webfleet

PRO X Driver Terminal

Developer documentation

BRIDGESTONE

Solutions for your journey

1729933602057 - 1.0 - 2025-02-26

Contents

Welcome

The PRO X Platform is a solution to help your business move more efficiently. It is a platform to enable you to seamlessly connect your business applications with solutions from Bridgestone Mobility Solutions. Now, you can easily create your own customised business applications that harness the power of TomTom's award-winning navigation technology and fleet management features from Bridgestone Mobility Solutions.

The PRO X Developer documentation provides the growing base of developers a wealth of information and documentation.

Develop

1. Demo application

A PRO X API demo integration application is available.

The source code can be downloaded from https://webfleet.com/webfleet/partners/integration/developer-resources/ - Under PRO X Driver Terminals named **pro-x-api-demo.zip** - and can then be compiled and installed locally. This is a modern android gradle project.

2. Webfleet integration via PRO.connect SDK

This feature is not yet supported. If you are interested in this feature please contact customer support.

3. Navigation app integration

3.1. Advanced integration via NavApp SDK

Important: TomTom GO Fleet APIs will only be accessible when your app is registered with TomTom GO Fleet. Please contact customer support about the setup. To start the process we will need a production apk file of your application (i.e. if your app is also distributed via Google Play, make sure to share the apk with the same signing).

TomTom GO Fleet offers a navigation SDK that allows deep integration. It is a Android/Java library that can be integrated with your application.

3.1.1. Features of the SDK

The following features are available when using the NavApp SDK:

- Planning a route: Free text search via address description (like in Google Maps) or directly via coordinates.
- Planning a route with waypoints or tracks (gpx files).
- Changing routing parameters.
- Changing vehicle profile or listening to profile changes done with the app.
- Getting feedback about the current route & routing state (ETA, remaining distance, events, POIs, etc).
- Receiving guidance instructions.
- Seeing installed maps.
- Setting behavior for map updates (WiFi, Mobile, disabled).

3.1.2. How to use it

The SDK has to be kept up to date with the TomTom GO Fleet application version. It will therefore be shared separately as part of the release communication to registered partners.

The SDK includes full javadoc and an example application.

3.2. Basic integration via intents

General information

TomTom GO Fleet supports platform navigation intent integration. Please take into consideration the following:

- Please take note that Google Maps will be shown in the intent names but this is fully supported with TomTom GO Fleet.
- The examples here show how to use the Android Debug Bridge (adb) to fire those intents. The same can be achieved programmatically from your application.

3.2.1. Location intents

Reference: Android intents - Search for a location.

Display a single location via coordinate

```
adb shell am start -a android.intent.action.VIEW -d "geo:52.52082,13.40938"
```

Display a single search result for a search query

PRO X API demo integration application:

Replacing the coordinates with 0,0 allows you to query for an actual address instead of coordinates.

```
adb shell am start -a android.intent.action.VIEW -d "geo:0,0?q=Berlin
+Alexanderplatz+1"
```

3.2.2. Navigation intents

Reference: [Android intents - Launch turn-by-turn navigation]()

PRO X API demo integration application:

This will directly launch a navigation to the provided address or coordinates.

Note: TomTom GO Fleet does not support ‘avoid’ or ‘traveling mode’ parameters for navigation intents.

```
adb shell am start -a android.intent.action.VIEW -d "google.naviga-
tion:q=Berlin+Alexanderplatz+1"
```

3.3. Getting ETA via WEBFLEET.connect

While there is no direct access to TomTom GO Fleet via on-device API, the unit's current ETA to its destination can also be acquired via WEBFLEET.connect.

For setting up refer to https://www.webfleet.com/static/help/webfleet-connect/en_gb/index.html#data/introductiontowebfleet.connect.html.

For actually acquiring the ETA, see https://www.webfleet.com/static/help/webfleet-connect/en_gb/index.html#data/showobjectreportextern.html (dest_eta).

4. Platform APIs

Important: The following APIs are only available with PRO X firmware SR10g and higher.

The PRO X platform API offers direct control of certain platform behavior.

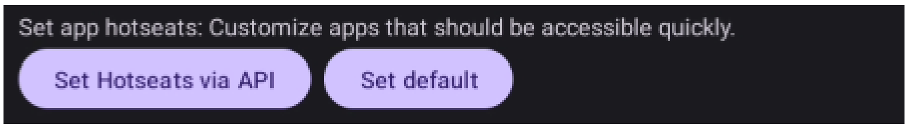
The latest version can be downloaded here: https://webfleet.com/webfleet/partners/integration/developer-resources/ - Under PRO X Driver Terminals named **PRO-X-API-v1.06.2.zip**.

The java library needs to be added to the resources of your application. E.g., under `app/src/main/libs` and then added to `app/src/build.gradle.kts`:

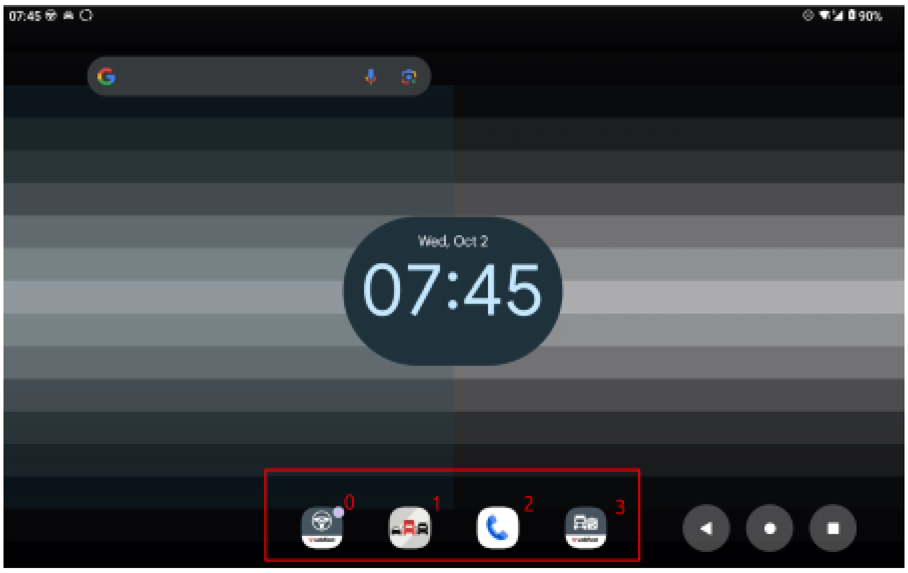
```
dependencies {
    implementation(files("src/main/libs/mitac-api-libs-poseidon-1.06.1-re-
lease.aar"))
    // Other app dependencies
}
```

4.1. Hotseat configuration

PRO X API demo integration application:


Set app hotseats: Customize apps that should be accessible quickly.
Set Hotseats via API
Set default

Hotseats can be configured for quick access to the most important apps. There are 4 slots available, the index starts at 0:



Hotseat configuration

Note: For more information about package and activity names please see the 7. Frequently asked questions section.

```
import com.mitac.api.libs.Launcher;

mLauncher = new Launcher(getContext().getApplicationContext(), new Ser-
viceStatusCallback() {
    @Override
    public void ready() {
        Log.d(TAG, "Launcher service is ready. Setting hotseats.");

        mLauncher.addOnHotseat("com.tomtom.videodockcamera", // package
 name
            "com.tomtom.videodockcamera.VideoDockCameraActivity", // Main
 activity
            "External Camera", // Label
            3); // Hotseat position
    }

    @Override
    public void stopped() {

    }
});
```

4.2. Restrict application usage

It's possible to setup a blacklist of applications that should not be available on the device. Blacklisted apps will not be visible in the launcher or apps overview. Blacklisted apps can include pre-installed applications that are usually shipped with the PRO X Platform by default.

Blacklist example

Note: For more information about package and activity names please see the 7. Frequently asked questions section.

```
com.google.android.apps.messaging
com.android.music
com.google.android.dialer
com.android.chrome
com.google.android.apps.photos
com.google.android.gm
com.google.android.googlequicksearchbox
com.google.android.apps.maps
com.google.android.apps.tachyon
com.android.vending
com.android.video
com.google.android.apps.docs
```

This static configuration file needs to be added to your app resources (e.g., `app/src/main/res/raw`) and can then be used like described below. A reboot is required for the changes to take effect.

Setting an app blacklist

```
// Import the FactoryUtility
import com.mitac.api.libs.FactoryUtility;
...

// The platform will only recognize the blacklist with this filename.
```

```
private static final String FILE_APPS_BLACKLIST = "available_apps_black-
list.txt";

private FactoryUtility factoryUtility;
...

// Instantiate the FactoryUtility together with your Application or View
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bun-
dle savedInstanceState)
{
    factoryUtility = new FactoryUtility(getContext().getApplicationContex-
t());
    factoryUtility.bindService();
    ...
}

// Set the blacklist. The blacklist takes effect only after a device re-
boot.
private void setAppBlacklist() {
    removeAppBlackList();
    Log.d(TAG, "Setting app blacklist file");
    // Acquire the blacklist configuration from the provided file
    final InputStream inputStream = getResources().openRawRe-
source(R.raw.available_apps_blacklist);
    final Scanner scanner = new Scanner(inputStream).useDelimiter("\\A");
    final String result = scanner.hasNext() ? scanner.next() : "";
    factoryUtility.writeReservedTextFile(FILE_APPS_BLACKLIST, result,
 false, false);
    Log.d(TAG, "Finished setting app blacklist file. Reboot required.");
}


// Remove blacklist on demand. Removing the blacklist will make the apps
 available again after a reboot.
private void removeAppBlackList() {
    Log.d(TAG, "Removing app blacklist file");
    if (factoryUtility.reservedFileExists(FILE_APPS_BLACKLIST)) {
        factoryUtility.removeReservedFile(FILE_APPS_BLACKLIST);
        Log.d(TAG, "Finished removing app blacklist file. Reboot re-
quired.");
    } else {
        Log.d(TAG, String.format("File %s could not be removed as it does
 not exist", FILE_APPS_BLACKLIST));
    }
}
```

If removing the blacklist via API is not possible (e.g., due to having a broken configuration or similar), please refer to these steps: 6.1.2. Resetting custom app restrictions

4.3. Enforcing online/allowing offline setup

We offer the possibility to enforce online or allow offline setup of the device.

```
// Import the FactoryUtility
import com.mitac.api.libs.FactoryUtility;
...

// The platform will only recognize the offline setup file with this file
 name
```

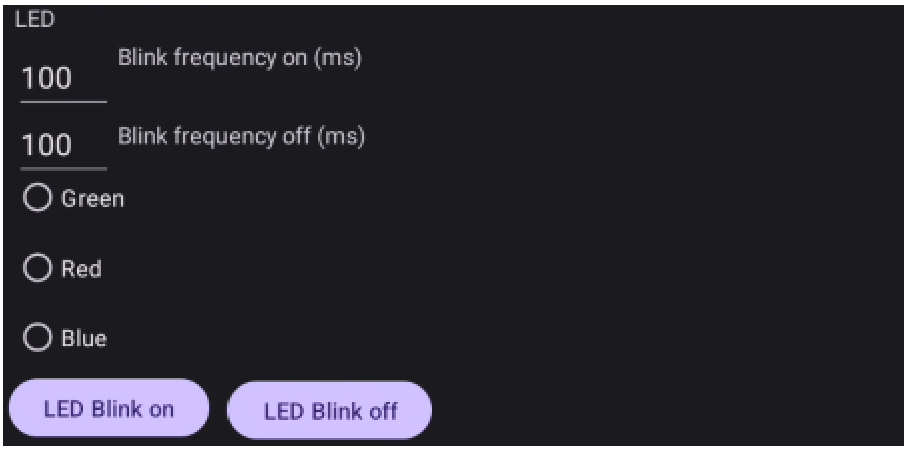
```
private static final String CONTROL_FLAG_FILENAME_OFFLINE_SETUP = "setup-
wizard_require_network";

private FactoryUtility factoryUtility;
...

// Instantiate the FactoryUtility together with your Application or View
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bun-
dle savedInstanceState)
{
    factoryUtility = new FactoryUtility(getContext().getApplicationContex-
t());
    factoryUtility.bindService();
    ...
}
// Enforce online setup on demand. This requires a reboot to take effect
private void enforceOnlineSetup()
{
    Log.d(TAG, "Enforcing online setup");
    if (!factoryUtility.reservedFileExists(CONTROL_FLAG_FILENAME_OF-
FLINE_SETUP))
    {
        factoryUtility.writeReservedTextFile(CONTROL_FLAG_FILENAME_OF-
FLINE_SETUP, "", false, false);
    }
    else
    {
        Log.d(TAG, "Control flag file for enforcing online setup already
 exists");
    }
}
// Enable offline setup on demand. This requires a reboot to take effect
private void allowOfflineSetup()
{
    Log.d(TAG, "Allowing offline setup");
    if (factoryUtility.reservedFileExists(CONTROL_FLAG_FILENAME_OF-
FLINE_SETUP))
    {
        factoryUtility.removeReservedFile(CONTROL_FLAG_FILENAME_OF-
FLINE_SETUP);
    }
    else
    {
        Log.d(TAG, "Control flag file to enforce online setup does not ex-
ist");
    }
}
```

4.4. LED

PRO X API demo integration application:



The device LED behaviour can be customised.

Customising the LED behaviour

```
// Import the LED API
import com.mitac.api.libs.LED;
...

private LED led;
...

// Instantiate the LED API together with your Application or View
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bun-
dle savedInstanceState) {
    led = new LED(getContext().getApplicationContext());
    led.bindService();
    ...
}

private void setLedBlinking() {
    // For more options see the API documentation
    final int color = LED.LED_ID.ID_RED;
    // Specify on/off intervals
    final int on = 100;
    final int off = 200;

    Log.d(TAG, String.format("Set LED blinking on. Color: %s, On: %d, Off:
 %d, service ready: %s", color, on, off, led.isServiceReady()));
    // Turn the LED off before setting a different blink status
    led.controlLed(color, LED.LED_STATUS.STATUS_OFF_ALL, 0, 0);
    // Set the new blink behavior
    led.controlLed(color, LED.LED_STATUS.STATUS_BLINK, on, off);
}

// Make sure to properly unbind the service to avoid memory leaks
public void onDestroyView() {
    super.onDestroyView();
    led.unbindService();
    ...
}
```

Note: Different LED colour controls can be combined at the same time.

4.5. System properties

The API allows reading and writing certain system properties. The current state of the device can be read out via adb: `adb shell getprop`.

Interacting with system properties

```
import com.mitac.api.libs.SystemUtilities;
...

private SystemUtilities systemUtilities;
...

// Instantiate the SystemUtilities API together with your Application or
 View
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bun-
dle savedInstanceState) {
    systemUtilities = new SystemUtilities(getContext().getApplicationCon-
text());
    systemUtilities.bindService();
    ...
}

// Getting a system properties value
systemUtilities.getSystemProp("persist.sys.ship.mode.current");

// Setting a system property to a certain value
systemUtilities.setSystemProp("persist.sys.ship.mode.current", "1");

// Make sure to properly unbind the service to avoid memory leaks
public void onDestroyView() {
    super.onDestroyView();
    systemUtilities.unbindService();
    ...
}
```

4.6. Controlling power behaviour

On PRO X devices, third party apps are able to change the default power button behavior.

4.6.1. Receiving suspend/shutdown intents

To intercept power button intents coming from the system, a broadcast receiver needs to be register dynamically or statically for the **com.webfleet.prosystemapp.ACTION_REQUEST_CONFIRM_SUSPEND_SHUTDOWN** intent action which requires the **com.webfleet.prosystemapp.permission.REQUEST_SUSPEND_SHUTDOWN** permission.

An example for statically registered receiver looks as follows in the Android manifest:

```
...
<uses-permission android:name="com.webfleet.prosystemapp.permission.RE-
QUEST_SUSPEND_SHUTDOWN"/>
...
```

```
<receiver
    android:name="com.example.StaticPowerOffReceiver"
    android:exported="true"
    android:permission="com.webfleet.prosystemapp.permission.RE-
QUEST_SUSPEND_SHUTDOWN">
    <intent-filter android:priority="10" >
        <action android:name="com.webfleet.prosystemapp.ACTION_REQUEST_CON-
FIRM_SUSPEND_SHUTDOWN" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</receiver>
...
```

t's imported to provide a higher than default (0) priority on the intent filter to receive the intent.

Note: Once a broadcast receiver has been registered for the com.webfleet.prosystemapp.ACTION_REQUEST_CONFIRM_SUSPEND_SHUTDOWN intent action, the default power button behavior of the PRO X device will not work anymore and you as a third party developer are responsible for handling it properly.

4.6.2. Actively suspend or shutdown the device

In order to suspend or shutdown the device (reboot is always handled via the power system dialog) the implementation of the broadcast receiver could look as follows:

```
/**
 * Broadcast receiver to handle intents for suspend or shutdown.
 */
public class StaticPowerOffReceiver extends BroadcastReceiver {
    private static final String TAG = "StaticPowerOffReceiver";
    public enum PowerOffReason {
        USER(4),
        TIMEOUT(2),
        UNPLUGGED(13),
        SYSTEM(0),
        UNDEFINED(-1);

        private final int val;

        PowerOffReason(int val) {
            this.val = val;
        }

        public int getVal() {
            return this.val;
        }

        public static PowerOffReason fromInt(int val) {
            for (PowerOffReason reason : PowerOffReason.values()) {
                if (val == reason.getVal()) {
                    return reason;
                }
            }
            return UNDEFINED;
        }
    }

    public enum PowerOffState {
        SHUTDOWN(1),
        SUSPEND(2),
        UNDEFINED(-1);

        private final int val;

        PowerOffState(int val) {
            this.val = val;
        }

        public int getVal() {
            return this.val;
        }

        public static PowerOffState fromInt(int val) {
            for (PowerOffState state : PowerOffState.values()) {
                if (val == state.getVal()) {
                    return state;
                }
            }
            return UNDEFINED;
        }
    }

    public static final String ACTION_REQUEST_CONFIRM_SUSPEND_SHUTDOWN =
 "com.webfleet.prosystemapp.ACTION_REQUEST_CONFIRM_SUSPEND_SHUTDOWN";
    public static final String REQUEST_SUSPEND_SHUTDOWN_PERMISSION =
 "com.webfleet.prosystemapp.permission.REQUEST_SUSPEND_SHUTDOWN";
    public static final String REQUEST_SHUTDOWN_ACTION = "com.tomtom.p-
nd.navpadsystemapp.ACTION_REQUEST_SHUTDOWN";
    public static final String REQUEST_SUSPEND_ACTION = "com.tomtom.p-
nd.navpadsystemapp.ACTION_REQUEST_SUSPEND";
    public static final String POWEROFF_REASON_KEY = "poweroff_reason";
    private static final String POWEROFF_STATE_KEY = "poweroff_state";
    private static final String PRO_SYSTEM_APP = "com.tomtom.pnd.navpadsys-
temapp";

    @Override
    public void onReceive(final Context context, final Intent intent) {

        if (ACTION_REQUEST_CONFIRM_SUSPEND_SHUTDOWN.equals(intent.getAc-
tion())) {
            Log.d(TAG, "ACTION_REQUEST_CONFIRM_SUSPEND_SHUTDOWN received on
 static receiver");

            final Bundle bundle = intent.getExtras();
            if (bundle != null) {
                // Every intent contains a power off reason and state indi-
cating what the
                // source of the intent was and whether suspend or shutdown
 should be done.
                final int powerOffReasonValue = bundle.getInt(POWEROF-
F_REASON_KEY, -1);
                final PowerOffReason powerOffReason = PowerOffRea-
son.fromInt(powerOffReasonValue);
                final int powerOffStateValue = bundle.getInt(POWEROFF_S-
TATE_KEY, -1);
                final PowerOffState powerOffState = PowerOffState.fromIn-
t(powerOffStateValue);
                Log.i(TAG, "Power off reason: " + powerOffReason + " state:
 " + powerOffState);
```

```
                switch (powerOffReason) {
                    case UNPLUGGED: // Device was unplugged from power
                        // intended fall-through
                    case SYSTEM: // System wants suspend or shut down
                        // intended fall-through
                    case USER: // User pressed the power button
                        // For every reason other than TIMEOUT we need de-
termine whether
                        // the device should be suspended or shut down. We
 then send the
                        // appropriate intent back to the system app han-
dling the actual
                        // suspend or shutdown.
                        if (powerOffState == PowerOffState.SUSPEND) {
                            final Intent suspendIntent = new Intent(RE-
QUEST_SUSPEND_ACTION);
                            suspendIntent.setPackage(PRO_SYSTEM_APP);
                            context.sendBroadcast(suspendIntent);
                        } else if (powerOffState == PowerOffState.SHUTDOWN)
 {
                            final Intent shutdownIntent = new Intent(RE-
QUEST_SHUTDOWN_ACTION);
                            shutdownIntent.setPackage(PRO_SYSTEM_APP);
                            context.sendBroadcast(shutdownIntent);
                        } else {
                            Log.w(TAG, "Unsupported power off state: " +
 powerOffState);
                        }
                        break;
                    case TIMEOUT:
                        // In case of a screen timeout we directly suspend
 the device
                        final Intent suspendIntent = new Intent(RE-
QUEST_SUSPEND_ACTION);
                        suspendIntent.setPackage(PRO_SYSTEM_APP);
                        context.sendBroadcast(suspendIntent);
                        break;
                    default:
                        break;
                }
            }
        }
    }
}
```

4.7. Further options

So far, we've highlighted certain important aspects of the API and we invite you to see which other options the platform API might be providing for your use case. The full javadoc for the API is included in the zip file available here: https://webfleet.com/webfleet/partners/integration/developer-resources/ - Under PRO X Driver Terminals.

5. Platform customisation

Certain behavior of the platform can be customized to your needs without additional development, just by using configuration files.

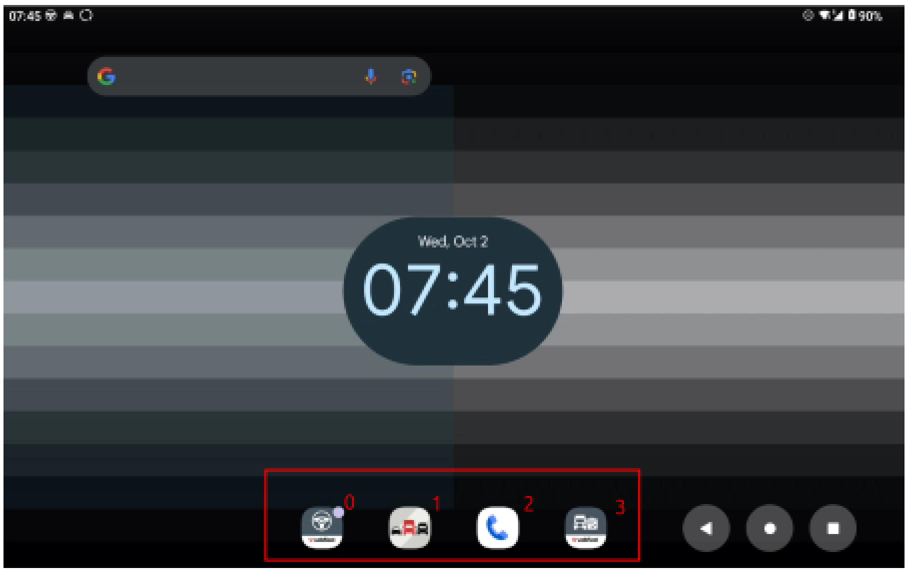
5.1. Delivery via Webfleet MDM

We suggest using Webfleet MDM as a distribution platform for those changes (see 6.2. Applying customisations via MDM), but technically any form of delivery of the discussed files to their respective location can achieve the same results (manual provisioning or 3rd-party EMMs).

5.2. Hotseat configuration

Hotseats can be configured via a static configuration file `hotseat_customer.json`. This is an example hotseat file.

Hotseats can be configured for quick access to the most important apps. There are 4 slots available, the index starts at 0:



The behaviour is as follows:

- 4 Hotseats can be configured (0-3).
- **Important:** If the hotseats (or any other configuration done via this file) is updated, the version tag must be increased. Example: From 1.0 to 2.0.
- The file must be saved to one of the following locations:
  - Root of an external SD card.
  - Root of internal device storage (`/sdcard`).
  - If it exists on both locations, external SD card takes prevalence.

- A reboot is required for the changes to take effect.
- If the hotseats is reset, a device factory reset is required. Removing the file will not revert it back to the default. Instead the default can be applied via a default config, which can be found on our developer resources page, the file is named 'hotseat_customer.json'.

Hotseat configuration

Note: For more information about package and activity names please see the 7. Frequently asked questions section.

```
{
    "customer_hotseat_version": "1.0",
    "hotseat": [
        {
            "package": "com.webfleet.proapp",
            "class": "com.webfleet.proapp.MainActivity",
            "x": "0"
        },
        {
            "package": "com.tomtom.gplay.navapp.gofleet",
            "class": "com.tomtom.mykonosapp.MykonosAppActivity",
            "x": "1"
        },
        {
            "package": "com.google.android.dialer",
            "class": "com.google.android.dialer.extensions.GoogleDialtacts-
Activity",
            "x": "2"
        },
        {
            "package": "com.prox.shortcut.language",
            "class": "com.prox.shortcut.MainActivity",
            "x": "3"
        }
    ]
}
```

5.2.1. Control visibility of hotseats

In case you wish to use the Kiosk mode provided by your EMM together with your PRO X, it's advised to disable hotseats completely to prevent users from breaking out of the kiosk mode. This can be achieved via blocking or allowing the pre-installed application `com.prox.hotseatscontroller` via EMM policy:

- Blocking the application: hotseats won't be visible.
- Allowing the application: hotseats will be visible.

Note: While by default hotseats are allowed and visible, should you have blocked them once on a device, you actively need to allow hotseats again via policy (instead of just removing the policy completely). Alternatively, a factory reset device will show the hotseats again, even without any related policy.

5.3. Settings shortcut apps

The platform offers 5 shortcut apps into certain settings. They are intended for EMM use cases where the default Android settings app use could be restricted, while some specific detailed settings apps might still be enabled. The following shortcut apps are supported:

- com.prox.shortcut.bluetooth
- com.prox.shortcut.display
- com.prox.shortcut.language
- com.prox.shortcut.security
- com.prox.shortcut.wifi

5.3.1. Using setting shortcuts for hotseat configuration

All setting shortcut apps share the same activity class and can therefore be configured like this:

```
{
    "package": "com.prox.shortcut.bluetooth",
    "class": "com.webfleet.proapp.MainActivity",
    "x": "0"
}
```

5.4. Automatic app startup

It's possible to configure automatic startup of apps and/or services after device boot.

A configuration file looks like follows:

autostart_app_list.json example

Note: For more information about package and activity names please see the 7. Frequently asked questions section.

```
{
   "auto_activity": [
      {
         "component": "com.webfleet.proapp/com.webfleet.proapp.MainActivi-
ty",
         "show_time": 10000
      },
      {
         "component": "com.tomtom.gplay.navapp.gofleet/com.tomtom.mykonos-
app.MykonosAppActivity",
         "show_time": 1000
      }
   ],
   "auto_service": [
      "com.example.app/.ExampleService"
   ]
}
```

This file called `autostart_app_list.json` needs to be moved to the root of the internal device storage (`/sdcard`).

5.4.1. Behaviour

- One of ‘auto_activity’ or ‘auto_service’ (or both) needs to be present.
- ‘show_time’ controls how long this app will be visible until the next app will be started:
    - If this element is missing, a default of 1000ms will be applied.
    - This element needs to be tuned in case you want to allow your auto-start apps to do certain initialisation before being moved to the background again.
- The sorting of auto-start is top-to-bottom:
    - The first app will be started immediately after the device booted.
    - It will be shown for the specified ‘show_time’, after which the next app will be started.

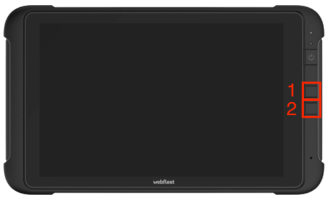
5.4.2. Example

With this example configuration, Work App will be auto-started with enough time to initialise itself, and afterwards TomTom GO Fleet will be started. This way, drivers just need to boot the device and can start navigating, while the fleet manager can be sure that Work App will communicate with Webfleet.

1. Download example file available here: https://webfleet.com/webfleet/partners/integration/developer-resources/ - Under PRO X Driver Terminals named **autostart_app_list.json**.
2. Move the provided example to the correct location with `adb push autostart_app_list.json /sdcard/`.
3. Reboot the device.

5.5. Hardware button mapping

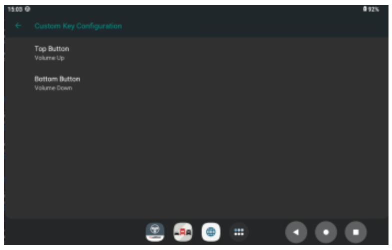
The hardware buttons below the **Power button** on the right side of the device frame can be mapped to specific actions.

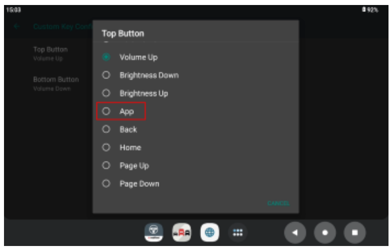


5.5.1. Via UI

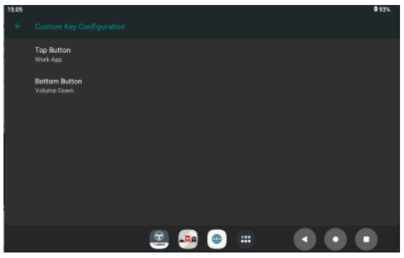
Go to Android Settings and scroll down to ‘Custom Key Configuration’.

Setting up an app for the hardware buttons is done like this:


Custom Key Configuration
Top Button
Volume Up
Bottom Button
Volume Down


Top Button
- Volume Up
- Brightness Down
- Brightness Up
- App
- Back
- Home
- Page Up
- Page Down

CANCEL


Custom Key Configuration
Top Button
Work App
Bottom Button
Volume Down

5.5.2. Via configuration file

The `hotseat_customer.json` configuration mentioned above can be extended to configure the hardware button mapping. The full package name plus main activity need to be specified. If the `app` field is longer than 90 characters, the short cut notation needs to be used (e.g. `com.webfleet.proapp/.MainActivity`).

Additional information about the key codes used for the hardware:

- Keycode 114 stands for the top button.
- Keycode 115 stands for the bottom button.

Note: For more information about package and activity names please see the 7. Frequently asked questions section.

```
{
    "customer_hotseat_version": "2.0",
    "hotseat": [
        // This is optional for this scenario...
    ],
    "system_properties": [
        {
            "persist.sys.key.114.app": "com.tomtom.gplay.navap-
p.gofleet/com.tomtom.mykonosapp.MykonosAppActivity",
            "persist.sys.key.114.keycode": "F1",
            "persist.sys.key.114.remap": "true",
            "persist.sys.key.115.app": "com.webfleet.proapp/
com.webfleet.proapp.MainActivity",
            "persist.sys.key.115.keycode": "F2",
            "persist.sys.key.115.remap": "true"

        }
    ]
}
```

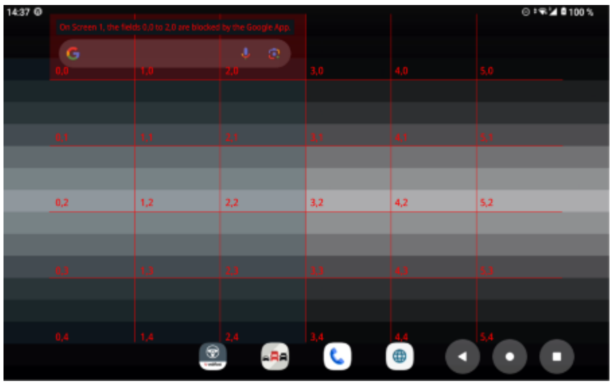
5.6. Homescreen configuration

The `hotseat_customer.json` configuration can also be extended to configure the home screen layout. After the file is added to the correct location (see above) changes take effect after a reboot.

5.6.1. Available options

The home screen follows a grid layout of 6 (horizontal) by 5 (vertical) elements.

This example shows screen 1. A section in the top left is highlighted, which is due to Google certification needs to be occupied by the 'Google' app and cannot be changed. On all other screens it's possible to occupy each grid with a custom element.


On Screen 1, the fields 0,0 to 2,0 are blocked by the Google App.

The following options are available for the `add_appwidget` object:

- package: The app's package name.
- class: The main class name to launch.
- screen: The screen index, optional, default is 0.
- x,y: The element's (starting) position on the grid.
    - Grid coordinates can be seen in the screenshot above. Per page they start on 0,0 and go to 5,4.
- span_x, span_y: Optional, default is 1 in both dimensions. Used for widgets stretching over multiple fields. Not applicable for single app icons.
    - Maximum span size is x=6 and y=5, this element would stretch over the whole grid of the screen.

5.6.2. Example configuration

The example file can be downloaded from: https://webfleet.com/webfleet/partners/integration/developer-resources/ - Under PRO X Driver Terminals named **hotseat_customer.json**.

```
{
    "customer_hotseat_version": "3.0",
    "hotseat": [
        // This is optional for this scenario...
    ],
    "system_properties": [
        // This is optional for this scenario...
    ],
    "add_appwidget": [
        {
            "package": "com.android.deskclock",
            "class": "com.android.alarmclock.DigitalAppWidgetProvider",
            "screen": "0",
            "x": "0",
            "y": "2",
            "span_x": "3",
            "span_y": "1"
        },
        ...
        {
            "package": "com.webfleet.proapp",
            "class": "com.webfleet.proapp.MainActivity",
            "x": "2",
            "y":"1"
        },
        ...
        {
            "package": "com.webfleet.proapp",
            "class": "com.webfleet.proapp.MainActivity",
            "x": "4",
            "y":"4"
        }
    ]
}
```

5.7 Restrict application usage

Complementary to the API described in 4.2. Restrict application usage, app usage restrictions can also be achieved via configuration file. Just specify the blacklist as described in the linked section and put a file `available_apps_blacklist.txt` to `/sdcard` (root of internal device storage). After a reboot, the restrictions will be applied and the blacklist configuration file will be automatically removed from the devices' `/sdcard` folder.

Removing restrictions can be done by supplying a new empty blacklist file or as described in 6.1.2. Resetting custom app restrictions.

6. Device updates and backup

6.1. Device recovery

PRO X offers 3 supported ways of updating the device:

- Over the air update via WiFi/Mobile with the Webfleet Updater.
- SD card update with the Webfleet Updater.
- Recovery update via SD card.

The recovery process is intended to resolve issues that cannot otherwise be fixed by a factory reset.

The recovery zip file can be downloaded via MDM and as such it's part of every downloaded SD card.

Certain triggers can be applied during the recovery to influence its behaviour, these can be found below.

6.1.1. Keeping user data during the recovery

Important: As some problems that lead to a device recovery will be caused by having a broken state inside the userdata partition, using this flag should be done with caution.

Adding the 2 files below to the `/sdupdate` folder on the recovery SD card ensures that all userdata (see https://source.android.com/docs/core/architecture/partitions) is kept (including e.g. downloaded TomTom GO Fleet maps or other application data).

- keep_metadata.txt
- keep_userdata.txt

6.1.2. Resetting custom app restrictions

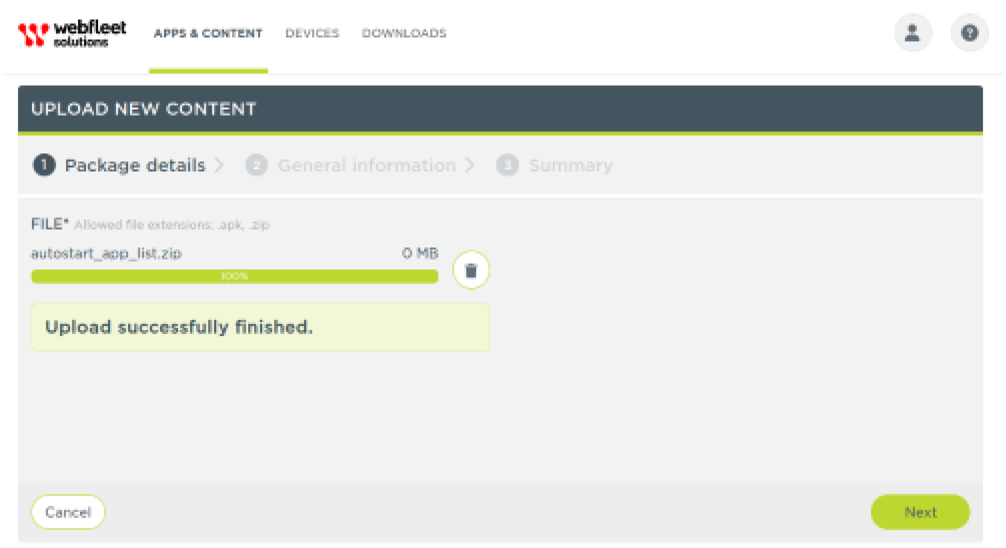
Adding a file called `remove_app_blacklist.txt` to the `/sdupdate` folder will reset the app blacklist during the recovery process.

6.2. Applying customisations via MDM

All customizations described in this document that can be applied via a configuration file in `/sdcard` can be delivered via the Webfleet MDM to all of your devices. Please find the instructions below.

Important: If you are operating a mixed fleet of PRO devices, the config files not applicable for your device will be downloaded and installed by the on-device updater but will have no impact on the device functionality.

1. Create your customization config file.
2. Zip the config file.
3. Go to MDM → My Content → Add new content.
4. Select the zip file:



UPLOAD NEW CONTENT

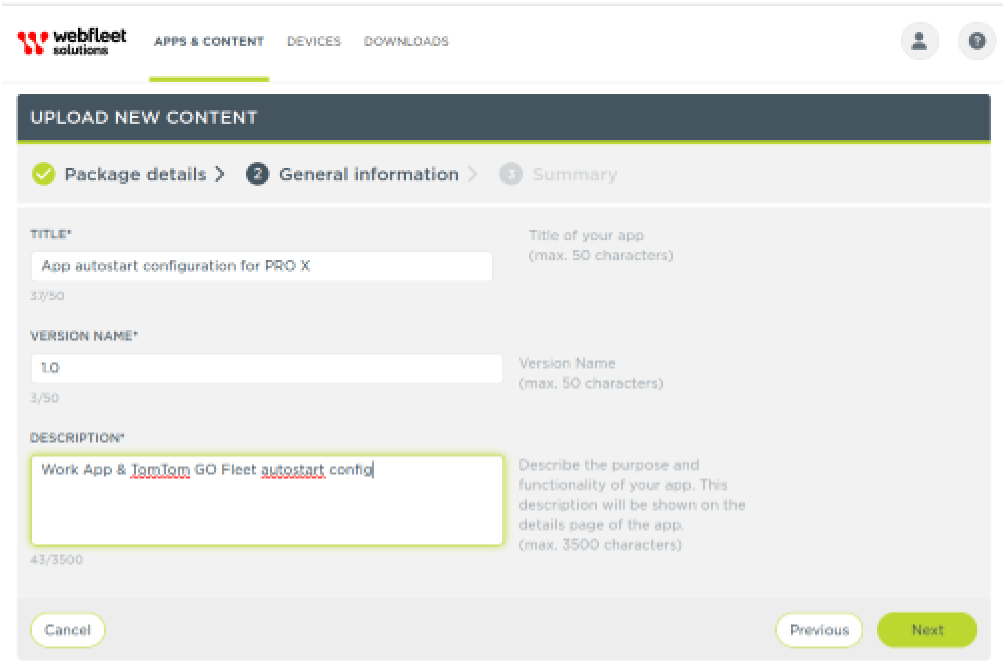
1 Package details > 2 General information > 3 Summary

FILE* Allowed file extensions: .apk, .zip

autostart_app_list.zip 0 MB 100%

Upload successfully finished.

5. Finish the upload process:



UPLOAD NEW CONTENT

Package details > 2 General information > 3 Summary

TITLE*: App autostart configuration for PRO X — 37/50 — Title of your app (max. 50 characters)

VERSION NAME*: 1.0 — 3/50 — Version Name (max. 50 characters)

DESCRIPTION*: Work App & TomTom GO Fleet autostart config — 43/3500 — Describe the purpose and functionality of your app. This description will be shown on the details page of the app. (max. 3500 characters)

6. Assign the new content to your account and update your devices.

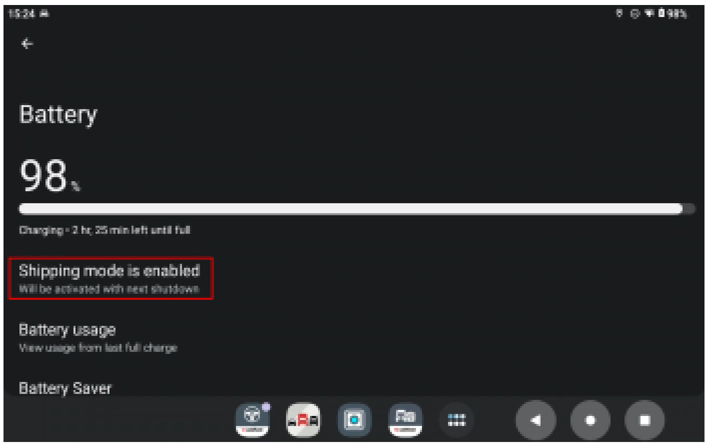
6.3. Shipping mode

To help preserve battery capacity over longer periods of time, shipping mode can be enabled.

When shipping mode is enabled, it will take effect on the next shutdown. The device will then disconnect the connection to the battery for the whole time it stays powered off.

After booting for the first time after the device had shipping mode enabled, it will again be disabled.

Note: Only when shipping mode is enabled, this element will be visible inside Settings # Battery.



15:24

98%

Battery

98%

Charging - 2 hr, 25 min left until full

Shipping mode is enabled
Will be activated with next shutdown

Battery usage
View usage from last full charge

Battery Saver

6.3.1. Enabling shipping mode

- Can be done via System Properties API, for an example see 4.5. System properties.
    - `persist.sys.ship.mode.current` - If set to '1', the device will immediately enter shipping mode on the next shutdown. After the first boot, the value will reset to 0.
- Shipping mode is disabled after a manual reboot.
- If turned on, shipping mode is visible under Settings → Battery.

6.4. Device backup

These settings are covered by the Android device backup functionality:

- **Network:** Wi-Fi & password, Mobile network / APN setting.
- **Display:** Keep Screen on when powered, Switch off screen when disconnected from Power.
- **Battery:** Protect batteryStart device automatically when connected to Power.
- **Sound:** Phone ringtone, Default notification sound,Default alarm sound.
- **Security & location:** Screen lock.
- **System:** Language.

Backups can be created by connecting the device to the PC and executing this command:
`adb backup -system android com.android.providers.settings`

Backups can be applied in 2 ways:

1. On the device directly via this command: `adb restore backup.ab`
2. Via Webfleet MDM:
    - Create `backup.ab` file.
    - Zip the file.
    - Go to MDM → My Content → Add new content → Upload the zip and assign to your account.

7. Frequently asked questions

What are commonly used applications on the PRO X?

In case any automated interaction with one of the pre-installed apps is required, this table might come in handy.

Displayed app name	Package	Main activity
Work App	com.webfleet.proapp	com.webfleet.proapp/.MainActivity
Vehicle Check App	com.webfleet.vcheckapp	com.webfleet.vcheckapp/.MainAc-tivity
TomTom GO Fleet	com.tomtom.gplay.navapp.gofleet	com.tomtom.mykonosap-p/.MykonosAppActivity
Webfleet Up-dater	com.tomtom.pnd.updater	com.tomtom.pnd.updater/.Updat-erActivity
External Camera	com.tomtom.videodockcamera	com.tomtom.videodockcamer-a.VideoDockCameraActivity
Quick setting: Bluetooth	com.prox.shortcut.bluetooth	com.prox.shortcut.MainActivity
Quick setting: Display	com.prox.shortcut.display	com.prox.shortcut.MainActivity
Quick setting: Language	com.prox.shortcut.language	com.prox.shortcut.MainActivity
Quick setting: Security	com.prox.shortcut.security	com.prox.shortcut.MainActivity
Quick setting: WiFi	com.prox.shortcut.wifi	com.prox.shortcut.MainActivity
Phone	com.google.android.dialer	com.google.android.dialer.exten-sions.GoogleDialtactsActivity
Chrome	com.android.chrome	com.google.android.app-s.chrome.Main
Clock	com.google.android.deskclock	com.google.an-droid.deskclock/com.an-droid.deskclock.DeskClock
Messages	com.google.android.apps.messag-ing	com.google.android.apps.messag-ing.ui.ConversationListActivity
Play Store	com.android.vending	com.google.android.finsky.activi-ties.MainActivity
Camera	com.android.camera2	com.android.camera.Camer-aLauncher

Displayed app name	Package	Main activity
Files	com.android.documentsui2	com.android.documentsui2.files.FilesActivity
Photos	com.google.android.apps.photos	com.google.android.apps.photos.home.HomeActivity
Calendar	com.google.android.calendar	com.google.android.calendar.AllInOneCalendarActivity
Contacts	com.google.android.contacts	com.google.android.contacts/com.android.contacts.activities.PeopleActivity
Drive	com.google.android.apps.docs	com.google.android.apps.docs/.drive.app.navigation.NavigationActivity
Gmail	com.google.android.gm	com.google.android.gm.ConversationListActivityGmail
Maps	com.google.android.apps.maps	com.google.android.apps.maps/.MapsActivity
Meet	com.google.android.apps.tachyon	com.google.android.apps.tachyon/.ui.main.MainActivity
Music	com.android.music	com.android.music/.MusicBrowserActivity
Google	com.google.android.googlequicksearchbox	com.google.android.googlequicksearchbox/.SearchActivity
Keep Notes	com.google.android.keep	com.google.android.keep/.activities.BrowseActivity
Sound Recorder	com.android.soundrecorder	com.android.soundrecorder/.SoundRecorder
Video	com.android.video	com.android.video/.VideoBrowserActivity

How to list packages & activities via adb

The command `adb shell pm list packages` allows you to see package names of all installed applications.

The command `adb shell dumpsys package <package_name> all` then allows you to find out which activity will respond to the category LAUNCHER, meaning this is the main activity that is started when opening this app from the Android launcher.

In the example of Work App, the response looks like this (excerpt):

```
> adb shell dumpsys package com.webfleet.proapp all

Activity Resolver Table:
...
```

```
  Non-Data Actions:
      android.intent.action.MAIN:
        e5ba5d1 com.webfleet.proapp/.MainActivity filter a317836
          Action: "android.intent.action.MAIN"
          Category: "android.intent.category.LAUNCHER"
...
```

How to show current package and activity

When wanting to find out which application and activity is currently running in the foreground of the PRO X, use this command:

```
adb shell "dumpsys activity activities | grep ResumedActivity"
```

In the case of Work App, this yields the following results:

```
topResumedActivity=ActivityRecord{91579e7 u0 com.webfleet.proapp/.MainAc-
tivity} t55}
ResumedActivity: ActivityRecord{91579e7 u0 com.webfleet.proapp/.MainActivi-
ty} t55}
```

How to find out details from the Android manifest

With the help of the Android build tools, you will be able to inspect the manifest of an application to find out important information about intents etc.

```
aapt dump xmltree <app_apk_file_name> AndroidManifest.xml
```

Copyright notices