Google
Automotive
Partner
Bootcamp

Car Framework Core

CONFIDENTIALITY REMINDER

Everything shared in this presentation is under **NDA**

Ethan Lee

Software Engineer

Gaurav Bhola

Software Engineer

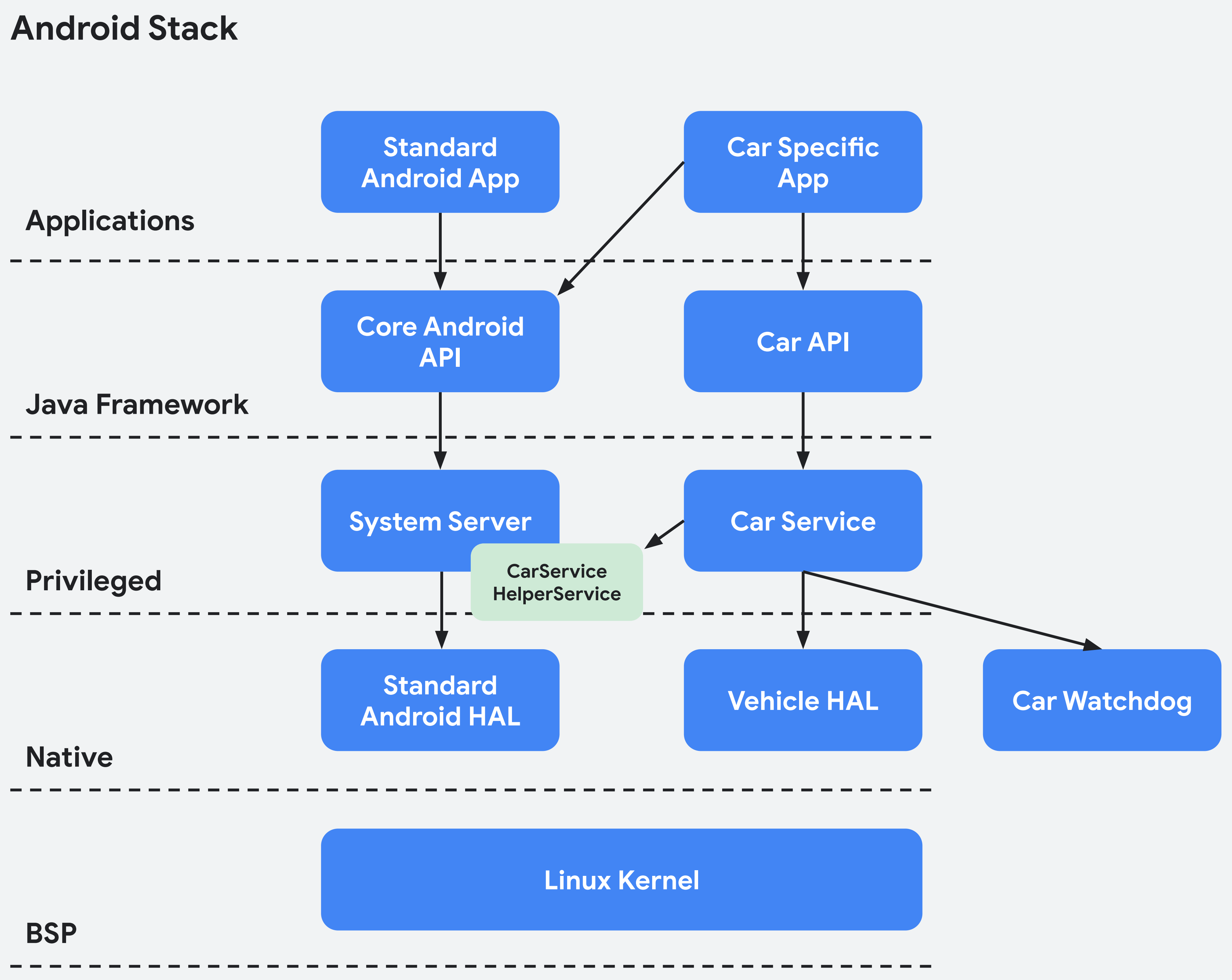
Agenda

Car Framework Architecture

Car Framework

Key Components in the Android Stack:

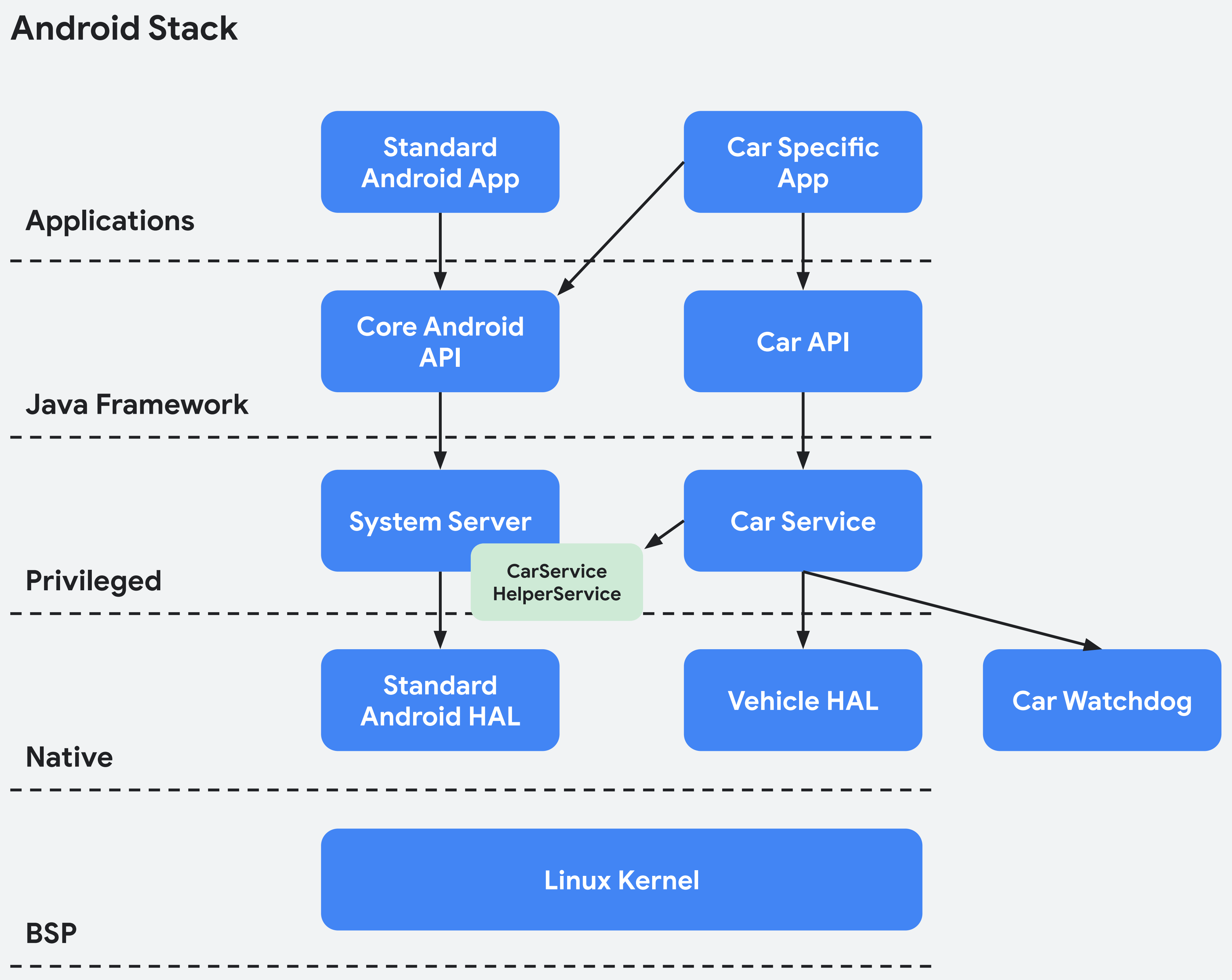
- App Layer
- Java Framework
- Native
- BSP



The Car Stack vs Traditional Android

Difference between the Traditional Android Stack and the Car Stack:

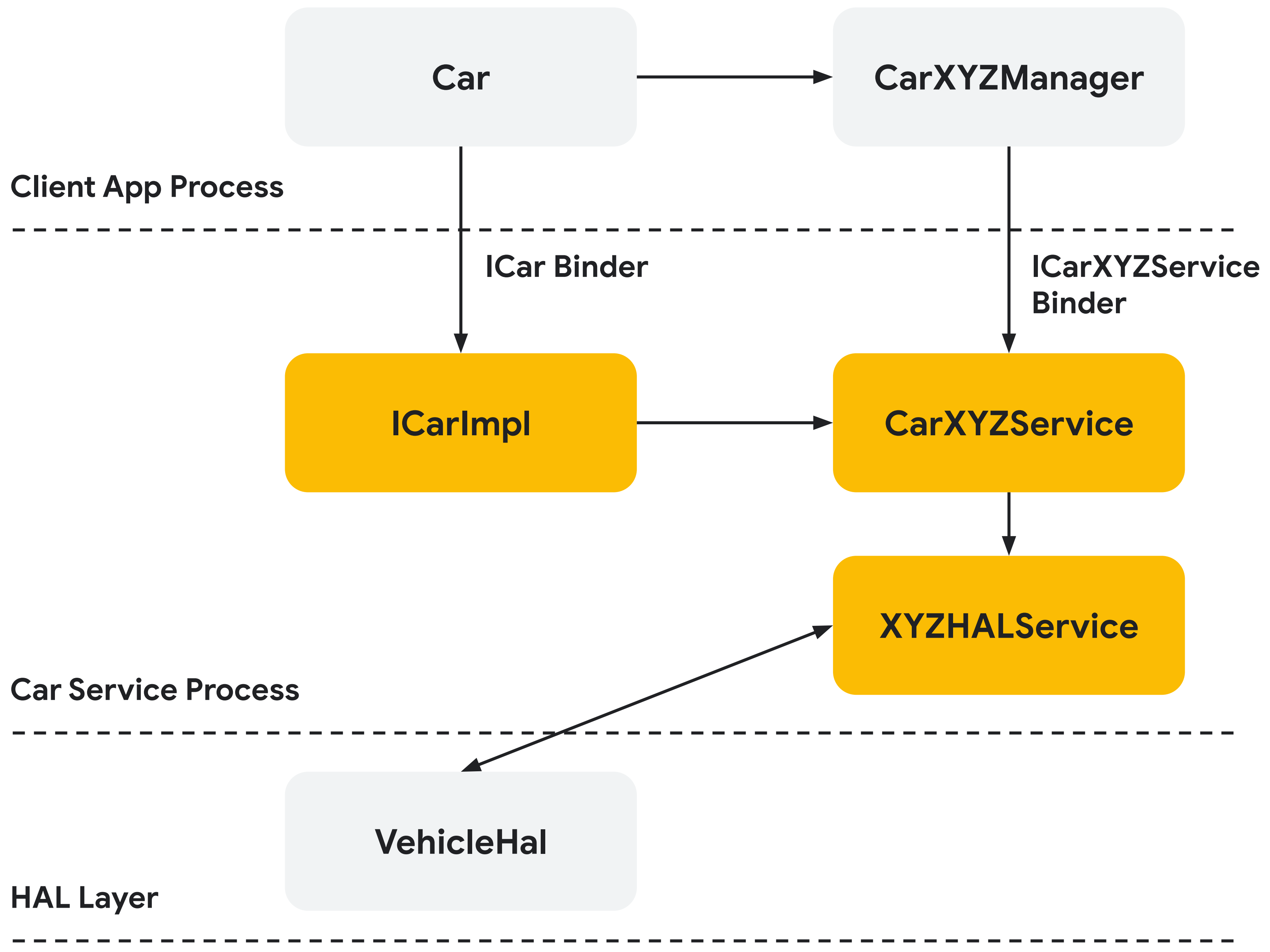
- Car-specific apps use Car APIs to access functionality implemented by Car Service.
- Car Service communicates with the System Server via CarServiceHelperService to access core Android functionalities.
- CarServiceHelperService's main purpose is to start Car service. However, CSHS is used when there is no specified API to communicate with System Server.
- Car Service also connects to car-specific native services such as CarWatchdog. These services handle Car-specific tasks before the system server and Car Service are intialized.
- Car hardware is abstracted away using Vehicle HALs and car-specific HALs.



Car Service and Car Managers

What is Car Service?

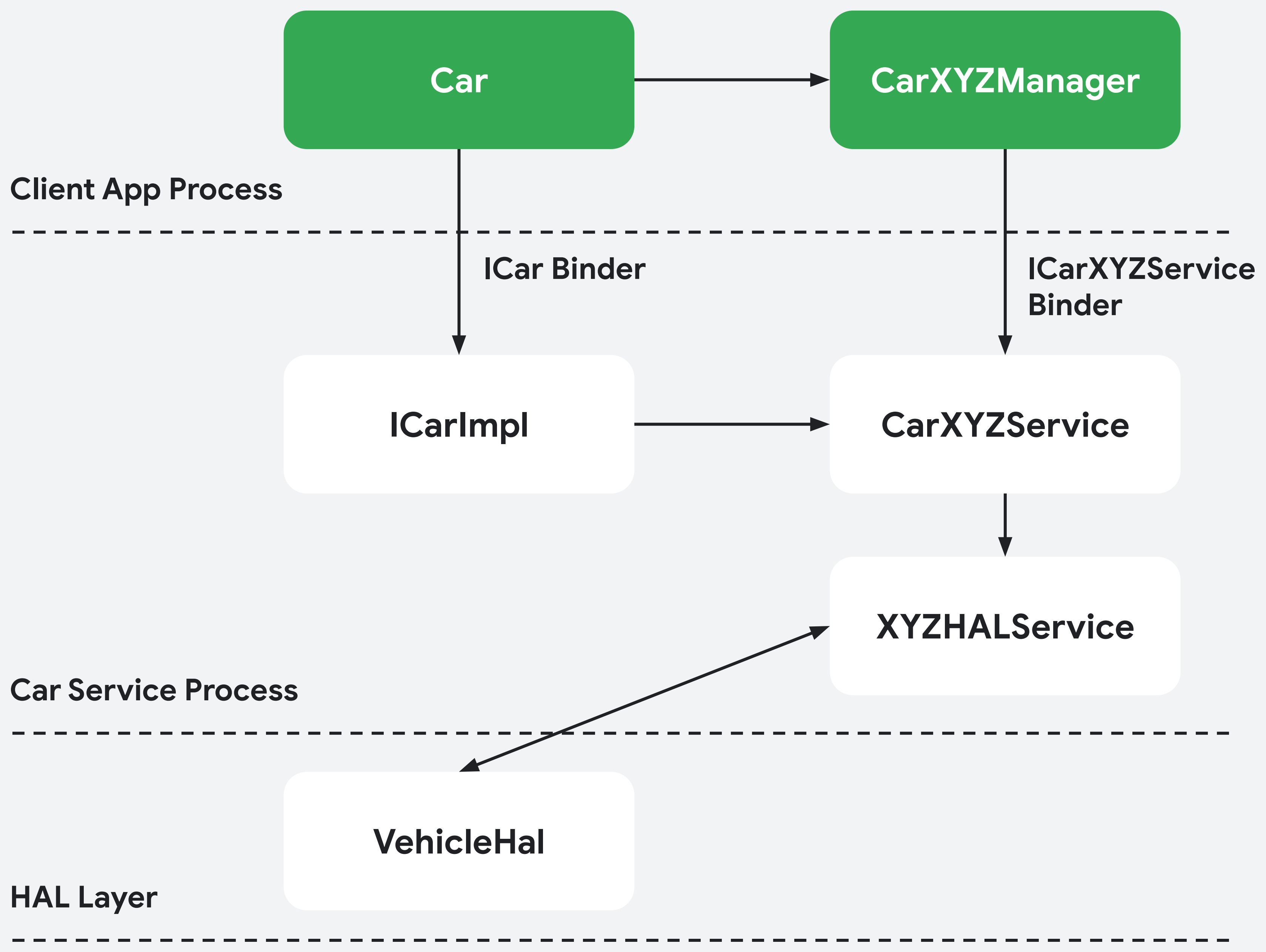
- Car Service provides the implementation for car specific functionalities and policies. It allows applications to interact with car hardware.
- Car Service (com.android.car) runs as user 0 and can serve all users regardless of user switching.
- Another version (CarPerUserService) runs for user 10+
  - This is because Bluetooth and LocationManager only work for the current user.
  - This service is only for internal car service usage and does not directly serve any Car APIs.
- When a Car object is created in a client app, it will contain an ICar Binder object so that it can communicate with Car Service.
- ICarImpl is responsible for creating the various CarXYZServices and implementing the ICar.aidl interface that apps interact with.



Car Service and Car Managers

What are Car Managers?

- A shared library used by applications to access car service functionalities.
- Expose Car specific API functionality.
- Every CarXYZManager has an associated CarXYZService counterpart.
- CarXYZManagers communicate with individual CarXYZServices using ICarXYZService binder interfaces defined by AIDL files.
- CarXYZServices also communicate with XYZHalServices within Car Service.



Car Service Inter-Process Communication

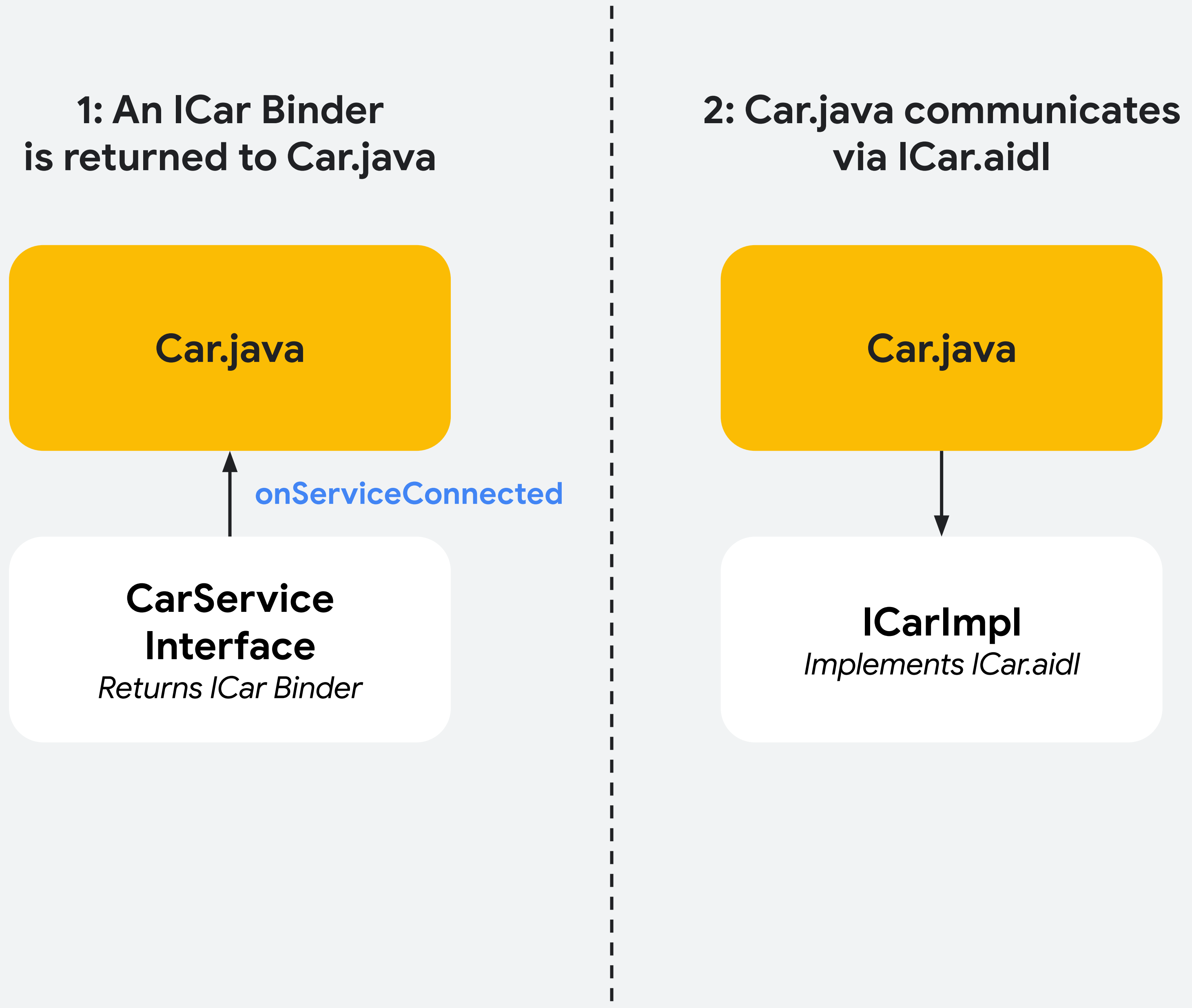
Communication between Apps and Car Service

- The Car SDK contains Car.java and other public classes. Any app can create a Car object. A client app will use the Car object to retrieve different managers (ex. CarUserManager, CarPowerManager).
- Internally, Car Service is an Android Service. It implements ICar.aidl in ICarImpl. Any app or service that connects to Car Service will receive an ICar binder object. The Car object will contain an ICar binder object to communicate with ICarImpl.
- When the onServiceConnected callback completes for Car object it will receive an ICar binder object.
- Every Car Manager will have a binder for its corresponding component. For example, CarUserManager will connect to CarUserService using ICarUserService.aidl.
- Important Note: Every service within Car Service is not a separate process, but a separate object within the Car Service process.

Car Service Inter-Process Communication

Communication between Car.java and Car Service

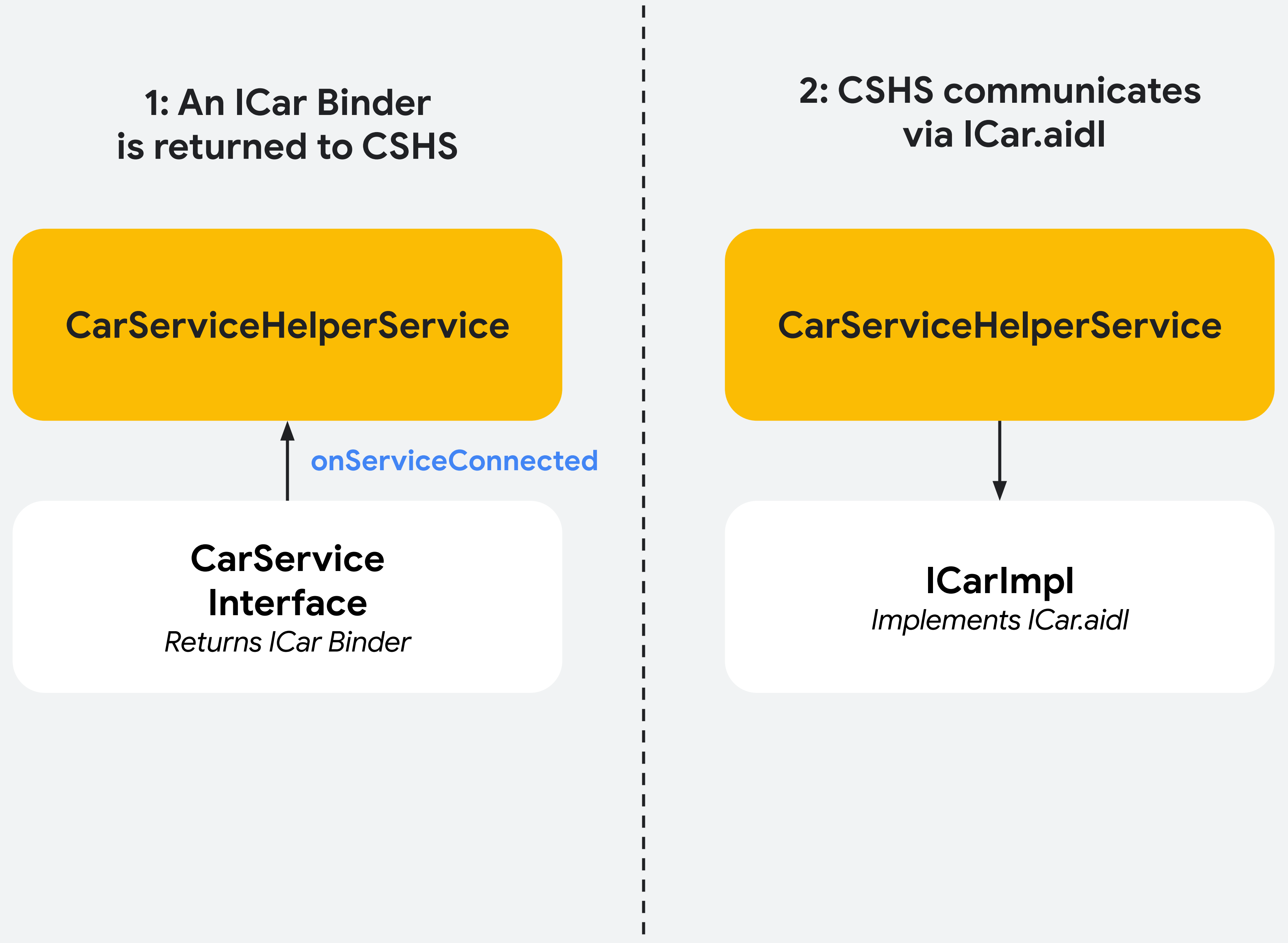
- The Car object receives the ICar binder object after the onServiceConnected callback completes.
- The Car object communicates with Car Service via the ICar binder object.



Car Service Inter-Process Communication

Communication between Car Service and System Server

- CarServiceHelperService's main purpose is to initialize Car Service.
- When the onServiceConnected callback completes, CarServiceHelperService will receive an ICar binder object as well to communicate with the Car Service process.



How the Car API Works

Accessing Car APIs

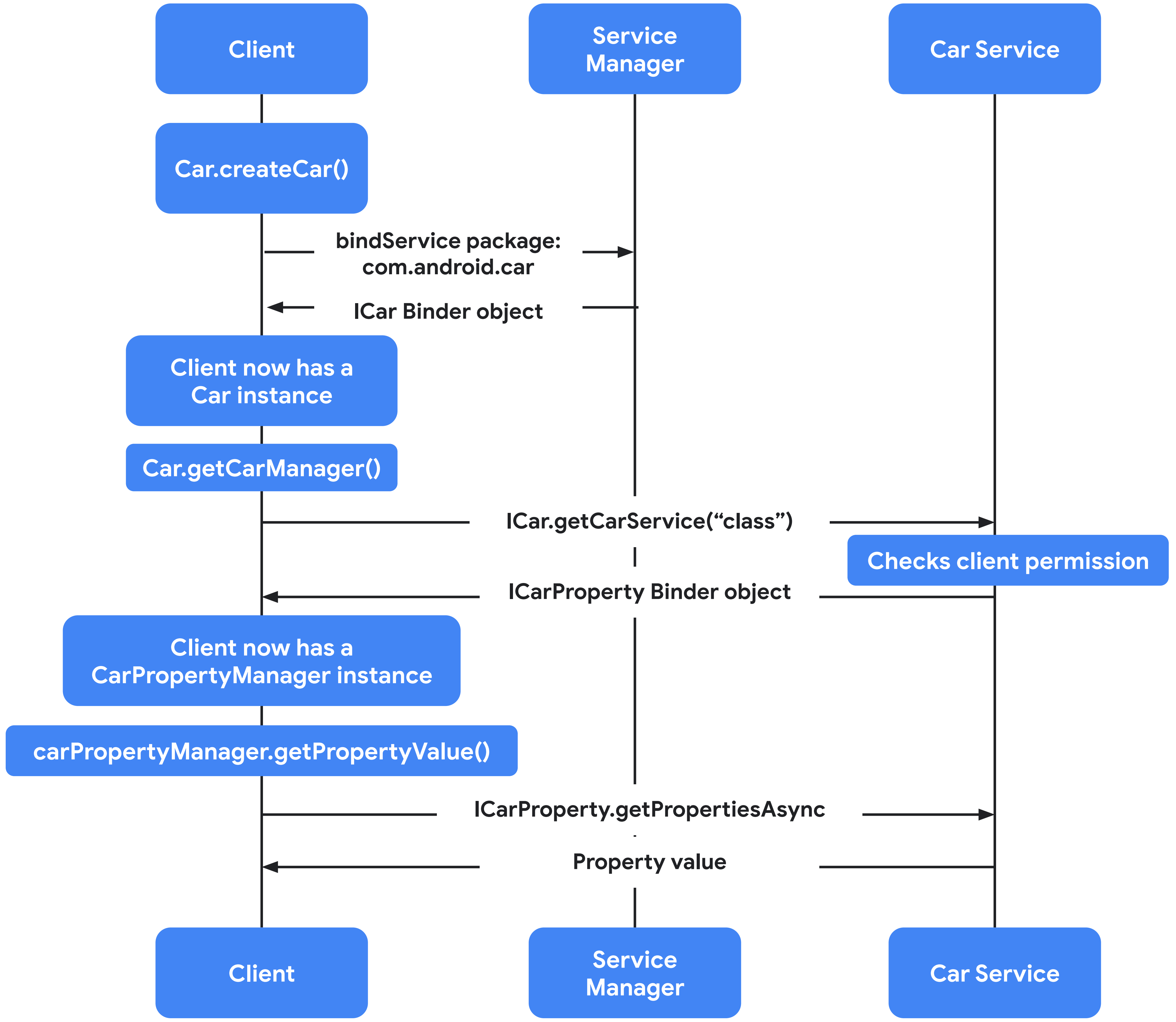
- Apps uses Car.createCar() to get a Car instance.
- If a statusChangeListener is not included in the arguments, the connection is synchronous. The client will be killed if Car Service dies.
- If a statusChangeListener is included in the arguments, the connection is asynchronous. The listener callback method will be called if Car Service is connected or disconnected (died).
- Apps use Car.getCarManager() to get a specific car manager, e.g. CarPropertyManager.
- Apps use APIs defined within car managers, e.g. CarPropertyManager.getProperty(propId, areaId)

```
private void init() {
  Car mCar = Car.createCar(mContext);
  mCarPropertyManager =
    (CarPropertyManager)
mCar.getCarManager(CarPropertyManager.class);
}

private int getProperty(int propId, int areaId) {
  return mCarPropertyManager.getProperty(propId, areaId);
}
```

What actually happens?

- Car.createCar() uses the Service Manager to get an ICar binder object that connects to Car Service.
- Car.getCarManager() uses ICar.getCarService to get a binder object for that specific car service. For example, ICarProperty for CarPropertyService if the client wants to obtain CarPropertyManager.

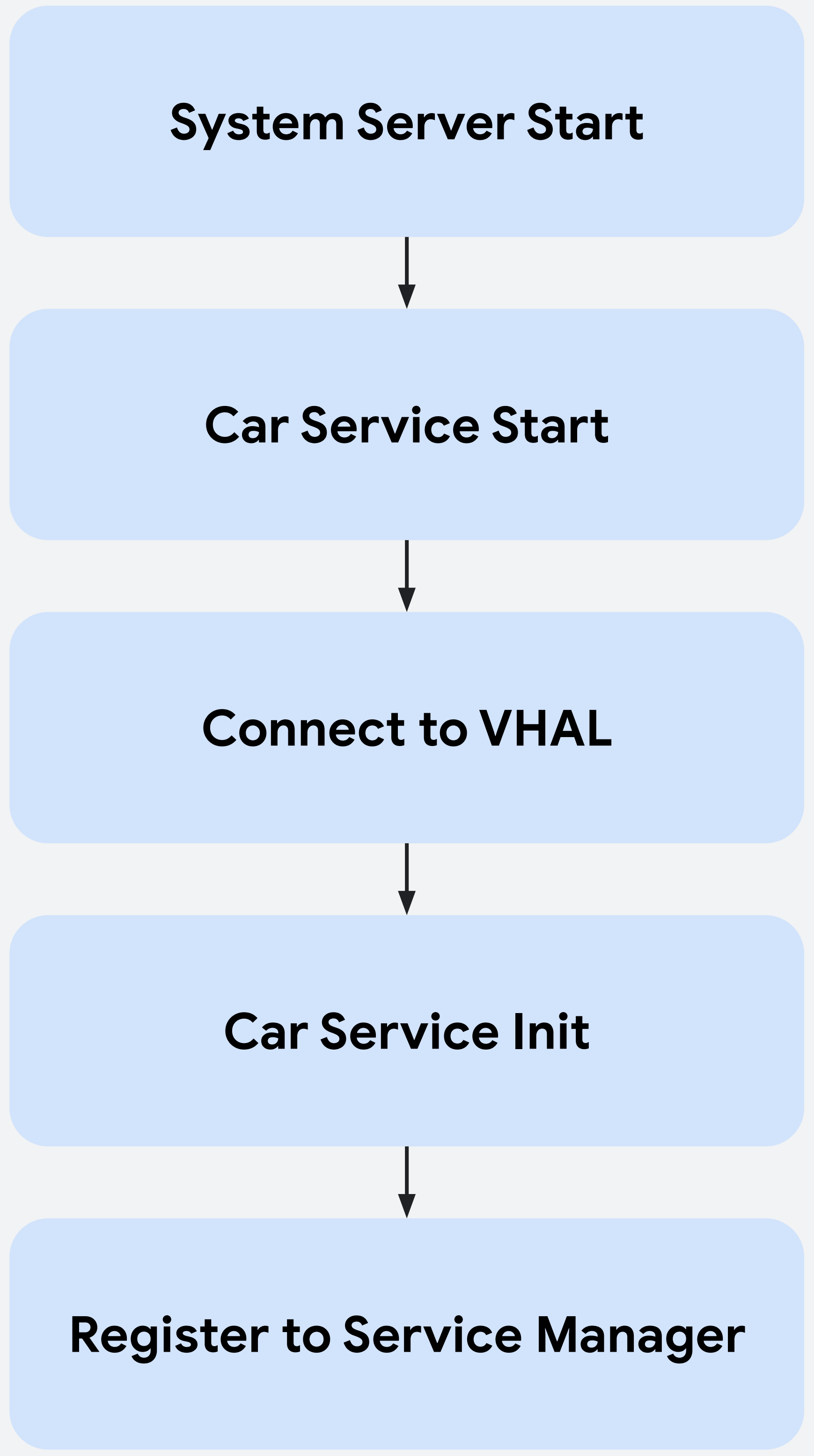
```
private void init() {
  Car mCar = Car.createCar(mContext);
  (CarPropertyManager) mCarPropertyManager =
    mCar.getCarManager(CarPropertyManager.class);
}

private int getProperty(int propId, int areaId) {
  return mCarPropertyManager.getProperty(propId, areaId);
}
```


Client
Service Manager
Car Service
Car.createCar()
bindService package: com.android.car
ICar Binder object
Client now has a Car instance
Car.getCarManager()
ICar.getCarService("class")
Checks client permission
ICarProperty Binder object
Client now has a CarPropertyManager instance
carPropertyManager.getPropertyValue()
ICarProperty.getPropertiesAsync
Property value
Client
Service Manager
Car Service

How the Car API works

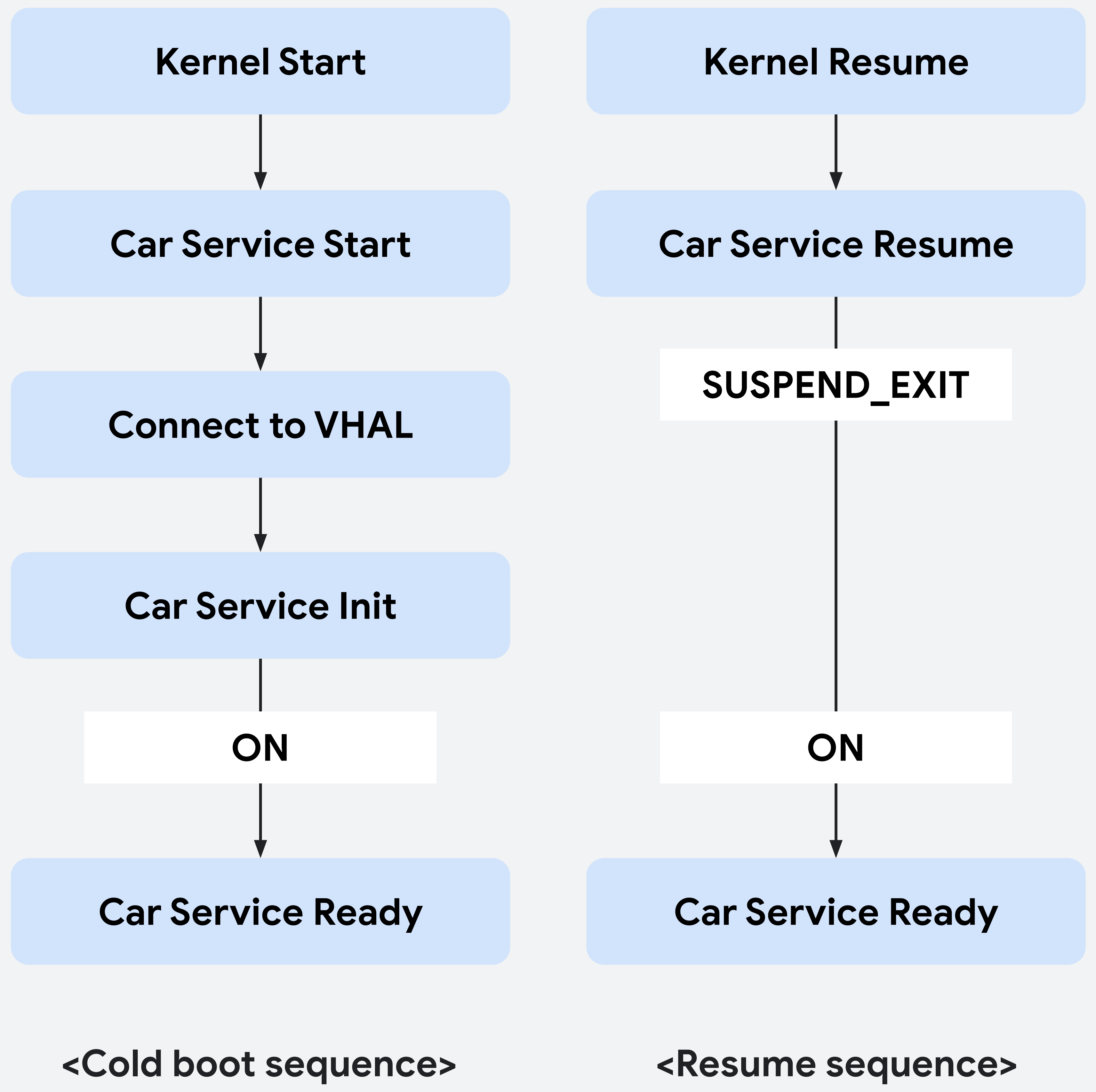
Car Service

- Car Service is launched by System Server via CarServiceHelperService during the device boot phase.
- If CarService crashes, it will be restarted.
- CarXXXServices are initialized sequentially during Car Service init.
- See 'CarService.initAllServices' trace in logcat.
- Car Service runs as 'com.android.car' process and is highly privileged.
- All permission checks for Car APIs are performed in Car Service.
- The majority of Car API business logic is implemented within Car Service.
- Car Service depends on multiple HALs, e.g. audio HAL, vehicle HAL etc. and a few native daemons, e.g. car watchdog daemon and car power policy daemon.



Car Service at Resume

- At resume (from RAM or disk), Car Service resumes working from the last snapshot
- No reconnection to VHAL
- No Car Service init
- No registration to ServiceManager
- **After Resume**, a power state change is sent to notify:
  - DEEP_SLEEP_EXIT for resume from RAM
  - HIBERNATION_EXIT for resume from disk
- **After Cold Boot**, Car Service is initialized as usual, with no signal sent



Car Service Crash Recovery

- In the event that the vehicle HAL crashes, Car Service finishes and restarts. It can also crash due to bugs or configuration issues.
- By default, all clients of Car Services are killed as there is no guarantee that an app will function properly after Car Service is restarted.
- All Car instances created using Car.createCar will be invalidated.
- For critical apps that should not crash, special handling is required.
- These apps should use Car.createCar with a LifeCycle Listener that handles Car Service crash or restart.
- All existing CarXYZManager instances are invalidated and apps should recreate all CarXYZManager instances and do necessary re-initialization.

Car Service Crash Recovery

- Example Car Service Crash Recovery implementation:

```
private void init() {
  // This waits for the car service connection to be established.
  Car.createCar(mContext, null, CAR_WAIT_TIMEOUT_WAIT_FOREVER,
this::onLifecycleChanged);
}

private void onLifecycleChanged(Car car, boolean ready) {
  synchronized (mLock) {
    if (ready) {
      mCarPropertyManager =
        (CarPropertyManager) car.getCarManager(CarPropertyManager.class);
      // Do initialization with CarXXXManager here, e.g. subscribe to
property events.
    } else {
      Log.e("Car service is disconnected");
      mCarPropertyManager = null;
      // Clear all internal state associated with CarXXXManager.
    }
  }
}
```

Using Car Managers

There are different managers for a wide array of use cases:

CarPackageManager

(android.car.content.pm) provides car specific APIs for package management.

CarUxRestrictions Manager

(android.car.drivingstate) is used to register and receive User Experience restrictions imposed based on a car's driving state.

CarPowerManager

(android.car.hardware.power) is used to receive power policy change notifications.

CarPropertyManager

(android.car.hardware.property) provides an interface for interacting with Vehicle specific functionalities abstracted as "Vehicle Properties".

CarInputManager

(android.car.input) is used to capture selected input events.

Using Car Managers

There are different managers for a wide array of use cases:

CarAudioManager

(android.car.media) is used for handling audio in a car, such as dynamic audio routing.

CarRemoteAccess Manager

(android.car.remoteaccess) allows applications to listen to remote task requests even while Android is not running.

CarWatchdogManager

(android.car.watchdog) allows applications to collect the latest system resource overuse statistics.

CarUserManager

(android.car.user) is used to manage users in a car.

Permissions

Privileged Permissions

- Privileged permissions declared within an app without explicit grant leads to a crash during boot up.
- Each package's privileged permissions need to be added to an XML file under:
    - [system|vendor]/etc/permissions as privapp-permissions*.xml

```
<permissions>
    <privapp-permissions package="com.android.example">
        <permission name="android.car.permission.CONTROL_CAR_CLIMATE"/>
    </privapp-permissions>
</permissions>
```

- Each app can also add its own XML file as <package-name>.xml and it will be added from the app's Android.bp:

```
prebuilt_etc {
    name: "allowed_privapp_com.android.settings",
    sub_dir: "permissions",
    src: "com.android.settings.xml",
    filename_from_src: true,
}
```

```
android_app {
    name: "Settings",
    required: ["allowed_privapp_com.android.settings"],
    privileged: true,
```

Privileged Permissions

- The specific error can be found using the following command:

```
adb logcat *:e | grep -i -A 10 "FATAL EXCEPTION IN SYSTEM PROCESS:
```

- Sample output can be found below:

```
04-08 18:58:56.556  4282  4282 E AndroidRuntime: *** FATAL EXCEPTION IN
SYSTEM PROCESS: main
04-08 18:58:56.556  4282  4282 E AndroidRuntime:
java.lang.IllegalStateException: Signature|privileged permissions not in
privapp-permissions allowlist: {com.android.car.messenger
(/system/priv-app/CarMessengerApp):
android.car.permission.ACCESS_CAR_PROJECTION_STATUS}
04-08 18:58:56.556  4282  4282 E AndroidRuntime:       at
com.android.server.pm.permission.PermissionManagerService.systemReady(Permi
ssionManagerService.java:4449)
...
```

Users in Cars

User Management in Cars

- Each app runs under a UID and can run multiple processes.
- UID != User ID : uid = fn(userid, appid) = 100,000 * userid + appid
- The System Server, Car Service and native services run in their own processes under the system UID.

Headless System User:

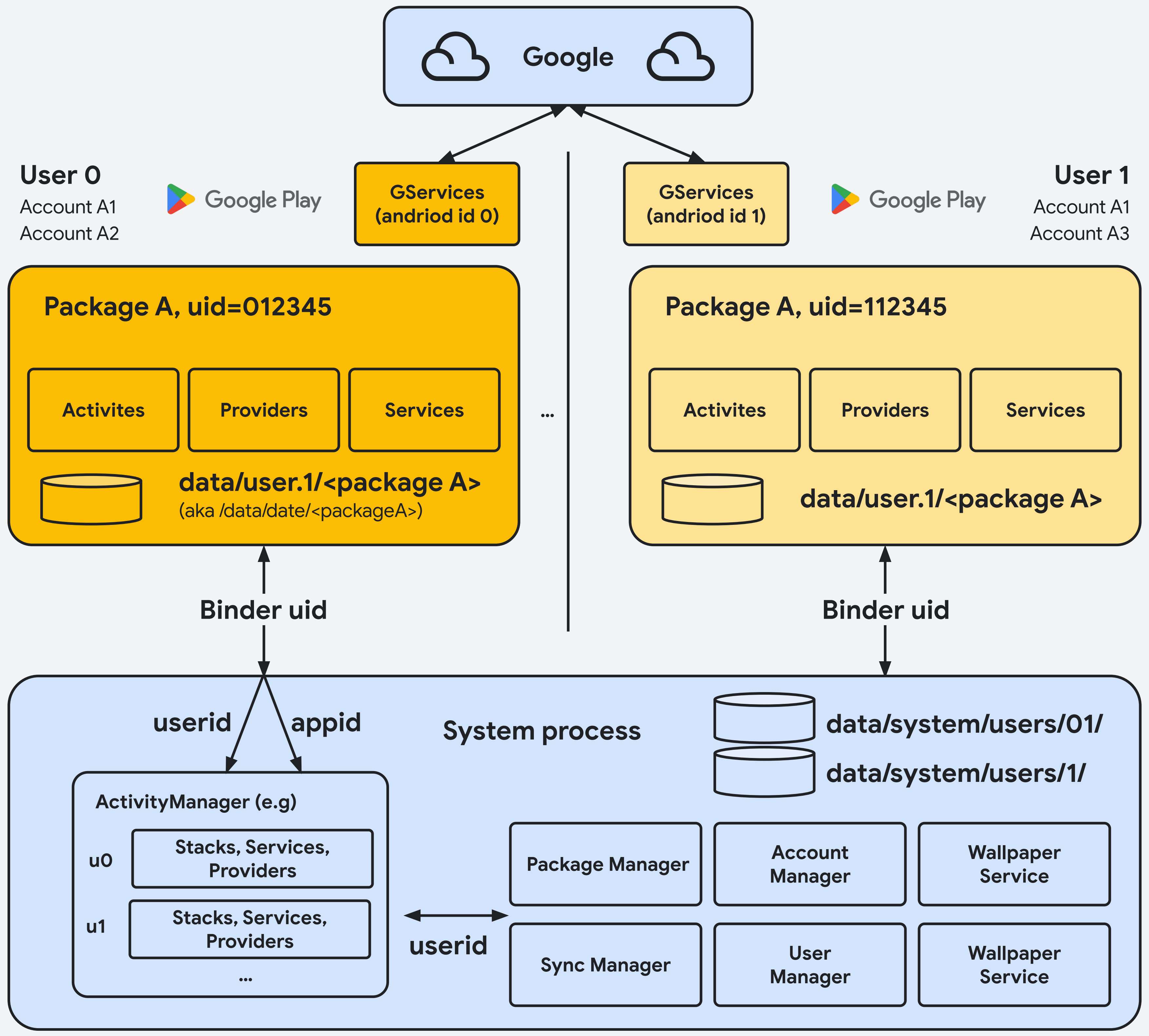
- User 0 is always running and runs without a UI (headless)
- This system user runs in the background

Other Users:

- Users 10 and above are started during boot up and have a UI, representing human users.
- In AAOS, user 0 is started first and then the current user is switched to user 10.

Multi-User Multi-Display (MUMD):

- For Automotive, there is also support for multiple concurrent users with their multiple displays. This concept is covered in its own dedicated GAPB presentation.



Boot User Initialization

How does the boot user get initialized?

- During Car Service's initialization, Car Service queries the User HAL to determine the initial (boot) user.
- Based on the User HAL response, Car Service will set the boot user.
- If there is no response from the User HAL or this functionality is not implemented, Car Service will by default set the last active user as the boot user.
- After the boot user is set, the framework switches the current user from user 0 to the boot user.

Listening for Car User LifeCycle Events

- CarUserManager exposes key user lifecycle events via UserLifecycleListeners
- System apps and services can add new listeners using CarUserManager.addListener()

.

Listeners can receive the following types of life cycle events:

- USER_LIFECYCLE_EVENT_TYPE_STARTING
- USER_LIFECYCLE_EVENT_TYPE_SWITCHING
- USER_LIFECYCLE_EVENT_TYPE_UNLOCKING
- USER_LIFECYCLE_EVENT_TYPE_UNLOCKED
- USER_LIFECYCLE_EVENT_TYPE_POST_UNLOCKED
- USER_LIFECYCLE_EVENT_TYPE_STOPPING
- USER_LIFECYCLE_EVENT_TYPE_STOPPED
- USER_LIFECYCLE_EVENT_TYPE_CREATED
- USER_LIFECYCLE_EVENT_TYPE_REMOVED
- USER_LIFECYCLE_EVENT_TYPE_VISIBLE
- USER_LIFECYCLE_EVENT_TYPE_INVISIBLE

User Switch Monitor App

- This test application can be used for debugging user lifecycle events.
- The app exists at packages/services/Car/tests/UserSwitchMonitorApp/
- Follow the below to install the app:

```
$ m UserSwitchMonitorApp
$ adb install --user 0
$OUT/system/app/UserSwitchMonitorApp/UserSwitchMonitorApp.apk
$ adb shell pm grant --user 0 com.google.android.car.userswitchmonitor
android.permission.INTERACT_ACROSS_USERS
$ adb shell am start-foreground-service --user 0
com.google.android.car.userswitchmonitor/.UserSwitchMonitorService
```

User Switch Monitor App

ADB Logcat Output:

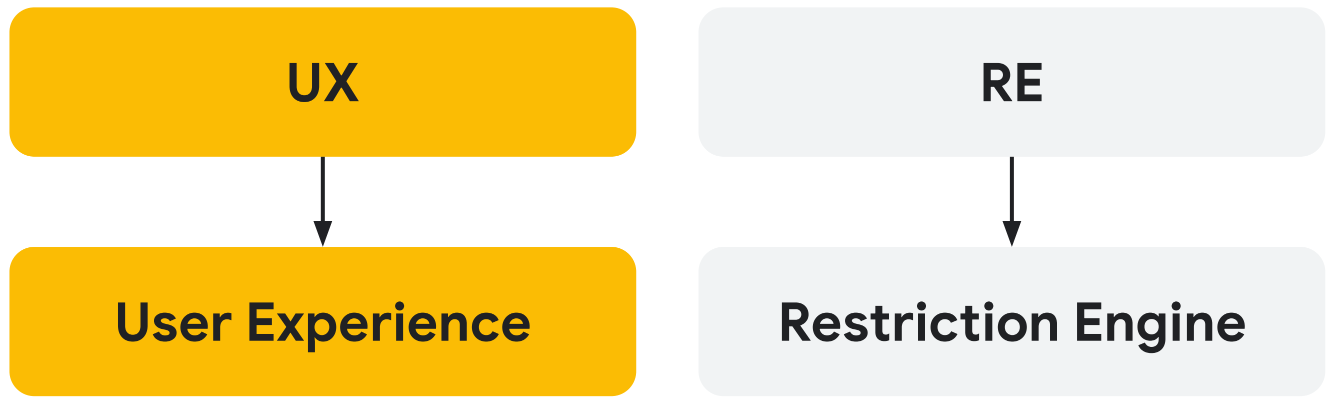
```
$ adb logcat UserSwitchMonitor *:s
... UserSwitchMonitor: onEvent(0): Event[type=STARTING,user=11]
... UserSwitchMonitor: onEvent(0): Event[type=SWITCHING,from=10,to=11]
... UserSwitchMonitor: onEvent(0): Event[type=UNLOCKING,user=11]
... UserSwitchMonitor: onEvent(0): Event[type=UNLOCKED,user=11]
... UserSwitchMonitor: onEvent(0): Event[type=STOPPING,user=10]
... UserSwitchMonitor: onEvent(0): Event[type=STOPPED,user=10]
... UserSwitchMonitor: onEvent(0): Event[type=STARTING,user=10]
... UserSwitchMonitor: onEvent(0): Event[type=SWITCHING,from=11,to=10]
```

Dumpsys Output:

```
$ adb shell dumpsys activity service
com.google.android.car.userswitchmonitor/.UserSwitchMonitorService
Received 12 events:
      1: Event[type=STARTING,user=11]
      2: Event[type=SWITCHING,from=10,to=11]
      3: Event[type=UNLOCKING,user=11]
      4: Event[type=UNLOCKED,user=11]
      5: Event[type=STOPPING,user=10]
      6: Event[type=STOPPED,user=10]
      7: Event[type=STARTING,user=10]
      8: Event[type=SWITCHING,from=11,to=10]
```

Demystifying UXRestrictions

What is UX RE?



A part of Car Service that orchestrates:

1. **Different kinds** of restrictions on user experiences,
2. **Selectively,**
3. **And under certain conditions**

What is UX RE?

Different kinds

Type 1: Restricting a no_keyboard, no_camera, no_video etc. (must be enforced by apps themselves)

Type 2: Restricting the entire UI if it is not distraction optimized.

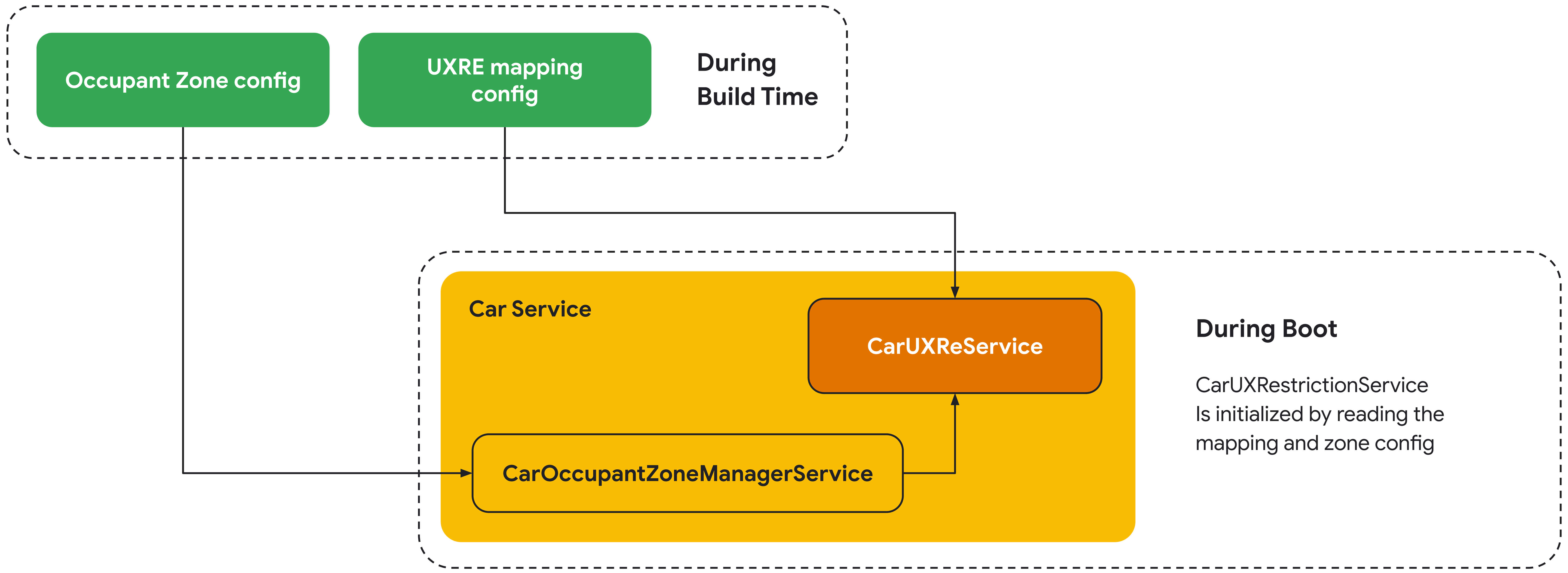
Selectively

The UXREngine selectively applies restrictions only to the relevant displays as set in the configuration.

Certain conditions

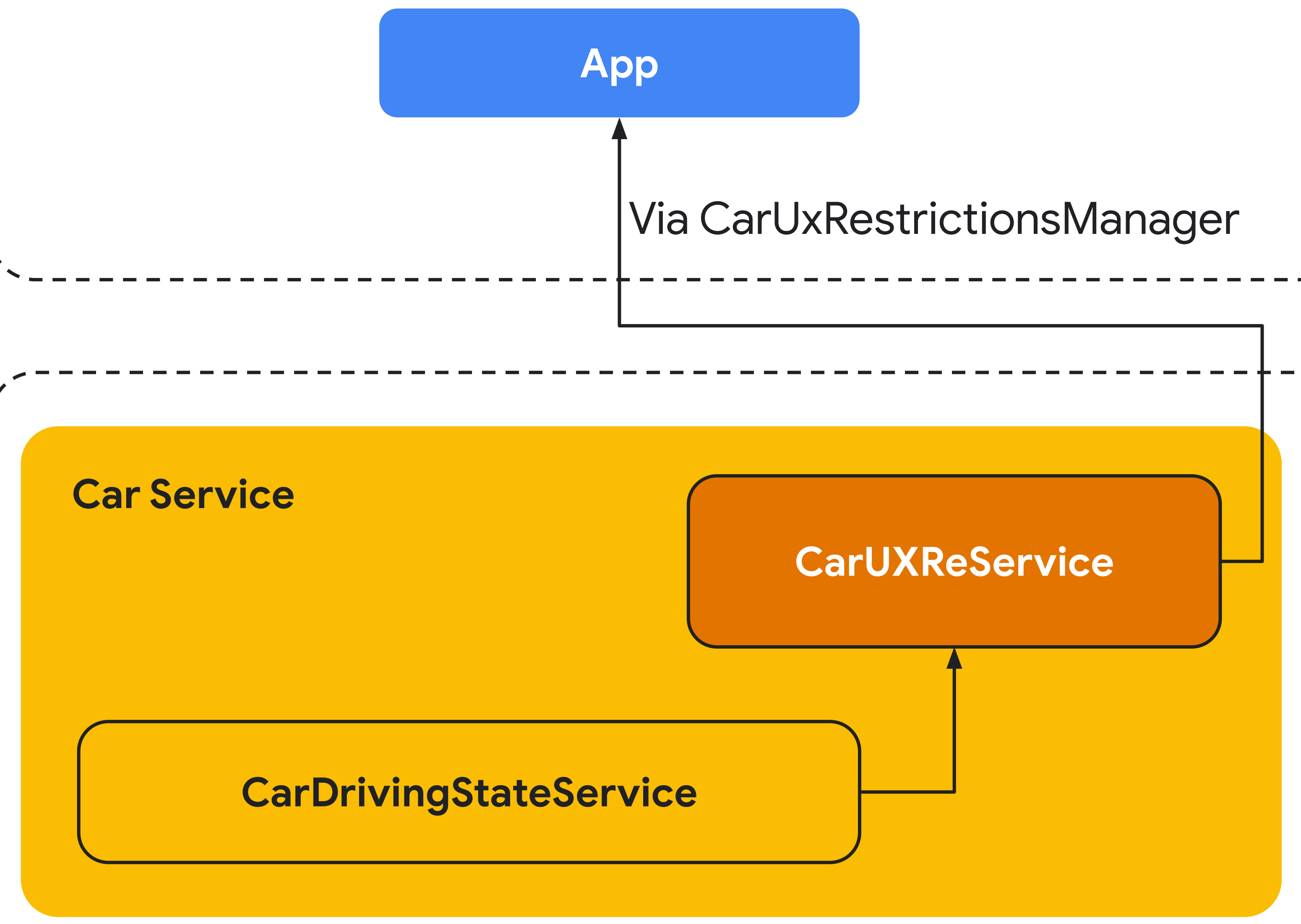
The conditions are derived from a combination of Driving State + UXRe Configuration + Occupant Zone Configuration

UXRestriction Lifecycle



CarUXRestrictionService Is initialized by reading the mapping and zone config

UXRestriction Lifecycle @ Runtime: Type 1 and Type 2



At Runtime

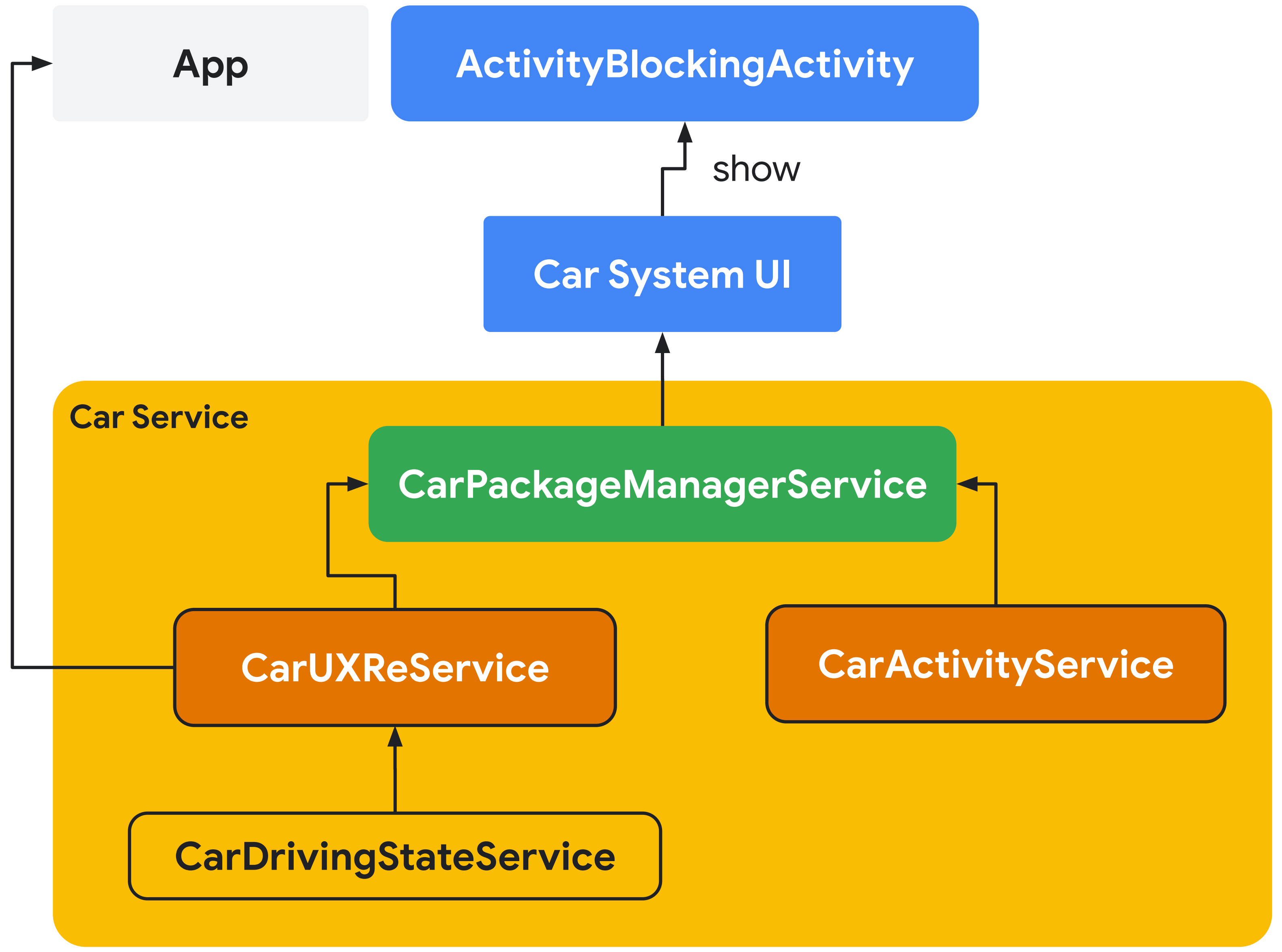
Examples include:

- Gboard restricts the keyboard when it receives the no_keyboard UXR.
- A CAL app restricts string length when a relevant UXR is received.
- Car Service may block the entire UI.

At Runtime

- Apps register a callback that receive new restrictions if they have changed.
- Car Service receives driving state changes from the CarDrivingStateService.
- New restrictions are generated using the UXRE mapping and driving state.

UXRestriction Lifecycle: Type 2



At Runtime

- Car Service receives driving state changes from CarDrivingStateService.
- CarUXReService computes restrictions from the UXRe mapping config and the driving state.
- CarPackageManagerService determines if the current activity is distraction optimized.
- If the current activity is not distraction optimized, then an ActivityBlockingActivity is launched that blocks the offending activity.

Thank you

Google
Automotive
Partner
Bootcamp