kiosk.

Why MQTT Image? - **Use any language** - Python, Node.js, Go, Rust - anything that can publish to MQTT - **Leverage existing libraries** - PIL/Pillow, Cairo, ImageMagick, matplotlib, D3.js (server-side) - **Pre-rendered content** - Complex visualizations rendered on powerful hardware - **Integration flexibility** - Connect to systems that already generate images

Create an MQTT Image widget:

```
{
  "command": "mqtt_image",
  "window_id": "external-render",
  "mqtt_topic": "myapp/rendered/frame",
  "is_base64": true,
  "title": "External Dashboard",
  "x": 100, "y": 100,
  "width": 800, "height": 600
}
```

Publish images from Python:

```
import paho.mqtt.client as mqtt
from PIL import Image, ImageDraw, ImageFont
import base64
import io

def render_dashboard(cpu, memory, status):
    # Create image
    img = Image.new('RGB', (800, 600), color='#1a1a2e')
    draw = ImageDraw.Draw(img)

    # Draw content
    draw.text((50, 30), "System Dashboard", fill='white')
    draw.text((50, 100), f"CPU: {cpu}%", fill='#3498db')
    draw.text((50, 150), f"Memory: {memory}GB", fill='#9b59b6')
    draw.text((50, 200), f"Status: {status}", fill='#2ecc71')

    # Convert to base64
    buffer = io.BytesIO()
    img.save(buffer, format='PNG')
    return base64.b64encode(buffer.getvalue()).decode()

```

```
# Connect and publish
client = mqtt.Client()
client.connect("your-broker", 1883)

# Render and send
frame = render_dashboard(cpu=45, memory=28, status="Online")
client.publish("myapp/rendered/frame", frame)
```

Publish from Node.js:

```
const mqtt = require('mqtt');
const { createCanvas } = require('canvas');

const client = mqtt.connect('mqtt://your-broker');

function renderFrame(data) {
  const canvas = createCanvas(800, 600);
  const ctx = canvas.getContext('2d');

  // Dark background
  ctx.fillStyle = '#1a1a2e';
  ctx.fillRect(0, 0, 800, 600);

  // Draw content
  ctx.fillStyle = 'white';
  ctx.font = '32px sans-serif';
  ctx.fillText('System Dashboard', 50, 50);

  ctx.fillStyle = '#3498db';
  ctx.font = '24px sans-serif';
  ctx.fillText(`CPU: ${data.cpu}%`, 50, 120);

  // Return as base64
  return canvas.toBuffer('image/png').toString('base64');
}

// Publish every 5 seconds
setInterval(() => {
  const frame = renderFrame({ cpu: Math.round(Math.random() * 100) });
  client.publish('myapp/rendered/frame', frame);
}, 5000);
```

Update interval:

For animated displays, the widget can poll for new images:

```
{
  "command": "mqtt_image",
  "mqtt_topic": "myapp/rendered/frame",
  "is_base64": true,
  "update_interval": 1000
}
```

0.1.9.13 Dynamic Native Widgets

Don't overlook the power of King Kiosk's built-in widgets. Gauges, charts, and animated text are highly configurable and can be dynamically updated via MQTT - often eliminating the need for custom development entirely.

Dynamic Gauge Updates:

Create a gauge once, then update its value from any system:

```
{
  "command": "create_gauge",
  "window_id": "temp-gauge",
  "gauge_type": "radial",
  "title": "Temperature",
  "min": 0, "max": 100,
  "value": 72,
  "unit": "°F",
  "thresholds": [
    { "value": 80, "color": "#f39c12" },
    { "value": 90, "color": "#e74c3c" }
  ]
}
```

Update from Home Assistant, Node-RED, or any MQTT client:

```
{
  "command": "set_gauge_value",
  "gauge_id": "temp-gauge",
  "value": 85
}
```

Dynamic Charts:

Create real-time charts that update as data arrives:

```
{
  "command": "create_chart",
  "window_id": "sensor-chart",
  "chart_type": "line",
  "title": "Temperature History",
  "max_points": 60,
  "mqtt_topic_prefix": "kingkiosk"
}
```

Append data points over time:

```
{
  "command": "append_chart_data",
  "chart_id": "sensor-chart",
  "value": 72
}
```

Dynamic Animated Text:

Create eye-catching displays without writing any code:

```
{
  "command": "open_animated_text",
  "window_id": "alert-display",
  "preset": "tickerMarquee",
  "text": "Welcome to King Kiosk!",
  "layout": {
    "alignment": "center",
    "vertical_alignment": "middle"
  },
```

```
  "background_mode": "gradient",
  "background_color": "#1a1a2e"
}
```

Update the text dynamically via element command: Topic: `kingkiosk/{device_id}/element/alert-display/cmd`

```
{
  "command": "set_text",
  "text": "ALERT: Server maintenance at 2:00 PM"
}
```

When to Use Each Approach:

Scenario	Recommended Approach
Complex dashboard with multiple data sources	Canvas with embedded widgets
Network topology / system diagram	Canvas with nodes and connections
Already have Python/Node rendering code	MQTT Image
Simple numeric displays	Dynamic Gauges
Time-series data visualization	Dynamic Charts
Scrolling announcements	Animated Text
Full interactivity with custom logic	Custom web widget (HTML/JS)

0.1.9.14 OS Widgets (Home Screen Widgets)

King Kiosk can create native widgets on your Android home screen or iOS/iPadOS home screen and lock screen. These widgets update automatically and work even when the King Kiosk app isn't open — perfect for at-a-glance information like temperature sensors, alarm status, or quick-action buttons.

What are OS Widgets?

OS widgets are small visual elements that appear on your phone or tablet's home screen (Android) or home/lock screen (iOS). Unlike the widgets *inside* King Kiosk (which appear in windows on the kiosk display), OS widgets are native to your device's operating system.

Think of it this way: - **King Kiosk window widgets** = Displayed inside the King Kiosk app on your wall-mounted tablet - **OS widgets** = Small info cards on your phone's home screen that you can glance at without opening any app

Why Use OS Widgets?

- **Always visible** — No need to open King Kiosk to check a sensor value
- **Updates automatically** — Widgets refresh their data from MQTT on their own schedule
- **Battery efficient** — Widgets use minimal power compared to running the full app
- **Quick actions** — Tap a button widget to control your home without opening the app
- **Multiple widgets** — Create as many as you need for different sensors or controls

Supported Widget Types:

Widget Type	What It Shows	Best For
Gauge	Numeric value with optional zones/thresholds	Temperature, humidity, battery levels
Chart	Small line chart showing value history	Temperature trends, power usage
Weather	Current conditions and forecast	Current weather, daily forecast
Alarmo	Security system status and arm/disarm controls	Home security, alarm status
MQTT Button	Toggle button with on/off states	Light switches, scene triggers
Sensor	Simple value display	Any numeric sensor value
Canvas	Visual diagram snapshot	System diagrams, network maps
Clock	Current time display	Quick time check

Creating an OS Widget:

To create a native OS widget, add the `os_widget: true` parameter when creating a widget via MQTT. The widget will be created both inside King Kiosk (if you want it there) and as a native OS widget.

Example: Temperature Gauge Widget

```
{
  "command": "create_gauge",
  "window_id": "living-room-temp",
  "gauge_type": "radial",
  "title": "Living Room",
  "min": 50,
  "max": 90,
  "value": 72,
  "unit": "°F",
  "os_widget": true,
  "mqtt_topic": "homeassistant/sensor/living_room_temperature/state",
  "json_field": "temperature"
}
```

What happens: 1. King Kiosk creates the gauge inside the app 2. King Kiosk also registers a native OS widget with your device 3. You can now add this widget to your home screen just like any other widget 4. The widget automatically subscribes to the MQTT topic and updates when new values arrive

Example: Security System Widget (Alarmo)

```
{
  "command": "alarmo_widget",
  "window_id": "home-alarm",
  "mqtt_base_topic": "alarmo",
  "require_code": true,
  "code_length": 4,
  "available_modes": ["armed_away", "armed_home", "armed_night"],
  "os_widget": true
}
```

The widget shows alarm status and provides arm/disarm buttons. Tapping the widget opens quick controls without opening the full app.

Example: MQTT Button Widget

```
{
  "command": "mqtt_button",
  "window_id": "porch-light",
  "label": "Porch Light",
  "mode": "toggle",
  "publish_topic": "home/lights/porch/set",
  "subscription_topic": "home/lights/porch/state",
  "icon": "light_bulb",
  "color_on": "0xFFFFC107",
  "color_off": "0xFF757575",
  "os_widget": true
}
```

Creates a home screen button that toggles your porch light. The button color updates based on the light's state.

Adding Widgets to Your Home Screen:

On Android: 1. Long-press on your home screen 2. Tap "Widgets" 3. Scroll to "King Kiosk" 4. Drag the widget type you created to your home screen 5. Select which widget (if you have multiple)

On iOS/iPadOS: 1. Long-press on your home screen 2. Tap the “+” button in the top corner 3. Search for “King Kiosk” 4. Choose the widget size and type 5. Add to home screen or lock screen 6. Edit the widget to select which data source

How Updates Work:

OS widgets connect to MQTT independently of the main King Kiosk app. They use a lightweight MQTT client built into the widget extension:

- **Automatic updates** — Widgets subscribe to their configured MQTT topic
- **Efficient refresh** — iOS widgets refresh every 15-60 minutes, Android widgets can refresh more frequently
- **Cached values** — King Kiosk also caches the latest values in shared storage for instant widget display

Platform Differences:

Feature	Android	iOS
Widget placement	Home screen only	Home screen, lock screen, Today view
Update frequency	Every few minutes	Every 15-60 minutes (OS controlled)
Interactive buttons	Yes	Yes (iOS 17+)
Resize	Yes	Limited sizes

Privacy & Battery:

- Widgets only connect to MQTT when refreshing — not continuously
- MQTT credentials are stored securely in your device’s keychain
- Widgets use minimal battery (less than 1% per day typically)
- Widgets can be removed at any time without affecting King Kiosk

Removing OS Widgets:

To remove a widget from your home screen, simply delete it like any other widget (long-press and remove). To fully unregister it from King Kiosk, send a remove command:

```
{
  "command": "close_window",
  "window_id": "living-room-temp"
}
```

This removes both the in-app widget and the OS widget registration.

Troubleshooting:

- **Widget not appearing in widget list?** Make sure you created it with `os_widget: true` and the device supports widgets
- **Widget showing old data?** The widget updates on its own schedule. iOS restricts update frequency to preserve battery
- **Widget showing "No data"?** Check that MQTT credentials are configured and the topic has published at least once
- **Button widget not responding?** Verify the publish topic and payload are correct in your MQTT broker

---

0.1.10 9. Home Assistant Integration

King Kiosk is designed to work seamlessly with Home Assistant. Basic auto-discovery setup is covered in Getting Started - once enabled, your displays automatically appear as devices in Home Assistant.

This section covers advanced integration patterns for power users.

0.1.10.1 Creating Custom Sensors

Monitor widget states in Home Assistant:

```yaml
mqtt:
  sensor:
    - name: "Living Room Clock Mode"
      state_topic: "kingkiosk/living_room/element/clock-1/state"
      value_template: "{{ value_json.mode }}"
      json_attributes_topic: "kingkiosk/living_room/element/clock-1/state"
```

0.1.10.2 Controlling King Kiosk from Home Assistant

Create buttons to control your displays:

```yaml
mqtt:
  button:
    - name: "Living Room Morning Mode"
      command_topic: "kingkiosk/living_room/system/cmd"
      payload_press: '{"command": "load_screen_state", "name": "Morning Dashboard"}'

    - name: "Toggle Clock Mode"
      command_topic: "kingkiosk/living_room/element/clock-1/cmd"
      payload_press: '{"command": "toggle_mode"}'
```

Use automations to control displays based on events:

```yaml
automation:
  - alias: "Turn on lobby display when motion detected"
    trigger:
      - platform: state
        entity_id: binary_sensor.lobby_motion
        to: "on"
    action:
      - service: mqtt.publish
        data:
          topic: "kingkiosk/lobby_display/system/cmd"
          payload: '{"command": "set_brightness", "value": 1.0}'
```

0.1.11 10. Tips and Best Practices

0.1.11.1 Performance Optimization

For smooth operation: - Limit the number of simultaneous video streams - Use appropriate image sizes (don't load 4K images for thumbnail displays) - Close unused widgets to free memory - Use the `reset_media` command if playback becomes sluggish

For battery-powered devices: - Reduce screen brightness - Extend carousel intervals - Disable unnecessary sensors - Use screen schedules to dim displays during off-hours

0.1.11.2 Network Considerations

For reliable MQTT: - Use a local broker when possible for lowest latency - Enable SSL for secure communications over the internet - For critical commands, consider using higher reliability settings (QoS - Quality of Service levels) to ensure delivery - Monitor your server's connection count if managing many displays

For media streaming: - Ensure adequate bandwidth for video content - Use local media servers when possible - Consider CDN-hosted content for distributed deployments

0.1.11.3 Troubleshooting Common Issues

MQTT Not Connecting: 1. Verify broker host and port in settings 2. Check username and password 3. Ensure the device can reach the broker (firewall rules) 4. Try disabling SSL to rule out certificate issues

Widget Not Responding to Commands: 1. Verify the widget ID in your command matches the ID you gave when creating the widget 2. Check for typos in the command name 3. Ensure the widget is actually created and visible on screen 4. Check the response messages for error details

Display Seems Slow: 1. Close unused windows 2. Check network connectivity 3. Reduce video quality or count 4. Use the `reset_media` command 5. Restart the app if necessary

Layout Not Saving: 1. Ensure you have permission to write to storage 2. Check available storage space 3. Try a different state name 4. Export and re-import as a workaround

---

0.1.12 11. Apple TV Experience

King Kiosk on Apple TV isn't just a port - it's a completely reimagined experience built from the ground up for the living room. While other digital signage apps offer basic single-content displays on tvOS, King Kiosk brings its full multi-window tiling system to your television, controlled entirely with the Siri Remote.

There's simply nothing else like it on the Apple TV platform.

0.1.12.1 A Revolutionary Window System for Television

Traditional TV apps show one thing at a time. King Kiosk shatters that limitation. On your Apple TV, you can simultaneously display:

- A clock in one corner
- Weather forecast in another
- A live security camera feed
- Your Home Assistant dashboard
- A news ticker
- And more - all visible at once, each in its own resizable tile

Two Layout Modes:

Floating Mode - Windows float freely on screen. Position them anywhere, resize them to any dimension, overlap them as needed. Perfect for creative dashboard layouts where you want precise control over every pixel.

Tiling Mode - Inspired by the i3 window manager beloved by power users, tiling mode automatically arranges your windows in a clean, non-overlapping grid. Add a new window and it intelligently splits the space. Remove one and the others expand to fill the gap. No manual arrangement needed.

Switch between modes anytime with the MQTT command:

```
{
  "command": "window_mode",
  "mode": "tiling"
}
```

Or toggle with `"mode": "toggle"` to flip between floating and tiling.

0.1.12.2 Controlling with the Siri Remote

The Siri Remote becomes your complete control center for King Kiosk. Here's what each button does:

Button	Function
D-pad (Up/Down/Left/Right)	Navigate between windows using spatial focus
Select (Center Click)	Interact with the focused window
Play/Pause	Enter/exit Move Mode (floating layout only)
Menu	Exit current mode, go back, or close menus
TV/Home	Standard tvOS home behavior

The Focus Ring

When navigating between windows, a glowing focus ring shows which tile is currently selected. This ring is smart:

- **Auto-hides** after 5 seconds of inactivity for a cleaner display
- **Instantly reappears** when you press any remote button
- **Changes color** to indicate different modes (white for normal, cyan for interactive)
- **Shows mode indicators** like "MOVE ARROWS" when in move mode

0.1.12.3 Understanding Focus Navigation

Unlike touch screens where you tap directly on what you want, Apple TV uses spatial navigation. When you press a direction on the D-pad, King Kiosk uses intelligent proximity detection to determine which window should receive focus next.

How it works:

1. The system calculates the center point of each visible window

2. When you press Right, it filters to windows with centers to the right of your current focus
3. It scores candidates by distance (closer = better)
4. The closest window in that direction receives focus

This feels natural and intuitive - press toward the window you want, and focus moves there.

Edge Cases:

If no window exists in the direction you pressed, focus either: - Stays on the current window, or - Cycles through windows by z-order (if you keep pressing)

This means you can never get "stuck" - keep pressing a direction and you'll eventually cycle through all windows.

0.1.12.4 Move Mode: Rearranging Your Layout

In floating mode, you can reposition windows using just the remote:

1. **Focus on the window** you want to move
2. **Press Play/Pause** to enter Move Mode
3. The focus ring changes and shows **"MOVE ARROWS"**
4. **Use the D-pad** to nudge the window (40 pixels per press)
5. **Press Play/Pause again** (or Menu) to exit Move Mode

While in Move Mode: - The moving window stays focused (you can't accidentally navigate away) - Other windows ignore D-pad input - Smooth animation makes positioning feel responsive

Note: Move Mode is only available in floating layout. In tiling mode, window positions are automatic.

0.1.12.5 Interactive Mode: Deep Widget Control

Some widgets need more than simple selection - they need to capture D-pad input for their own purposes. King Kiosk calls this **Interactive Mode** or **Input Capture**.

Widgets that support Interactive Mode:

Widget	D-pad Behavior in Interactive Mode
PDF Viewer	Left/Right = Previous/Next page
Media Player	Left/Right = Skip back/forward 10 seconds

Widget	D-pad Behavior in Interactive Mode
DLNA Player	Full playback controls
Remote Browser	Navigate web pages
Games	Full game controls
Custom WebView	Widget-defined controls

Entering Interactive Mode:

Select a widget that supports it, and it automatically captures D-pad input. The focus ring turns **cyan with a glow effect** and displays **"INTERACTIVE"** to indicate the mode change.

Exiting Interactive Mode:

Press the **Menu button** to release input capture and return to normal window navigation.

0.1.12.6 MQTT Control for Apple TV

Every feature of King Kiosk on Apple TV is controllable via MQTT - perfect for Home Assistant automations, Node-RED flows, or custom control systems.

All standard window commands work identically on Apple TV:

```
{
  "command": "open_clock",
  "window_id": "living-room-clock",
  "mode": "analog",
  "theme": "dark",
  "x": 50,
  "y": 50,
  "width": 300,
  "height": 300
}
```

```
{
  "command": "create_carousel",
  "window_id": "family-photos",
  "items": [
    {"type": "image", "url": "https://example.com/photo1.jpg"},
    {"type": "image", "url": "https://example.com/photo2.jpg"}
  ],
  "config": {
    "auto_play": true,
    "interval": 10
  }
}
```

Apple TV publishes its state too:

Subscribe to `kingkiosk/{device_id}/windows` to receive real-time updates about: - Window list and positions - Focus state changes - Mode changes (floating/tiling) - Window events (created, closed, moved, resized)

0.1.12.7 Remote Browser: Full Web on Your TV

Apple TV has no built-in ability to display web pages directly, which traditionally limits what TV apps can show. King Kiosk solves this brilliantly with the **Remote Browser** feature.

How it works:

1. A Remote Browser server (which you set up) renders web pages using a full browser engine
2. The rendered output streams to your Apple TV in real-time as video
3. Your remote control inputs are sent back to the server so you can interact with web pages

The result? Full, interactive web browsing on your TV - something Apple doesn't natively allow.

Setting up Remote Browser:

```
{
  "command": "create_remote_browser",
  "window_id": "ha-dashboard",
  "initial_url": "http://homeassistant.local:8123",
  "auto_connect": true
}
```

What you can display: - Home Assistant Lovelace dashboards - Grafana monitoring panels - Node-RED dashboards - Any web application - Custom HTML widgets (with full JavaScript)

Platform Convenience:

On Apple TV, the standard browser commands automatically use Remote Browser: - `open_browser` -> Creates a Remote Browser - `open_web` -> Creates a Remote Browser - `open_simple_web` -> Creates a Remote Browser

This means your automations work across all platforms without modification. Send the same `open_browser` command to an iPad and an Apple TV - each uses its optimal rendering method automatically.

Custom Widgets via Remote Browser:

The full Custom Widget SDK works on Apple TV through Remote Browser: - `KingKiosk.onCommand()` receives MQTT commands - `KingKiosk.sendCommand()` sends commands back - `KingKiosk.publishTelemetry()` publishes sensor data - `KingKiosk.storage.*` persists widget state

Build your custom widget once, deploy it everywhere - including Apple TV.

0.1.12.8 Apple TV Tips and Best Practices

Layout Design for 10-Foot UI: - Use larger fonts (minimum 24pt for body text) - High contrast colors work best on TVs - Leave generous margins - TV overscan can clip edges - Test your layout from across the room

Performance on Apple TV: - Video playback is hardware-accelerated and smooth - Limit simultaneous video streams to 2-3 for best performance - Remote Browser works best on local network (low latency) - Complex web dashboards may need a powerful Remote Browser server

Ideal Use Cases: - Smart home control center in the living room - Digital photo frame with weather and time overlay - Security camera monitor with multiple feeds - Meeting room display with calendar and room status - Retail digital signage with multiple content zones

Automation Ideas:

"Movie mode" when Apple TV starts playing:

```yaml
automation:
  - alias: "King Kiosk Movie Mode"
    trigger:
      - platform: state
        entity_id: media_player.apple_tv
        to: 'playing'
    action:
      - service: mqtt.publish
        data:
          topic: "kingkiosk/living_room_tv/system/cmd"
          payload: '{"command": "close_all_windows"}'
```

Show security cameras when doorbell rings:

```yaml
automation:
  - alias: "Doorbell Camera on TV"
    trigger:
      - platform: state
        entity_id: binary_sensor.doorbell
        to: 'on'
    action:
      - service: mqtt.publish
        data:
          topic: "kingkiosk/living_room_tv/system/cmd"
          payload: |
            {
              "command": "play_media",
              "window_id": "doorbell-cam",
              "url": "rtsp://camera.local/doorbell",
              "width": 640,
              "height": 480,
              "x": 100,
              "y": 100
            }
```

0.1.13 12. Feature Server

The Feature Server is the powerhouse behind King Kiosk's most advanced capabilities. It provides features that go beyond what a single tablet can do on its own — things like whole-house intercom, running a browser on AppleTV, streaming your tablet's camera to Go2RTC, Frigate, Home Assistant, and bridging your entire Home Assistant enabled smart home to MQTT.

Think of it as the brain that connects all your King Kiosk devices together and ties them into your broader smart home ecosystem.

0.1.13.1 What is the Feature Server?

The Feature Server is a set of services built into the KingKiosk Core server. You don't install it separately — it's already there. Each feature can be enabled or disabled independently based on what you need. The four main features are:

Feature	What It Does
Intercom & Broadcast	Send live video and audio from one kiosk to all others
Remote Browser	Run a full Chromium browser on your AppleTV
RTSP Camera Export	Turn your tablet into a security camera visible in Home Assistant, Frigate, etc.
Home Assistant MQTT Bridge	Control all your Home Assistant devices through MQTT
High-Quality TTS (Piper)	Natural-sounding text-to-speech with 100+ voices in 30+ languages

Important: When you enable the Feature Server, local on-device object/person detection and facial recognition are disabled. The camera stream is sent to the Feature Server instead of being processed locally. If you need local AI detection, keep the Feature Server disabled.

0.1.13.1.1 Automatic Feature Server Discovery King Kiosk can automatically find and connect to your Feature Server without you having to manually enter its address. This makes setup easier, especially if you have multiple devices or if your server's IP address changes.

How It Works:

When your KingKiosk Core server starts up, it publishes its location on the MQTT topic `kingkiosk/core3/api`. This message contains the server's URL and version information. Every King Kiosk

client device listens to this topic and automatically updates its connection settings when it sees this announcement.

Think of it like your server saying "Hey everyone, I'm here at this address!" and all your devices responding "Got it, connecting now!"

Setting Up Autodiscovery:

1. **Make sure MQTT is working** - Both your Feature Server and your King Kiosk devices must be connected to the same MQTT broker
2. **Start your Feature Server** - When it starts, it automatically announces itself
3. **Your devices connect automatically** - No manual configuration needed

Manual Override:

Sometimes you might want to manually set the Feature Server address (for testing, using a different server, etc.). In the King Kiosk settings under **Feature Server**, you can:

1. Enter a specific server address (like `http://192.168.1.100:3000`)
2. Enable **Manual Override Lock** - This prevents autodiscovery from changing your manual setting
3. When locked, the device will ignore autodiscovery announcements and only use your manually configured address

To re-enable autodiscovery, simply disable the manual override lock. The device will then listen for server announcements again.

Troubleshooting:

- **Device not connecting?** Check that both device and server are connected to the same MQTT broker
- **Server address keeps changing?** Enable manual override lock to force a specific address
- **Want to use a different server temporarily?** Enter the address manually and lock it, then unlock when done

---

0.1.13.2 High-Quality Text-to-Speech (Piper)

When the Feature Server is connected, TTS commands automatically use the Piper speech synthesis engine instead of the device's built-in TTS. Piper produces natural-sounding speech with a large library of voices across many languages.

0.1.13.2.1 How It Works

You don't need to change any of your existing TTS automations. The same MQTT commands work in both modes:

- **Feature Server connected** — Speech is synthesized on the server using Piper, then played back as audio on the device. Higher quality, more voices, consistent across all devices.
- **Feature Server disconnected** — Falls back to the device's built-in TTS engine (FlutterTts or AVSpeechSynthesizer). Works offline.

0.1.13.2.2 Choosing a Voice

First, discover what voices are available:

```
{
  "command": "tts",
  "action": "getVoices",
  "language": "en_US",
  "response_topic": "kingkiosk/my_device/system/response"
}
```

The response includes voice details like `voiceId`, `quality`, and whether the voice is already installed. Then use a voice in your speak commands:

```
{
  "command": "tts",
  "text": "Dinner is ready!",
  "voice": "en_US-lessac-medium"
}
```

0.1.13.2.3 Installing Voices

If a voice isn't installed yet, you can pull it:

```
{
  "command": "tts",
  "action": "voice_pull",
  "voice": "en_US-lessac-medium"
}
```

If `auto_download` is enabled in the server config, voices are downloaded automatically on first use.

0.1.13.2.4 Home Assistant Automation Example

```
automation:
  - alias: "Announce Front Door"
    trigger:
      - platform: state
        entity_id: binary_sensor.front_door
        to: 'on'
    action:
      - service: mqtt.publish
        data:
          topic: "kingkiosk/kitchen_tablet/system/cmd"
          payload: >
            {
              "command": "tts",
              "text": "The front door just opened.",
```

```
                "voice": "en_US-lessac-medium",
                "volume": 0.8
            }
```

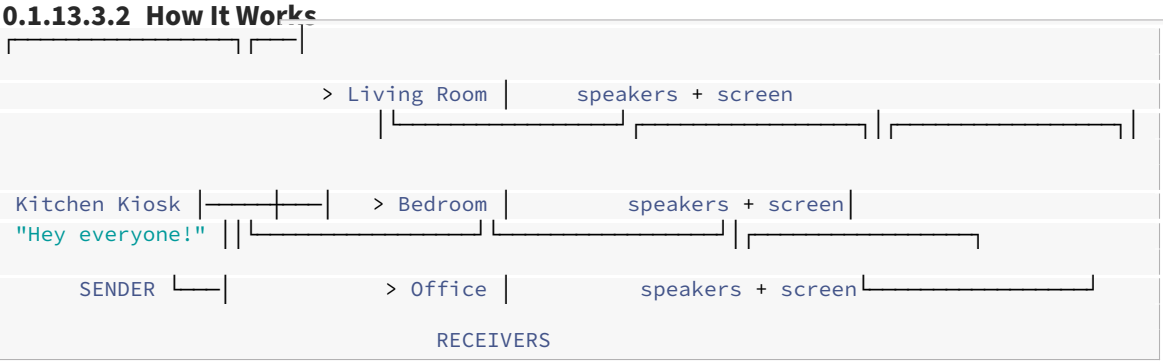
0.1.13.3 Intercom & Broadcast

Intercom lets you broadcast live audio and video from one King Kiosk device to all other enabled devices. Walk up to any kiosk, press a button, and instantly appear on every other screen — like a whole-house intercom system using your existing tablets.

0.1.13.3.1 Why Use It? Instead of yelling across the house, making phone calls for a quick message, or relying on one-way smart speaker announcements, King Kiosk Intercom gives you:

- **One-button broadcasting** — Start instantly from any kiosk
- **Video and audio** — They see and hear you
- **Every room at once** — All enabled kiosks receive the broadcast
- **No extra app needed** — It's built into your wall-mounted tablets

0.1.13.3.2 How It Works



1. You start a broadcast from one kiosk
2. All other enabled kiosks receive your video and audio
3. You see yourself on your own screen as a preview (but don't hear yourself — no echo)
4. You stop the broadcast when you're done

0.1.13.3.3 Getting Started **Enable intercom on each device** — Before a kiosk can participate, you need to enable intercom in the Feature Server settings:

1. Open **Settings** on your King Kiosk device

2. Go to the **Feature Server** section
3. Make sure the Feature Server is connected (enter your server hostname/IP if needed)
4. Toggle **Enable intercom** on

Do this for each kiosk you want to be part of the intercom system. The device will automatically join the default intercom group.

0.1.13.3.4 Adding an Intercom Button to Your Kiosk

The easiest way to use intercom is to add a dedicated toggle button directly on your kiosk screen. This creates a tap-to-talk button that turns green when broadcasting and gray when idle:

```
{
  "command": "mqtt_button",
  "action": "configure",
  "window_id": "intercom_btn",
  "mode": "icon_button",
  "publish_topic": "kingkiosk/{deviceId}/intercom/toggle",
  "subscription_topic": "kingkiosk/{deviceId}/intercom/status",
  "status_path": "broadcasting",
  "icon_on": "mic",
  "icon_off": "mic_off",
  "color_on": "0xFF4CAF50",
  "color_off": "0xFF757575",
  "label": "Intercom"
}
```

Replace `{deviceId}` with your device's ID. Tap the button to start broadcasting, tap again to stop. The icon and color update automatically based on the intercom status.

When you're broadcasting, a small self-view of your camera is overlayed into the mic icon so you can see what others are seeing. Audio is muted on the self-view to prevent echo — you only see yourself, you don't hear yourself.

0.1.13.3.5 Home Assistant Automation Example

Broadcast "dinner's ready" when a button is pressed:

```
automation:
  - alias: "Dinner Announcement"
    trigger:
      - platform: state
        entity_id: input_button.dinner_ready
    action:
      - service: mqtt.publish
        data:
          topic: "kingkiosk/kitchen-kiosk/intercom/start"
          payload: '{"group": "default"}'
      - delay: "00:00:10"
      - service: mqtt.publish
        data:
          topic: "kingkiosk/kitchen-kiosk/intercom/stop"
          payload: '{}'
```

0.1.13.3.6 Common Use Cases

- **Whole-house announcement** — "Dinner's ready!" from the kitchen to everywhere
- **Doorbell response** — Broadcast from the front door kiosk so all rooms see who's there

0.1.13.3.7 Privacy

- Any device can opt out by disabling intercom at any time
- Only devices with intercom enabled can send or receive
- Broadcasts are live only — nothing is recorded or stored

0.1.13.3.8 Troubleshooting

- **Broadcast not reaching other devices?** Make sure receivers have intercom enabled and are in the same group.
- **Hearing yourself (echo)?** The sender should only see video, not hear audio. Check that the client properly handles the `isSender` flag.
- **Video/audio won't auto-play?** KingKiosk handles this automatically, but web browsers may require user interaction first.

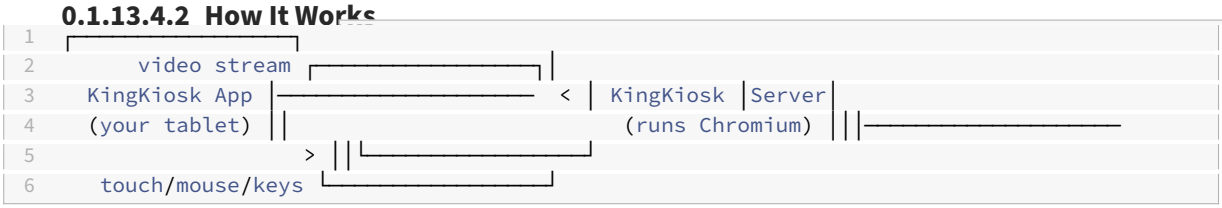
0.1.13.4 Remote Browser

Remote Browser lets you display and control a full web browser running on your server, streamed directly to your King Kiosk tablet. The heavy lifting happens on your server — your tablet just shows the video and sends back your touches.

0.1.13.4.1 Why Use It? Tablets and kiosks have limitations — weak hardware, limited memory, battery drain from complex sites, and logins that get lost on restart. Remote Browser solves all of this:

- **Server does the work** — A full Chromium browser runs on your powerful server
- **Tablet just displays** — It only needs to show a video stream
- **Stay logged in** — Sessions persist across restarts when you use a persistence ID
- **Touch or keyboard** — Control works however you prefer

0.1.13.4.2 How It Works



1. KingKiosk Server runs a headless Chromium browser
2. Video of the browser is captured and streamed to your tablet
3. Your touches are sent back to the server and applied to the browser
4. Everything happens in real-time with low latency

0.1.13.4.3 Creating a Remote Browser Send this MQTT command to your device's system topic to open a remote browser window:

```
{
  "command": "create_remote_browser",
  "window_id": "browser_1",
  "initial_url": "https://www.google.com",
  "auto_connect": true
}
```

Parameter	Required	Description
`window_id`	yes	Unique ID for this browser tile
`initial_url`	yes	URL to load when the session starts
`auto_connect`	yes	Always set to **`true`** to connect immediately
`name`	no	Title shown on the tile
`video_profile`	no	Quality: `auto`, `720p30`, `1080p30`, `1080p60`
`show_overlay`	no	Show URL bar and stats overlay

The server automatically picks the best video encoder for your hardware (NVIDIA GPU, Intel/AMD GPU, Raspberry Pi, or software fallback).

0.1.13.4.4 Navigating to a URL Once a remote browser session is running, you can navigate it:

```
{
  "command": "navigate_remote_browser",
  "window_id": "browser_1",
```

```
  "url": "https://youtube.com"
}
```

0.1.13.4.5 Controlling the Browser

- **Tap** = Click
- **Swipe** = Scroll
- **Long press** = Right-click
- **Physical or on-screen keyboard** = Type text

0.1.13.4.6 Common Use Cases

- **Dashboard kiosk** — Display a Home Assistant dashboard at full quality
- **Spotify player** — Stream music with album art on your wall display
- **YouTube TV** — Turn a tablet into a smart TV display
- **Security camera viewer** — View Frigate or other camera feeds in a browser

0.1.13.4.7 Troubleshooting

- **Session won't start?** Check server logs, verify Chromium is installed, and make sure the Feature Server is connected.
- **Video is laggy?** Try a lower quality profile, check WiFi, or reduce concurrent sessions.
- **Black screen?** Wait a moment for initial load, or try a simpler URL first.

0.1.13.5 RTSP Camera Export

RTSP Export lets you stream live video and audio from your King Kiosk tablet's camera to your smart home system. Once enabled, your tablet appears as a camera in Home Assistant, Frigate, or any app that supports RTSP streams.

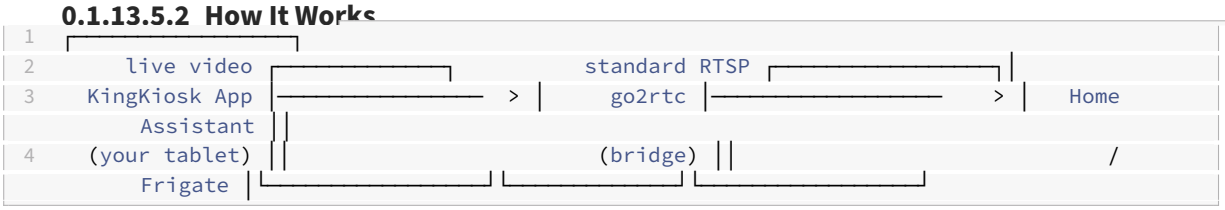
In simple terms: Turn your wall-mounted tablet into a security camera that shows up in Home Assistant just like your other cameras.

0.1.13.5.1 Why Use It? By default, King Kiosk only sends periodic snapshots (every few seconds) for AI detection. That works great for person detection and presence sensing, but it's not ideal when you want to:

- Watch a **live video feed** of what the tablet sees
- **Record continuous video** to Frigate
- Use the tablet as a **video doorbell or baby monitor**
- See **real-time motion** in Home Assistant

RTSP Export sends a continuous live video stream through go2rtc, making your tablet work exactly like any other security camera.

0.1.13.5.2 How It Works



0.1.13.5.3 Setup **Step 1:** Make sure go2rtc is running. Most Home Assistant users already have it (it's built into Frigate). Check by opening `http://your-home-assistant:1984`.

Step 2: Configure KingKiosk Server. For Home Assistant add-on users, go to Settings > Add-ons > KingKiosk Core > Configuration and set: - `go2rtc_url`: `rtsp://your-go2rtc-ip:8554` - `go2rtc_api_url`: `http://your-go2rtc-ip:1984`

For standalone users, add to `config.toml`:

```
[gstreamer]
go2rtc_url = "rtsp://192.168.1.100:8554"
go2rtc_api_url = "http://192.168.1.100:1984"
go2rtc_manage_streams = true
```

Step 3: Enable RTSP Export in KingKiosk's Feature Server Settings and turn the toggle on.

Step 4: Your device automatically appears in go2rtc using its device name (e.g., `kitchen-tablet`). View it at `rtsp://your-go2rtc:8554/kitchen-tablet`.

0.1.13.5.4 Adding to Home Assistant **Generic Camera:** Settings > Devices & Services > Add Integration > Generic Camera. Enter `rtsp://your-go2rtc:8554/kitchen-tablet`.

Frigate:

```
cameras:
  kitchen_tablet:
    ffmpeg:
      inputs:
        - path: rtsp://your-go2rtc:8554/kitchen-tablet
          roles:
            - detect
            - record
```

WebRTC Camera Card (lowest latency):

```
type: custom:webrtc-camera
url: kitchen-tablet
```

0.1.13.5.5 Snapshots vs. RTSP Export

Feature	Snapshots (Default)	RTSP Export
Use case	AI detection, periodic checks	Live viewing, recording
Updates	Every few seconds	Continuous (30fps)
Bandwidth	Very low	Higher (like a security camera)
CPU usage	Low	Moderate
Best for	Person detection, presence sensing	Baby monitor, security camera

You can use both at the same time — snapshots for AI detection and RTSP Export for live viewing.

0.1.13.5.6 Troubleshooting

- **Stream not showing in go2rtc?** Check server logs, verify go2rtc is reachable, confirm your config URLs.
- **Stream won't play?** Wait a few seconds for the first keyframe, or try VLC with TCP mode.
- **No audio?** KingKiosk uses Opus format. Add `default_query: video=h264&audio=aac` to your go2rtc config to convert to AAC.
- **Choppy video?** Use a wired connection if possible, or check WiFi signal strength.

0.1.13.6 Home Assistant MQTT Bridge

The Home Assistant MQTT Bridge is a two-way communication system that connects your Home Assistant smart home to MQTT. It makes all your Home Assistant devices (lights, switches, sensors, etc.) available through MQTT, and lets you control them by sending MQTT messages.

In simple terms: It's a translator that lets King Kiosk (and any other MQTT-connected system) see and control everything in your Home Assistant setup.

0.1.13.6.1 Why Use It?

Home Assistant is great at managing smart home devices, but sometimes other systems need to monitor or control those devices too. By bridging Home Assistant to MQTT, you can:

- **Connect external systems** that don't work with Home Assistant directly
- **Build custom automations** using tools that understand MQTT
- **Monitor your entire home** from any MQTT-capable application
- **Control devices** from King Kiosk or other systems without going through the HA API

0.1.13.6.2 How It Works

The bridge operates in both directions:

- **Home Assistant to MQTT:** When something changes in HA (a light turns on, a temperature changes, motion is detected), the bridge immediately publishes that update to MQTT.
- **MQTT to Home Assistant:** When you send a command through MQTT (turn on a light, set a thermostat), the bridge forwards it to Home Assistant.

0.1.13.6.3 What Gets Shared

Category	What It Contains
Entity States	Current status of all devices — lights (on/off, brightness, color), sensors (temperature, humidity), switches, covers, climate
Entity Registry	Device names, room assignments, manufacturer info, unique IDs
Device Registry	Physical device info — firmware versions, connection status, serial numbers
Area Registry	Rooms and locations in your home
Services Registry	All available actions and their parameters

0.1.13.6.4 Setup

Step 1: Get a Home Assistant Long-Lived Access Token. Go to your HA profile > Long-Lived Access Tokens > Create Token.

Step 2: Configure the bridge in your `config.toml`:

```
[home_assistant_bridge]
enabled = true
ha_url = "http://homeassistant.local:8123"
ha_token = "YOUR_LONG_LIVED_TOKEN_HERE"
```

```
mqtt_prefix = "kingkiosk/ha"
publish_registry = true
publish_services = true
publish_attributes = true
retain_state = true
```

Step 3 (optional): Filter what gets shared if you have many devices:

```
include_domains = ["light", "switch", "sensor"]
exclude_entity_globs = ["*_battery"]
```

0.1.13.6.5 Reading Device States

Subscribe to MQTT topics following this pattern:

```
kingkiosk/ha/state/{domain}/{object_id}
```

Example — living room light:

```
kingkiosk/ha/state/light/living_room
```

Returns:

```
{
  "entity_id": "light.living_room",
  "state": "on",
  "attributes": {
    "brightness": 200,
    "friendly_name": "Living Room Light"
  }
}
```

0.1.13.6.6 Controlling Devices

Send commands to:

```
kingkiosk/ha/command/{domain}/{object_id}
```

Turn on a light:

```
{"service": "turn_on", "data": {"brightness": 255}}
```

Set thermostat temperature:

```
{"service": "set_temperature", "data": {"temperature": 72}}
```

Turn off a switch:

```
{"service": "turn_off"}
```

Command results are published to `kingkiosk/ha/command/{domain}/{object_id}/result`.

0.1.13.6.7 Quick Reference

I Want To…	MQTT Topic
See all devices	`kingkiosk/ha/registry/entities`
Monitor a light	`kingkiosk/ha/state/light/{name}`
Turn on a light	`kingkiosk/ha/command/light/{name}` with `{"service": "turn_on"}`
See all available commands	`kingkiosk/ha/registry/services`
Check if bridge is working	`kingkiosk/ha/status`

0.1.13.6.8 Troubleshooting

- **Bridge shows "offline"?** Check that Home Assistant is running and accessible, verify your token, confirm the MQTT broker is connected.
- **Can see states but can't control devices?** Check your exclude filters, look at the command result topic for errors, verify the entity ID matches exactly.
- **Too much MQTT traffic?** Use filters to limit domains, disable registry/attribute publishing if you only need states.

0.1.14 13. Quick Reference Card

0.1.14.1 Essential Topics

Purpose	Topic
Send system commands	`kingkiosk/{device_id}/system/cmd`
Send widget commands	`kingkiosk/{device_id}/element/{widget_id}/cmd`
System responses	`kingkiosk/{device_id}/system/response`
Widget state	`kingkiosk/{device_id}/element/{widget_id}/state`

Purpose	Topic
Device info	`kingkiosk/{device_id}/info`
Online status	`kingkiosk/{device_id}/status`

0.1.14.2 Most-Used Commands

Action	Command
Create clock	`{"command": "open_clock", "window_id": "clock-1"}`
Create web browser	`{"command": "open_browser", "url": "...", "window_id": "web-1"}`
Close window	`{"command": "close_window", "window_id": "..."}`
Close all windows	`{"command": "close_all_windows"}`
Set volume	`{"command": "set_volume", "value": 0.5}`
Set brightness	`{"command": "set_brightness", "value": 0.8}`
Show notification	`{"command": "notify", "title": "...", "message": "..."}`
Text-to-speech	`{"command": "tts", "text": "..."}`
Take screenshot	`{"command": "screenshot"}`
Save layout	`{"command": "save_screen_state", "name": "..."}`
Load layout	`{"command": "load_screen_state", "name": "..."}`
Backup full config + layouts	`{"command":"get_config","include_secrets":true,"include_layouts":true}`

Action	Command
Restore/clone full config + layouts	`{"command":"provision","settings":{...},"screen_states":[...],"current_layout":{...},"overwrite":true}`
List windows	`{"command": "list_windows"}`

0.1.14.3 Widget Types

Widget	Create Command	Key Parameters
Clock	`open_clock`	`mode`, `theme`, `show_numbers`
Weather	`open_weather_client`	`api_key`, `location`, `units`
Web Browser	`open_browser`	`url`, `title`
Remote Browser	`create_remote_browser`	`server_url`, `initial_url`
Custom Widget	`open_browser`	`url`, `title`
Video/Audio	`play_media`	`type`, `url`, `loop`
YouTube	`youtube`	`url`
Carousel	`create_carousel`	`items`, `config`
Gauge	`create_gauge`	`gauge_type`, `min`, `max`, `unit`
Chart	`create_chart`	`chart_type`, `max_points`
Map	`open_map`	`initial_camera`, `provider`
Canvas	`open_canvas`	`doc`, `interaction`
Animated Text	`open_animated_text`	`text`, `preset`, `effects`
MQTT Image	`mqtt_image`	`mqtt_topic`, `is_base64`
MQTT Button	`mqtt_button`	`publish_topic`, `subscription_topic`
Calendar	`calendar`	`action`, `uid` (for sync)
Timer	`timer_widget`	`config.duration`
Stopwatch	`stopwatch`	-

Widget	Create Command	Key Parameters
DLNA Player	`dlna_player`	-
PDF	`open_pdf`	`url`
Alarmo	`alarmo`	`mqtt_base_topic`, `entity`, `require_code_to_arm`, `require_code_to_disarm`, `code_required_modes`, `force`, `skip_delay`
Games	`stop_the_missiles`	`game_type`, `config`

Thank you for being a King Kiosk beta tester! Your feedback shapes the future of digital signage. Report issues and share ideas in our Facebook group.

King Kiosk - Where Every Screen Becomes Extraordinary