OCPP 2.0.1 vs OCPP 1.6: Comprehensive Guide for EV Charger Board Developers

Introduction

The Open Charge Point Protocol (OCPP) is the de-facto standard for communication between EV charging stations (charge points) and a central management system (CSMS). OCPP 1.6 laid the groundwork for basic interoperability and smart charging, while **OCPP 2.0.1** (released 2020) is a major upgrade with new features and structural changes. This guide helps firmware engineers and software developers transition from OCPP 1.6 to OCPP 2.0.1, explaining **core differences**, **changes in transaction flows**, **deprecated vs. new commands**, and providing **Java code examples** and **best practices**. We also include a **glossary of OCPP 2.0.1 messages** with brief descriptions for quick reference.

Core Differences Between OCPP 1.6 and OCPP 2.0.1

OCPP 2.0.1 is not backwards compatible with 1.6 and introduces significant improvements in architecture and capabilities (OCPP 1.6 vs 2.0.1 - Key Differences & Updates - ChargePanel) (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). Key differences include:

- **Device Model & Station Architecture:** OCPP 1.6 models a charger as a station with multiple independent connectors. In OCPP 2.0.1, there is a **three-tier device model: Charging Station -> EVSE -> Connector** (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium). An **EVSE** (Electric Vehicle Supply Equipment) represents a logical charging unit that can manage one session at a time, even if it has multiple connectors (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). This hierarchy allows a station to clearly describe its components (connectors, power modules, etc.) and capabilities to the CSMS (plug-and-play). It improves handling of multi-connector chargers (e.g., dual-cable DC fast chargers) which in 1.6 had no formal EVSE concept. Inventory reporting of all components and configuration is now possible, and any event/error is reported with a reference to the specific component, far surpassing the limited StatusNotification in 1.6 (What is new in OCPP 2.0.1) (What is new in OCPP 2.0.1). In practice, developers must update their firmware to maintain a **component tree** (station, each EVSE, each connector) and use new messages to report component status and metrics.

- **Unified Transaction Handling:** OCPP 1.6 used separate messages for starting and stopping a session (`StartTransaction` and `StopTransaction`), plus periodic `MeterValues` and `StatusNotification` updates during a transaction. OCPP 2.0.1 **consolidates all transaction-related events into a single `TransactionEvent` message** (What is new in OCPP 2.0.1). This one message (with different event types) carries the info that was spread across multiple 1.6 messages, reducing protocol overhead. The 1.6 `StartTransaction`, `StopTransaction`, and transaction-related `MeterValues` are **replaced by `TransactionEvent`** (with eventType values *Started*, *Updated*, *Ended*) (What is new in OCPP 2.0.1). The legacy `StatusNotification` is still present in 2.0.1 **only for non-transaction events** (e.g. connector availability or fault status) (What is new in OCPP 2.0.1). Additionally, OCPP 2.0.1 gives charging stations control of transaction IDs and sequence numbers for offline scenarios, eliminating the need for temporary IDs from the CSMS when offline (What is new in OCPP 2.0.1). (In 1.6, the CSMS issued transaction IDs, complicating offline operation.) Overall, transactions are more flexibly tracked, and offline recovery is improved by sequence numbering of events (What is new in OCPP 2.0.1).

- **Enhanced Messaging Efficiency:** OCPP 1.6 introduced JSON over WebSockets which greatly reduced overhead compared to SOAP. OCPP 2.0.1 goes further by supporting **WebSocket compression**, allowing messages to be compressed to reduce bandwidth (OCPP 1.6 vs 2.0.1 - Key Differences & Updates - ChargePanel). Additionally, OCPP 2.0.1 allows **message batching** (sending multiple OCPP calls in one WebSocket frame) for efficiency (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium). These help high-traffic sites or cellular-connected chargers minimize data usage. Also, OCPP 2.0.1 messages are organized into *functional profiles* (or feature blocks) — developers can implement the *Core* profile and add others (like Smart Charging, Security, ISO 15118) as needed (What is new in OCPP 2.0.1) (What is new in OCPP 2.0.1), making the implementation modular.

- **Smart Charging & Energy Management:** Both versions support smart charging, but 2.0.1 is far more advanced. OCPP 1.6 allowed setting charging profiles (TxProfile, TxDefaultProfile, ChargingStationMaxProfile) but they were relatively static and mainly controlled by the CSMS (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). OCPP 2.0.1 introduces **dynamic smart charging** – profiles can be updated in real-time based on grid load, energy price, or EV needs (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). New capabilities include **recurring schedules** (daily/weekly profiles) and integration with **ISO 15118** for **Plug & Charge** and **Vehicle-to-Grid (V2G)** support (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). Importantly, OCPP 2.0.1 can take direct input from the EV about its charging needs (e.g. desired energy, departure time) and handle **load balancing** between EVSEs natively (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium) (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium). In practice, this means the charger firmware should handle new messages like `NotifyEVChargingNeeds` (car communicates needs) and possibly adjust charging based on local controller input. The charger also needs to handle more complex profile scenarios, but the core smart charging commands (`SetChargingProfile` , etc.) remain (with extended parameters for things like charging power constraints from the EV).

- **Improved User Interaction:** OCPP 1.6 was primarily designed for RFID card authentication and didn't support in-charger user interfaces beyond simple status lights. OCPP 2.0.1 adds features to enhance the **EV driver's experience** (OCPP 1.6 vs 2.0.1 - Key Differences & Updates - ChargePanel) (OCPP 1.6 vs 2.0.1 - Key Differences & Updates - ChargePanel). For example, a CSMS can now send display messages to the station (`SetDisplayMessage`) to show instructions or information on a screen. It supports multiple languages for messages (What is new in OCPP 2.0.1) and can show pricing or cost information to the user **before, during, and after a session** (OCPP 1.6 vs 2.0.1 - Key Differences & Updates - ChargePanel) (e.g. display the running cost in real-time, or final cost when charging ends). More authentication methods are supported: **credit card, NFC, app-based tokens, Plug & Charge, even PIN code** in addition to RFID (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium) (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium). For developers, this means implementing new request/response pairs for display management and handling different token types in `Authorize` requests. Chargers may require additional hardware (displays, keypads, etc.) to fully leverage these features, but OCPP 2.0.1 provides the protocol support.

- **Security Enhancements:** Security is a major focus in OCPP 2.0.1. OCPP 1.6 offered basic TLS encryption (often optional) and simple authentication (HTTP Basic or station-specific IDs), which are insufficient by modern standards (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium) (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium). OCPP 2.0.1 defines **three Security Profiles**:
  - *Profile 1:* No encryption (for local trusted networks only, generally **not recommended**).
  - *Profile 2:* TLS encryption (server-authenticated), equivalent to 1.6's typical TLS usage.

- *Profile 3:* TLS with **mutual authentication** using X.509 certificates (both CSMS and station authenticate each other) (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison).

The highest profile mandates installing digital certificates on the station. OCPP 2.0.1 includes a suite of **certificate management** messages, allowing the CSMS to remotely install and update certificates (for example, the station's own client certificate, or a trusted CA list for validating the CSMS) (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). There are messages like `InstallCertificate`, `DeleteCertificate`, `GetInstalledCertificateIds`, and a flow for certificate signing (`SignCertificate` request from station, followed by `CertificateSigned` from CSMS). Additionally, OCPP 2.0.1 stations maintain a **security event log** and can send `SecurityEventNotification` for incidents (e.g. invalid login attempts, firmware signature check failures) (What is new in OCPP 2.0.1). For developers, implementing OCPP 2.0.1 means integrating a **public-key infrastructure (PKI)** on the device: managing keys and certificates in secure storage and handling these new messages to stay compliant with network security policies. (Note: Some of these security improvements have been back-ported as optional patches to OCPP 1.6-J, but they are native in 2.0.1 (What is new in OCPP 2.0.1).)

- **Firmware & Diagnostics Management:** OCPP 1.6 did allow remote firmware updates (`UpdateFirmware`) and diagnostics download (`GetDiagnostics`), but OCPP 2.0.1 makes these processes more secure and transparent. Firmware updates in 2.0.1 are expected to be delivered via **secure download (HTTPS)** and are accompanied by **digital signatures** to verify integrity (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). The station should not install firmware unless the signature is valid (the CSMS provides the signature or a certificate to verify it). OCPP 2.0.1 adds granular status updates during firmware update: the station sends `FirmwareStatusNotification` with states like *Downloading, Downloaded, Installing, Installed, Failed*, etc., giving the operator real-time insight into the update process (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). Diagnostics retrieval is generalized through `GetLog` and `LogStatusNotification` messages (replacing 1.6's `GetDiagnostics/DiagnosticsStatusNotification`). Stations can send not just one type of log but various logs (e.g. regular diagnostics, security log, etc.) requested by the CSMS, and report the upload status via `LogStatusNotification` ([PDF] OCPP 2.0.1: Part 2 - Errata - Regulations.gov). All these improvements require developers to implement robust file handling: the station must download files over network, verify signatures, possibly store logs, and handle retries or failures gracefully.

- **Miscellaneous Improvements:** OCPP 2.0.1 introduces many other improvements:
  - **Offline Behavior:** As noted, stations now generate transaction IDs themselves and keep sequence numbers, simplifying offline transaction queueing (What is new in OCPP 2.0.1). There is no strict requirement to send events in chronological order anymore (sequence numbering allows reordering on the server) (What is new in OCPP 2.0.1).
  - **Start/Stop Transaction Triggers:** OCPP 2.0.1 allows **configurable triggers** for when a transaction is deemed started or stopped (What is new in OCPP 2.0.1) (What is new in OCPP 2.0.1). For example, an implementation can configure that the transaction *starts when the EV plugs in* (before authentication) and ends when unplugged, or stick to the classic behavior (start at authorization, stop at session termination). It's even possible to use **parking bay occupancy** (via a sensor) as a start/stop trigger (What is new in OCPP 2.0.1). This flexibility means firmware should be prepared to send TransactionEvent(Started) at different moments based on configuration.
  - **Reservation Improvements:** Similar to 1.6, OCPP 2.0.1 supports reserving chargers (`ReserveNow` and `CancelReservation`) for a specific EV driver. However, with the EVSE model, a reservation can target a specific EVSE (which may correspond to one connector or a group of connectors).
  - **New Message Types:** OCPP 2.0.1 includes new messages for specific functions, like `NotifyEvent` (station reports an internal event or error with component info), `ClearedChargingLimit` (station confirms a charge

limiting restriction was cleared), `GetChargingProfiles` / `ReportChargingProfiles` (to query active charging schedules on the station), `CustomerInformation` (to retrieve or send customer-related data, e.g. for auditing or billing), and several others. These will be outlined in the glossary.

In summary, OCPP 2.0.1 is a **superset with a more complex but powerful design**. It brings richer device modeling, better transaction control, stronger security, and extensibility for future needs (like bi-directional charging). Migrating from 1.6 to 2.0.1 will require firmware refactoring, but it "future-proofs" the charger to participate in advanced charging networks (supporting features like Plug & Charge, finer remote control, and integration with energy management systems) (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison) (What is new in OCPP 2.0.1).

Transaction Flow: OCPP 1.6 vs OCPP 2.0.1

One of the most significant changes in OCPP 2.0.1 is **how a charging transaction is handled in terms of message flow**. Below we compare an example sequence of messages for starting and stopping a charging session in OCPP 1.6 and OCPP 2.0.1.

OCPP 1.6 Transaction Flow Example

Scenario: An EV driver plugs in and authenticates via RFID card, charges for a while, then unplugs to stop the session.

1. **Boot Notification:** When the station first came online or rebooted, it sent a `BootNotification` to register with the central system (CSMS). (This step is prior to the transaction sequence but essential for context.)
2. **Idle Status:** The station likely sent a `StatusNotification` indicating the connector is **Available** (idle).
3. **Authorize:** The driver presents an RFID card. The station sends an `Authorize.req` with the IdTag to the CSMS, which replies `Authorize.conf` (accepted or rejected).
4. **Start Transaction:** After a successful authorization and when the EV is plugged, the station sends `StartTransaction.req` to the CSMS with details: connector ID, IdTag, timestamp, and starting meter value. The CSMS responds with `StartTransaction.conf` including a **transactionId** (assigned by the CSMS) and an `IdTagInfo` status (which could e.g. confirm accepted or tell if the tag was whitelisted).
5. **Status Notification (Charging):** The station sends `StatusNotification` to report that the connector status changed from "Available" to "Charging" (or "Occupied"). This may occur around the time the transaction starts (exact order can vary – some implementations send it just before StartTransaction).
6. **Meter Values (Periodic):** During the charging session, if configured, the station sends periodic `MeterValues.req` messages to report intermediate meter readings (e.g. every minute or every 10% of battery). Each contains one or more sampled values (energy, power, etc.). The CSMS replies `MeterValues.conf` (usually an empty payload confirming receipt). These interim values help the CSMS track consumption in near real-time.
7. *(Optional other notifications:* If any error or significant event occurs (e.g. a fault), the station might send a `StatusNotification` with status "Faulted", etc. Similarly, if a reservation or remote command interrupts, other messages would be involved.)*
8. **Stop Transaction:** When the driver unplugs or selects "stop", the station sends a `StopTransaction.req` to the CSMS. This message includes the transactionId (from the StartTransaction), the reason for stopping (e.g. "EVDisconnected"), timestamp, and final meter reading. It may also include a transaction summary (e.g. cost or duration if calculated by the station). The CSMS responds with `StopTransaction.conf` (often with an IdTagInfo to indicate if further charging is authorized for that IdTag).
9. **Status Notification (Available):** After the session ends and the connector is free, the station sends a `StatusNotification` to mark the connector status back to **Available** (or "Finishing/Available").

In OCPP 1.6, the **CSMS is heavily involved** in assigning transaction IDs and confirming each stage. The station relies on multiple message types to convey the full picture of a session. There are separate pathways for status vs. meter data vs. transaction state.

OCPP 2.0.1 Transaction Flow Example

The same scenario in OCPP 2.0.1 uses the new unified messages:

1. **Boot Notification:** (Same concept as 1.6) Station registers with `BootNotification` at startup with its details; CSMS responds with acceptance and heartbeat interval, etc.
2. **Idle Status:** Station sends an initial `StatusNotification` (if needed) to declare connectors/EVSEs are Available. (Note: StatusNotification in 2.0.1 is primarily for availability or fault states, not for transaction state changes.)
3. **Authorize (optional):** Driver presents RFID. Station sends `Authorize.req` with the IdToken. CSMS responds `Authorize.conf` with `IdTokenInfo` (status, possibly an allowed energy limit or group ID). In 2.0.1, authorization might also happen implicitly via Plug & Charge if supported (in which case an `Authorize` message might not be needed because the EV's certificate is used). Assuming RFID for this example, we use `Authorize`.
4. **Transaction Event (Started):** Once the EV is plugged in and/or authorization is confirmed (depending on configuration of start triggers), the station sends **`TransactionEvent.req`** with `eventType = Started`. This single message replaces the StartTransaction and accompanying status in older versions (What is new in OCPP 2.0.1). It carries:
   - The **timestamp** of event,
   - A generated **transactionId** (now created by the station itself),
   - The EVSE ID and connector ID involved,
   - The IdToken used (if any, for an authorized start),
   - `triggerReason` (e.g. "Authorized" or "CablePluggedIn" depending on what triggered the start),
   - Initial meter value, and other relevant details (e.g. charging priority, EV information if available).
     The CSMS responds with `TransactionEvent.conf` acknowledging it (and may include an updated "remoteStartId" or monitoring information if this start was triggered remotely, etc.). At this point, the transaction is officially begun. (No separate transactionId issuance from CSMS – the station's ID is used.)
5. **(No separate status notification for charging):** Unlike OCPP1.6, the station typically does **not send a StatusNotification for "Charging"**. The CSMS knows the EVSE is occupied because it received a TransactionEvent Started. The **connector's status is implicitly occupied** by an active transaction. A `StatusNotification` would only be sent if, say, the connector or EVSE had a fault or became unavailable outside of the transaction context.
6. **Transaction Event (Updated):** During the charging session, the station can send periodic updates using `TransactionEvent.req` with `eventType = Updated`. These might be sent at intervals or on significant changes. An Updated event can carry:
   - Current meter reading(s),
   - `triggerReason` such as "MeterValuePeriodic" or "EnergyLimitReached", etc.,
   - `seqNo` (sequence number of this event in the transaction),
   - Optional info like current charging state (e.g. "Charging" or "SuspendedEV"), the SOC or remaining time (if the EV provided it via ISO 15118), and so on.
     The CSMS responds to each with `TransactionEvent.conf`. Because these messages are flexible, they **eliminate the separate MeterValues message** – all the data can be part of TransactionEvent payload (What is new in OCPP 2.0.1). The frequency and triggers for sending updates are configurable in the station (possibly controlled by the CSMS via monitoring profiles). For example, if using **Charging Profiles**, the station might send an Updated event whenever the limit changes or whenever a period in the schedule is reached.

7. *(Optional events:* If an error occurs or the session is paused, the station can send a `TransactionEvent` with `eventType = Updated` and an appropriate `triggerReason` (like "ChargingStationFaulted" or "EVCommunicationLost") to inform the CSMS. There is also a `NotifyEvent` message for non-transaction component events, but for anything affecting the session, a TransactionEvent is typically used so it ties to the transaction.)*
8. **Transaction Event (Ended):** When the session terminates (EV disconnected, or remote stop, etc.), the station sends `TransactionEvent.req` with `eventType = Ended`. This final event includes:
    - The last meter reading,
    - `triggerReason` (e.g. "EVDisconnected" or "RemoteStop" etc.),
    - The total transaction duration or energy might be included if calculated,
    - A reason for stopping (there is a field for `transactionData.stoppedReason`, e.g. "Local" or "EV" or "Deauthorized" etc.),
    - The same transactionId and a final `seqNo`.
      The CSMS replies with `TransactionEvent.conf`. After this, the CSMS knows the transaction is finished and can process billing, etc. The station can now consider the EVSE free.
9. **Post-Transaction Status:** If the EV is unplugged, the station will likely send a `StatusNotification` for that connector/EVSE to indicate it's Available again (or perhaps "Occupied"->"Available"). In many cases, the CSMS can infer availability from the TransactionEvent Ended, but typically a status update for clarity or any change (like connector physically went from occupied to available) is still sent.

As shown, OCPP 2.0.1's flow revolves around **fewer message types**. The `TransactionEvent` carries what was in Start/StopTransaction + MeterValues + some StatusNotifications (What is new in OCPP 2.0.1). The station has more autonomy (it decides transaction IDs and when to send updates). For developers, implementing this flow means you will create and parse the `TransactionEvent` payload carefully for each phase of a session. The payload is more complex than the old messages (nested structures for idToken, chargingState, etc.), but it's logically consistent. The CSMS logic also simplifies on receiving side as it gets a single stream of events it can reconstruct via sequence numbers.

Diagrammatic Summary: According to the Open Charge Alliance, **"All StartTransaction, StopTransaction, and transaction-related MeterValue and StatusNotification messages are replaced by a 'TransactionEvent' message."** (What is new in OCPP 2.0.1). An official mapping for a simple case would be:

- *OCPP 1.6:* Authorize -> **StartTransaction** -> MeterValues (repeated) -> **StopTransaction**
- *OCPP 2.0.1:* Authorize -> **TransactionEvent(Started)** -> **TransactionEvent(Updated)** (repeated) -> **TransactionEvent(Ended)**

The overall sequence of initiating a transaction (Authorize -> start session -> stop session) remains, but the messaging is consolidated.

Remote Start/Stop: Another flow difference is remote control of transactions. In 1.6, the CSMS would send `RemoteStartTransaction` and the station would respond and then proceed to start a transaction (and similarly `RemoteStopTransaction`). In 2.0.1 these are renamed to `RequestStartTransaction` and `RequestStopTransaction` with more detailed payloads (for example, including an IdToken for who to authorize in a remote start) (Command and Control | enua) (Command and Control | enua). The station, upon receiving `RequestStartTransaction`, will begin the process and eventually issue a TransactionEvent(Started) if successful. We will cover these command changes in the next section.

Deprecated and Replaced Commands (1.6 vs 2.0.1)

With the protocol overhaul in OCPP 2.0.1, many OCPP 1.6 commands are deprecated or significantly changed. Below is a mapping of important commands from 1.6 to their 2.0.1 equivalents (or noting if they are removed). Engineers updating firmware should replace or update their handlers accordingly:

- **StartTransaction / StopTransaction (1.6) – TransactionEvent (2.0.1):** As detailed above, use `TransactionEvent` with eventType *Started* or *Ended* to signal start/stop of charging (What is new in OCPP 2.0.1). Intermediate `MeterValues` should be sent as `TransactionEvent` with eventType *Updated*. The station must generate a unique transactionId (typically a UUID or incrementing number) instead of expecting one from the server. Also, handle `seqNo` for offline order reconstruction (What is new in OCPP 2.0.1).

- **MeterValues (1.6) – (integrated into TransactionEvent or monitoring in 2.0.1):** There is no standalone `MeterValues` request initiated by the station in 2.0.1 for transaction data. All periodic or trigger-based meter readings during a transaction go inside `TransactionEvent`. If the station needs to report meter values outside of an active transaction (e.g. for an idle connector), 2.0.1 may use `NotifyReport` or `NotifyMonitoringReport` as part of the device data reporting. The CSMS can also explicitly request a one-time meter read via `GetVariables` or even a `TriggerMessage` for a MeterValues (if implemented for backward compatibility). But in general, **no separate MeterValues message** is used in normal operation (What is new in OCPP 2.0.1).

- **Authorize (1.6) – Authorize (2.0.1):** *(Name unchanged)*. The purpose remains to verify an Id token (RFID, etc.) with the CSMS. However, in 2.0.1 the `Authorize.req` message can carry more complex identifiers (the `idToken` object includes a `type` field, e.g. keycode, local, etc.). OCPP 2.0.1 `Authorize.conf` returns an `IdTokenInfo` which includes status (Accepted/Blocked/Expired/etc.), and possibly a parent IdToken or groupId, and a new field `cacheExpiryDate` for how long the token can be cached locally. Also new in 2.0.1, additional authentication methods like certificates (for Plug & Charge) mean `Authorize` might not be used in those cases (instead, certificate exchange via separate messages).

- **RemoteStartTransaction / RemoteStopTransaction (1.6) – RequestStartTransaction / RequestStopTransaction (2.0.1):** The CSMS-initiated start/stop commands have been renamed and enhanced. In 1.6, `RemoteStartTransaction.req` contained just an IdTag (and optional connector id), and `RemoteStopTransaction.req` contained a transactionId. In 2.0.1, `RequestStartTransaction.req` includes an `idToken` (with Id value and type) **and may include an EVSE id** to target a specific EVSE (Command and Control | enua). It also carries a `remoteStartId` correlation number. The station will respond with `Accepted` or `Rejected` (and in 2.0.1, `Accepted` means it will attempt to start when conditions allow, e.g. car is plugged in) (Command and Control | enua) (Command and Control | enua). When the station does start the session, it sends TransactionEvent(Started) with a triggerReason “RemoteStart”. Similarly, `RequestStopTransaction.req` just contains the transactionId to stop (Command and Control | enua) (no longer optional IdTag, since transaction is identified by station’s ID now). The station replies and if accepted, proceeds to stop the session (TransactionEvent with triggerReason “RemoteStop”). The logic on the station firmware needs to map these new requests to the internal process of starting/stopping a session.

- **ChangeConfiguration (1.6) – SetVariables (2.0.1):** In OCPP 1.6, the CSMS could send `ChangeConfiguration(key, value)` to change a configuration key (e.g. Heartbeat interval) on the charger, and the charger responded with Accepted/Rejected/NotSupported. In OCPP 2.0.1, this is replaced by the much more powerful `SetVariables` request ([PDF] Alternative Fuels Infrastructure Regulation - Alfen). `SetVariables.req` contains a list of one or more `setVariableData` entries, each specifying a *component* and *variable* name, and the new value to set. The component is part of the Device Model (for example component “ChargingStation” with variable “HeartbeatInterval”, or component “EVSE(1)” with variable “Availability” etc.). The station will respond with a

`SetVariables.conf` containing status for each variable change (Accepted, Rejected, UnknownComponent, etc.). For developers: existing config keys from 1.6 (like "HeartbeatInterval") still exist but are now scoped under components/variables. **Getting configuration** is similarly changed: 1.6's `GetConfiguration` is replaced by `GetVariables` (to read specific variables) and/or `GetReport` / `GetBaseReport` for bulk reports of configuration and inventory. **Example:** In 1.6, CSMS sends `ChangeConfiguration("HeartbeatInterval","300")`. In 2.0.1, CSMS sends `SetVariables` with component="CommunicationCtrl" (or similar) and variable="HeartbeatInterval" set to "300" – this achieves the same effect (Command and Control | enua) (Command and Control | enua).

- **GetConfiguration (1.6) – GetVariables / GetReport (2.0.1):** There is no direct analog of `GetConfiguration` that dumps all keys. Instead, OCPP 2.0.1 offers multiple ways to retrieve station info. `GetVariables.req` can specify a list of Component+Variable names to query specific configuration values. The station replies with those values (or appropriate status if not available). For a broader overview, `GetBaseReport.req` can be used with a scope (like "FullInventory", "ConfigurationInventory", etc.) and the station will respond by sending one or more `NotifyReport` messages containing a comprehensive list of components and variables or settings (Command and Control | enua) (Command and Control | enua). Essentially, *GetVariables for pinpoint reads, GetReport for structured dumps*. The old `ConfigurationKey/ConfigurationValue` concept is now subsumed by the Device Model's components and variables.

- **StatusNotification (1.6) – StatusNotification (2.0.1):** The message name is the same, but the usage is narrower in 2.0.1. In 1.6, stations sent StatusNotification frequently for every status change (Available, Preparing, Charging, SuspendedEV, Finishing, etc.) for each connector. In 2.0.1, **transaction-related status changes are handled by TransactionEvents**, so `StatusNotification` is used for **connector or EVSE availability changes outside of transactions** (What is new in OCPP 2.0.1). The status values in 2.0.1 are slightly different: for example, "Available" and "Occupied" are used (and "Preparing/Finishing" might be deprecated or have different semantics). Also, there are status for an entire EVSE vs an individual connector if needed. As a developer, you'll still implement StatusNotification sending (for cases like a connector is plugged in but not yet in transaction = maybe status "Occupied" with no transaction, or a fault occurs), but you won't send "Charging" status – the CSMS infers that from a TransactionEvent(Started). This simplifies state management.

- **DataTransfer (1.6) – DataTransfer (2.0.1):** *(Same name and concept)*. DataTransfer is the vendor-specific escape hatch for any custom or proprietary messages. It still exists in 2.0.1 with the same structure: a vendorId and an opaque data payload. If you used custom extensions in 1.6 via DataTransfer, those can be carried over. The station should respond with DataTransfer.conf indicating success or unknown message.

- **Firmware Update & Diagnostics:**
  - **UpdateFirmware (1.6) – UpdateFirmware (2.0.1)**: The command to initiate a firmware update from CSMS to station remains `UpdateFirmware.req` (with a URL, retrieveDate, etc.), but 2.0.1 expects the station to support HTTPS and possibly include a **firmware signature** (there are fields for signing in 2.0.1). The station's response is immediate (Accepted/Rejected), but then station must download the firmware and send `FirmwareStatusNotification` periodically to report progress (e.g. Downloading, Downloaded, Installing, Installed, or error states). In 1.6, firmware status was reported via `FirmwareStatusNotification` too, but 2.0.1 defines the states more rigorously and ties into security (it should verify signature). So, while the message name is the same, ensure to implement signature verification and use the extended status codes.
  - **GetDiagnostics (1.6) – GetLog (2.0.1)**: In 1.6, the CSMS could request diagnostic logs via `GetDiagnostics.req` (with an URL to upload to, etc.), and the station would upload its log and then send `DiagnosticsStatusNotification`. In 2.0.1, this was generalized. The CSMS uses `GetLog.req` specifying log parameters (which log type – e.g. "DiagnosticsLog" or "SecurityLog", a URL, optional filters like time range). The station responds, and then uploads the requested log. Upon completion or failure, the station sends

`LogStatusNotification` (with status: Uploaded, UploadFailed, AccessDenied, etc.). So, implement `GetLog` handling and use `LogStatusNotification` instead of DiagnosticsStatusNotification ([PDF] OCPP 2.0.1: Part 2 - Errata - Regulations.gov). Also, note there is a separate `GetMonitoringReport` that can request the current values of all active monitors (related to device management).

- **ReserveNow / CancelReservation (1.6) – ReserveNow / CancelReservation (2.0.1):** These features remain with the same names and function. The CSMS can send `ReserveNow` to ask a station to reserve a specific EVSE/connector for a user (IdToken or e.g. e-mail id, etc.) for a time window. In 2.0.1, the payload includes an `idToken` object and optionally an `evseId` instead of `connectorId` (since reservations are at EVSE level). The station will respond and later send `ReservationStatusUpdate` to inform if the reservation started or expired. `CancelReservation` also remains for the CSMS to cancel a pending reservation by reservationId.

- **TriggerMessage (1.6) – TriggerMessage (2.0.1):** Still available and similar usage. The CSMS sends `TriggerMessage.req` with a requested message type (like “BootNotification” or “StatusNotification”) and possibly an EVSE id. The station should, if accepted, send the requested notification immediately. In 2.0.1, TriggerMessage includes new trigger options (it can request messages like `FirmwareStatusNotification` or `Heartbeat` as well). Implement it as a way to force an immediate update from the station.

- **Heartbeat (1.6) – Heartbeat (2.0.1):** Same concept: station-initiated periodic ping to indicate it’s alive. If using persistent WebSocket, Heartbeat mainly serves as a keep-alive and to let CSMS know the station clock. In 2.0.1, Heartbeat is optional if WebSocket pings are used, but typically still implemented. The interval can be configured via the `HeartbeatInterval` variable (set by SetVariables).

- **UnlockConnector (1.6) – UnlockConnector (2.0.1):** Same message name and function: CSMS requests the station to mechanically unlock a connector latch (usually in case a cable is stuck). The only difference is that in 1.6, you specify a `connectorId`, whereas in 2.0.1 you specify an `evseId` and connectorId (if multiple connectors on an EVSE). Many stations have one connector per EVSE, so evseId and connectorId might be the same or the connectorId may be omitted if the EVSE has only one connector. The response status enums are similar (Unlocked, UnlockFailed, OngoingTransaction). Code that handled UnlockConnector in 1.6 will mostly carry over, just adapt to new id semantics.

- **ChangeAvailability (1.6) – ChangeAvailability (2.0.1):** Functionally similar: the CSMS instructs the station to change an EVSE or the whole station to Operative (available for use) or Inoperative (not available for new sessions). In 1.6, `ChangeAvailability.req` had a `connectorId` (0 for entire station or a specific connector) and a `type` (“Inoperative”/“Operative”). In 2.0.1, it uses `evseId` (0 for entire station, or specific EVSE) and `operationalStatus` (“Inoperative”/“Operative”) (Command and Control | enua) (Command and Control | enua). The station will either immediately or scheduled (after current transaction) change availability and reply accordingly (`Accepted/Scheduled/Rejected`). The logic to implement is the same, but now you might mark an entire EVSE (which could affect multiple connectors).

- **ClearChargingProfile / SetChargingProfile (1.6) – ClearChargingProfile / SetChargingProfile (2.0.1):** These messages still exist and have been extended. In 1.6 they allowed the CSMS to set a charging limit profile (for a connector or overall station) and clear profiles. In 2.0.1, because of the EVSE model and additional features like TxDefault profiles for specific users or vehicle constraints, the structure is richer. `SetChargingProfile.req` in 2.0.1 includes an `evseId` (0 meaning station level) and a `chargingProfile` composed of charging schedule periods, an optional limit in cost, etc., similar to 1.6 but possibly with new attributes (like the concepts of charging profile purposes: TxProfile, TxDefaultProfile, etc., remain but might be renamed slightly). `ClearChargingProfile` can target by charging profile ID, by EVSE, or other criteria. If you had these implemented, you’ll adapt to slightly different JSON structure but concept remains to regulate power or current draw.

- **GetCompositeSchedule (1.6) – GetCompositeSchedule (2.0.1):** Unchanged in name and purpose: the CSMS asks the station for a composite charging schedule when multiple profiles are active (like to know how much power the station expects to draw over time). OCPP 2.0.1 still supports this (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). Implementations might simply carry over, possibly adjusting to EVSE id usage. The station returns a schedule that aggregates its limitations and active profiles.
- **ClearCache (1.6) – ClearCache (2.0.1):** Same message, same function: instruct station to clear its local authorization cache (if caching IdTags). If your station caches authorized IdTokens, implement this as before.
- **FirmwareStatusNotification (1.6) – FirmwareStatusNotification (2.0.1):** The station-initiated message to report firmware update progress still exists. The set of status values is expanded in 2.0.1 (e.g., includes "Authenticating" if the station is verifying a firmware package signature, etc.). Ensure to send these statuses during update process.
- **DiagnosticsStatusNotification (1.6) – (Removed in 2.0.1):** The concept is replaced by `LogStatusNotification` after a `GetLog`. So no separate DiagnosticsStatus in 2.0.1 core.
- **GetLocalListVersion / SendLocalList (1.6) – GetLocalListVersion / SendLocalList (2.0.1):** These messages for managing a local authorization whitelist remain largely the same. CSMS can ask the station which version of the local list it has, and send incremental or full updates. The station should support local list if needed (often not used if always online).
- **Security Extensions:** Some functions that were not in core 1.6 now have official messages in 2.0.1:
  - Certificate handling: `InstallCertificate`, `DeleteCertificate`, `GetInstalledCertificateIds` are new. (In 1.6 there was no standardized way; some vendors had custom DataTransfer for this.)
  - `SignCertificate`: station asks CSMS to sign its CSR (this was only via a custom in 1.6 if at all; now standardized).
  - `CertificateSigned`: CSMS's reply with the signed certificate.
  - Use of these is required for security profile 3 (mutual TLS).
  - There's also `GetCertificateStatus` (station can check if a cert signing request was accepted if it didn't get a quick response).
  - `SecurityEventNotification`: station to CSMS for security events (no 1.6 equivalent).
- **New ISO 15118 related:** In 2.0.1, for Plug & Charge and smart EV communication, we have:
  - `Get15118EVCertificate` (station requests a contract certificate from CSMS for the EV, part of Plug&Charge).
  - `PublishFirmware` (CSMS instructs a station or group of stations to download new firmware from a location and be ready, possibly used for V2G certificates as well).
  - `NotifyEVChargingNeeds` (station informs CSMS of EV's energy needs, obtained via ISO 15118 from the car).
  - `NotifyEVChargingSchedule` (station sends the schedule the EV agreed on or needs, for coordination). These had no equivalent in 1.6 (except via custom extensions).
- **Miscellaneous Renamed or New Commands:**
  - `SetDisplayMessage` / `ClearDisplayMessage` / `GetDisplayMessages` / `NotifyDisplayMessages` – new in 2.0.1 for managing on-screen text. Not in 1.6.
  - `CostUpdated` – new message where CSMS can send an updated cost per kWh or tariff info during a transaction (if price changes), and station can display it. No 1.6 equivalent.

- `CustomerInformation` / `NotifyCustomerInformation` – new in 2.0.1, allows CSMS to request certain info or station to push certain info related to a customer (e.g., to support new use cases like showing messages about the customer's account or retrieving audit data).
- `ReserveNow` / `CancelReservation` – same names, different payload as mentioned.
- `Reset` – still exists; in 2.0.1 you can specify *which* reset (soft or hard) and possibly a specific subsystem.
- `ReportChargingProfiles` – new, station -> CSMS, used to report all active charging profiles (in response to `GetChargingProfiles`).
- `GetTransactionStatus` – new, CSMS -> station, to inquire if a given EVSE has an ongoing transaction or if a particular IdToken is being used. Useful to check before remote start or for CSMS sync.
- `SetMonitoringBase`, `SetMonitoringLevel`, `SetVariableMonitoring` – new messages for the CSMS to configure the station's monitoring features (thresholds, what events to report). No direct 1.6 analog (monitoring is new in 2.0.1's device model). These allow dynamic setup of the station's event triggers.

Tip: When migrating, review the official OCPP 2.0.1 documentation or the OCA's *"What is new in OCPP 2.0.1"* guide to ensure all message pairs you used in 1.6 are correctly replaced (What is new in OCPP 2.0.1) ([PDF] Alternative Fuels Infrastructure Regulation - Alfen). Not every 1.6 message is listed above, but the above covers the common and important ones. Many messages carry over with minor name changes or payload differences (e.g. types like IdToken vs IdTag). Below we'll demonstrate some of these with Java code.

Implementing OCPP 2.0.1 Features: Java Examples

For developers implementing OCPP 2.0.1 on an EV charger (charge point), it's helpful to see how certain tasks are handled in code. The examples below use a generic approach (pseudo-code using typical OCPP library structures) to illustrate handling of key new commands on the charger side. We assume a Java OCPP library that provides request/response classes for OCPP 2.0.1 messages (e.g., the open-source Java-OCA-OCPP library).

Example 1: Sending a TransactionEvent (Started/Updated/Ended)

In OCPP 2.0.1, the charger must send `TransactionEventRequest` messages to report transaction state. The code below shows how one might construct and send these events in a Java application when a session starts and ends. We use an imaginary `OcppClient` API to send the request.

```
// When a new charging session is about to start (e.g., after authorization and plug in):
TransactionEventRequest startEvent = new TransactionEventRequest();
startEvent.setEventType(TransactionEventRequest.EventTypeEnum.Started);
startEvent.setTimestamp(Instant.now());                          // current time
startEvent.setTriggerReason(TransactionEventRequest.TriggerReasonEnum.Authorized);
startEvent.setSeqNo(1);                                          // sequence number 1 for
start
// Set the identification token (if an RFID/card was used)
IdToken idToken = new IdToken();
idToken.setIdToken("ABC1234567");
idToken.setType(IdTokenTypeEnum.ISO14443);  // e.g., ISO14443 for RFID card
startEvent.setIdToken(idToken);
// Specify the EVSE and connector involved
startEvent.setEvse(new EVSE(1, 1)); // EVSE id 1, connector id 1
// Transaction info: we generate a transaction ID now
TransactionInfo txnInfo = new TransactionInfo();
txnInfo.setTransactionId(UUID.randomUUID().toString());           // unique transaction ID
txnInfo.setChargingState(ChargingStateEnum.Charging);             // initial state
txnInfo.setTimeSpentCharging(0);                                 // just started
startEvent.setTransactionInfo(txnInfo);
// (Optional) Include meter value at start
MeterValue startMeter = new MeterValue();
startMeter.setTimestamp(Instant.now());
SampledValue sampledVal = new SampledValue("0", ReadingContextEnum.Begin,
UnitOfMeasureEnum.Wh);
startMeter.setSampledValue(Collections.singletonList(sampledVal));
startEvent.setMeterValue(Collections.singletonList(startMeter));

// Send the TransactionEvent.Started request to CSMS
ocppClient.sendRequest(startEvent, response -> {
    TransactionEventResponse startResp = (TransactionEventResponse) response;
    System.out.println("Transaction start acknowledged, status: " +
startResp.getStatus());
});
```

Later, during the charging session, suppose we want to send an update (periodic meter value):

```
// Periodically (e.g., every minute) during the session:
TransactionEventRequest updateEvent = new TransactionEventRequest();
updateEvent.setEventType(TransactionEventRequest.EventTypeEnum.Updated);
updateEvent.setTimestamp(Instant.now());
updateEvent.setTriggerReason(TransactionEventRequest.TriggerReasonEnum.MeterValuePeriodic)
;
updateEvent.setSeqNo(nextSequenceNumber()); // increment sequence
updateEvent.setEvse(new EVSE(1, 1));
updateEvent.setTransactionInfo(txnInfo);    // reuse the same TransactionInfo (with same
transactionId)
txnInfo.setChargingState(ChargingStateEnum.Charging); // still charging (could also be
SuspendedEV if applicable)
// Include current meter value
MeterValue currentMeter = new MeterValue();
currentMeter.setTimestamp(Instant.now());
SampledValue energyVal = new SampledValue(currentEnergyWh + "",
ReadingContextEnum.SamplePeriodic, UnitOfMeasureEnum.Wh);
currentMeter.setSampledValue(Collections.singletonList(energyVal));
updateEvent.setMeterValue(Collections.singletonList(currentMeter));
// Send update
ocppClient.sendRequest(updateEvent, response -> {
    TransactionEventResponse updateResp = (TransactionEventResponse) response;
    // typically no significant payload in response, just acknowledgment
});
```

And when the transaction ends (EV disconnected or stopped):

```
TransactionEventRequest stopEvent = new TransactionEventRequest();
stopEvent.setEventType(TransactionEventRequest.EventTypeEnum.Ended);
stopEvent.setTimestamp(Instant.now());
stopEvent.setTriggerReason(TransactionEventRequest.TriggerReasonEnum.EVDisconnected);
stopEvent.setSeqNo(nextSequenceNumber());
stopEvent.setEvse(new EVSE(1, 1));
stopEvent.setTransactionInfo(txnInfo);
txnInfo.setChargingState(ChargingStateEnum.EVDisconnected);
txnInfo.setStoppedReason(StoppedReasonEnum.EVDisconnected); // reason for stopping
// Final meter reading
MeterValue endMeter = new MeterValue();
endMeter.setTimestamp(Instant.now());
SampledValue endVal = new SampledValue(finalEnergyWh + "", ReadingContextEnum.End,
UnitOfMeasureEnum.Wh);
endMeter.setSampledValue(Collections.singletonList(endVal));
stopEvent.setMeterValue(Collections.singletonList(endMeter));
// Optionally, add transactionData like overall cost, if station calculated it
stopEvent.setTotalCost(new BigDecimal("5.00")); // example: send total cost of session

ocppClient.sendRequest(stopEvent, response -> {
    TransactionEventResponse stopResp = (TransactionEventResponse) response;
    System.out.println("Transaction ended, CSMS response: " + stopResp.getStatus());
    // Now transaction is complete
});
```

In these snippets, we see how OCPP 2.0.1 encodes the information. We create one `TransactionEventRequest` object per event. The library classes (e.g., `IdToken`, `EVSE`, `MeterValue`, etc.) correspond to OCPP 2.0.1 data structures. The CSMS's response (`TransactionEventResponse`) typically just has a status (Accepted/Rejected or OK) and possibly some info like updated `timeouts` or certificates if provided in future versions. The station should check the response for any relevant info (usually nothing for TransactionEvent, it's mostly an acknowledgment).

Example 2: Changing Configuration with SetVariables

In OCPP 2.0.1, configuration changes from the CSMS come via `SetVariables`. Let's simulate handling a request on the charger side, and also how the CSMS might send one. Suppose the CSMS wants to change the charger's **StopTransactionOnEVSideDisconnect** setting to true (so that if the EV unplugs, the station automatically ends the transaction), and the **HeartbeatInterval** to 60 seconds.

On the CSMS side (sending the request):

```
SetVariablesRequest setReq = new SetVariablesRequest();

// Prepare first variable entry: StopTxOnEVSideDisconnect = true
SetVariableData var1 = new SetVariableData();
Component component1 = new Component();
component1.setName("TxCtrlr");                      // e.g., Transaction Controller
component
var1.setComponent(component1);
Variable variable1 = new Variable();
variable1.setName("StopTxOnEVSideDisconnect");
var1.setVariable(variable1);
var1.setAttributeValue("true");
var1.setAttributeType(AttributeEnum.Target);        // 'Target' value to set

// Second variable entry: HeartbeatInterval = 60
SetVariableData var2 = new SetVariableData();
Component component2 = new Component();
component2.setName("OCPPCommCtrlr");                // e.g., OCPP Communication Controller
component
var2.setComponent(component2);
Variable variable2 = new Variable();
variable2.setName("HeartbeatInterval");
var2.setVariable(variable2);
var2.setAttributeValue("60");
var2.setAttributeType(AttributeEnum.Target);

setReq.setSetVariableData(Arrays.asList(var1, var2));

// Send to the charger
ocppClient.sendRequest(setReq, response -> {
    SetVariablesResponse setConf = (SetVariablesResponse) response;
    for (SetVariableResult result : setConf.getSetVariableResult()) {
        System.out.println("Set " + result.getComponent().getName() + "/"
            + result.getVariable().getName() + ": " + result.getAttributeStatus());
    }
});
```

On the charger side (handling the request):

If using a library, you would implement a handler for `SetVariablesRequest`. For example:

```java
public class ChargingStationRequestHandler implements RequestHandler {

    @OcppRequestHandler
    public SetVariablesResponse handleSetVariables(SetVariablesRequest request) {
        List<SetVariableResult> results = new ArrayList<>();
        for (SetVariableData entry : request.getSetVariableData()) {
            String comp = entry.getComponent().getName();
            String var = entry.getVariable().getName();
            String val = entry.getAttributeValue();
            // Attempt to apply the configuration change
            SetVariableResult result = new SetVariableResult();
            result.setComponent(entry.getComponent());
            result.setVariable(entry.getVariable());
            // Check if this component/variable is supported
            if (!configStore.supports(comp, var)) {
                result.setAttributeStatus(SetVariableStatusEnum.UnknownVariable);
            } else if (!configStore.isWritable(comp, var)) {
                result.setAttributeStatus(SetVariableStatusEnum.Rejected); // not writable
            } else {
                boolean applied = configStore.setValue(comp, var, val);
                result.setAttributeStatus(applied ? SetVariableStatusEnum.Accepted
                                                  : SetVariableStatusEnum.Rejected);
                // If applied, also change runtime behavior if needed
                if (applied) {
                    applyConfigChangeToSystem(comp, var, val);
                }
            }
            results.add(result);
        }
        SetVariablesResponse response = new SetVariablesResponse();
        response.setSetVariableResult(results);
        return response;
    }
}
```

In this pseudo-code, `configStore` is a hypothetical configuration registry for the station. We iterate through each requested variable change, check support and writability, apply it, and build a result list. The response sends back a status (`Accepted` , `Rejected` , `UnknownComponent` , `NotSupportedAttribute` , etc. as defined by OCPP) for each entry.

The mapping of component names like `"TxCtrlr"` or `"OCPPCommCtrlr"` comes from the OCPP 2.0.1 specification's Device Model. These names and the variables under them are standardized (for example, StopTxOnEVSideDisconnect is typically under component "TxCtrlr").

Developer Note: It's crucial to consult the OCPP 2.0.1 documentation for the exact component and variable names. The Open Charge Alliance provides a list of standard components/variables. In your code, you might use enums or constants for them instead of raw strings.

Example 3: Certificate Management (Installing a CA Certificate)

OCPP 2.0.1 allows remote installation of certificates, e.g. a new trusted CA for client-side TLS or a V2G certificate for Plug & Charge. Let's illustrate how a station might handle an `InstallCertificate` request from the CSMS to install a new charging station certificate (or a CA certificate).

CSMS side (sending a certificate):

```
InstallCertificateRequest installReq = new InstallCertificateRequest();
installReq.setCertificateType(CertificateUseEnum.CSMSRootCertificate);
// e.g., instruct station to install this as a trusted CSMS root CA, could also be
ChargingStationCertificate etc.
String certPem = "-----BEGIN CERTIFICATE-----\nMIIFujCC...==\n-----END CERTIFICATE-----";
installReq.setCertificate(certPem);

ocppClient.sendRequest(installReq, response -> {
    InstallCertificateResponse installConf = (InstallCertificateResponse) response;
    System.out.println("Certificate install result: " + installConf.getStatus());
});
```

Charger side (handling InstallCertificate):

```
@OcppRequestHandler
public InstallCertificateResponse handleInstallCertificate(InstallCertificateRequest
request) {
    InstallCertificateResponse resp = new InstallCertificateResponse();
    CertificateUseEnum certType = request.getCertificateType();
    String certPem = request.getCertificate(); // PEM format certificate string
    boolean success = false;
    try {
        switch(certType) {
            case CSMSRootCertificate:
                success = certificateStore.installCaCertificate(certPem,
CertificateStore.CA_CSMS);
                break;
            case ManufacturerRootCertificate:
                success = certificateStore.installCaCertificate(certPem,
CertificateStore.CA_MANUFACTURER);
                break;
            case ChargingStationCertificate:
                success = certificateStore.installLocalCertificate(certPem,
CertificateStore.CERT_CHARGING_STATION);
                break;
            // ... other types: V2G root cert, etc.
        }
        resp.setStatus(success ? InstallCertificateStatusEnum.Accepted
                               : InstallCertificateStatusEnum.Failed);
    } catch (SecurityException ex) {
        resp.setStatus(InstallCertificateStatusEnum.Rejected);
    }
    return resp;
}
```

In the above, `certificateStore` is a hypothetical utility managing the station's certificate storage (could interface with a secure element or filesystem). The station stores the provided certificate in the appropriate slot (depending on type, e.g., trust store vs personal certificate). It then returns `Accepted` or an error status.

For a full certificate signing flow: the station might send `SignCertificate.req` (with a CSR) to the CSMS, which responds with `CertificateSigned.conf` containing the signed certificate. Handling that would be similar: parse the cert chain and install it.

Example 4: Firmware Update Process

To illustrate best practices, here's a simplified view of how a station might handle an `UpdateFirmware` command with secure download and status notifications:

```
@OcppRequestHandler
public UpdateFirmwareResponse handleUpdateFirmware(UpdateFirmwareRequest request) {
    // Parse request
    String firmwareUrl = request.getLocation();
    Instant retrieveDate = request.getRetrieveDate(); // when to start download
    String firmwareHash = request.getRetrieveDate();  // optional checksum provided
    // Schedule the download if retrieveDate is future
    if (retrieveDate != null && retrieveDate.isAfter(Instant.now())) {
        scheduleDownload(retrieveDate, firmwareUrl, firmwareHash);
    } else {
        startFirmwareDownload(firmwareUrl, firmwareHash);
    }
    UpdateFirmwareResponse resp = new UpdateFirmwareResponse();
    resp.setStatus(UpdateFirmwareStatusEnum.Accepted);
    return resp;
}

private void startFirmwareDownload(String url, String expectedHash) {
    // Notify CSMS that download is starting
    sendFirmwareStatusNotification(FirmwareStatusEnum.Downloading);
    try {
        byte[] firmwareData = downloadFile(url);  // do an HTTPS download
        // Calculate and verify hash if provided
        if (expectedHash != null && !verifyHash(firmwareData, expectedHash)) {
            sendFirmwareStatusNotification(FirmwareStatusEnum.DownloadFailed);
            return;
        }
        // Save firmware to storage for installation
        saveFirmwareFile(firmwareData);
        sendFirmwareStatusNotification(FirmwareStatusEnum.Downloaded);
        // Start installation
        boolean installed = installFirmware(firmwareData);
        if (installed) {
            sendFirmwareStatusNotification(FirmwareStatusEnum.Installing);
            // ... upon success:
            sendFirmwareStatusNotification(FirmwareStatusEnum.Installed);
            rebootToNewFirmware();
        } else {
            sendFirmwareStatusNotification(FirmwareStatusEnum.InstallationFailed);
        }
    } catch (IOException e) {
        sendFirmwareStatusNotification(FirmwareStatusEnum.DownloadFailed);
    }
}
```

Here we outline the station receiving `UpdateFirmwareRequest` (with URL and possibly a starting time and a checksum). The station should schedule or immediately start the download. It then uses `FirmwareStatusNotification` to inform the CSMS of progress: *Downloading*, *Downloaded*, *Installing*, *Installed*, or errors. The download is done over HTTPS for security. We verify the hash to ensure integrity (and potentially a signature in real implementations). After installation, the station might restart. The CSMS, seeing the sequence of notifications, knows the outcome.

Note: Always use secure transport (HTTPS) and verify file signatures (OCPP can provide a **signed firmware certificate** out-of-band or as part of the message, depending on implementation). The above is simplified for clarity.

Best Practices for OCPP 2.0.1 Implementation

Upgrading to OCPP 2.0.1 is not just a message update; it involves embracing new paradigms for security and management. Here are some best practices and recommendations:

- **Use the Device Model fully:** Take advantage of OCPP 2.0.1's device model to organize your charger's internal data. Implement the inventory reporting (`BootNotification` and `NotifyReport` with full component list) so the CSMS knows your charger's capabilities (number of EVSEs, connectors, supported features). Configure your firmware to monitor important variables (temperatures, voltage levels, contactor states) using the **Monitoring** features. For example, support `SetVariableMonitoring` so the CSMS can ask your station to send a `NotifyEvent` if, say, the EVSE temperature exceeds a threshold. This helps preempt failures and is a big plus for operating large networks (What is new in OCPP 2.0.1) (What is new in OCPP 2.0.1). **Tip:** Map your hardware sensors and settings to OCPP's standard components/variables whenever possible.

- **Prioritize Security (Profile 3):** It is highly recommended to run OCPP 2.0.1 in its most secure mode. This means:

  - **TLS with mutual authentication:** Install a unique client certificate on each station (provisioned securely) and configure the CSMS's CA. The handshake ensures only trusted stations connect. Leverage the certificate management messages to update/renew these certificates before expiry (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison).
  - **Secure firmware updates:** Always sign firmware images and have the station verify the signature. The OCPP firmware update procedure allows including a **signing certificate** or a **signature** for the image. Use those fields and verify in station software. This prevents malicious firmware from being installed (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison).
  - **Maintain a Security Log:** Record security-relevant events (e.g., failed login attempts, invalid certificates presented, firmware verification failures) in a log on the station. Support the `GetLog` request for the security log and `SecurityEventNotification` to alert critical issues in real-time (What is new in OCPP 2.0.1). This is important for compliance in some regions and for general cybersecurity.
  - **Harden network interface:** Only allow OCPP traffic (e.g., restrict other services if not needed), and use the JSON/WebSocket transport (SOAP is not supported in 2.0.1 anyway). Ensure proper validation of messages since OCPP 2.0.1 is a large surface area – use robust JSON parsing and check enum bounds etc., to avoid injection attacks.

- **Testing and Certification:** OCPP 2.0.1 has an official Certification program via the Open Charge Alliance. Aim to certify your implementation – this ensures interoperability. Use the OCPP 2.0.1 Compliance Testing Tool and attend plugfests if possible. Test extensively the transaction sequences (including offline scenarios), as well as all optional profiles you choose to support (like ISO 15118, smart charging, display messages). Catching issues early will save headaches given the increased complexity.

- **Efficient Transaction Handling:** With the switch to TransactionEvent, pay attention to:

  - The **state management** on the station. You might maintain a state machine per EVSE that tracks whether a session is active. Use that to decide when to send StatusNotifications (only for transitions like to Faulted or back to Available) vs. TransactionEvents (for actual session data).
  - **Sequence numbering:** Always increment `seqNo` for TransactionEvents and reset it when a new transaction starts at an EVSE. If the station goes offline, buffer TransactionEvents and send them when reconnected; the CSMS will use seqNo to order them (What is new in OCPP 2.0.1).
  - **Configurable start/stop:** As mentioned, if your station wants to start transactions at EV plug-in (before authorization), make sure to set the appropriate variables (`ConnectionTimeOut` , etc., and the start/stop points configuration). If a transaction can start without an IdToken (plug-and-charge scenarios), you may send a TransactionEvent.Started with no idToken but with triggerReason "PowerPathClosed" or similar, then perhaps followed by an idToken in a later Updated event once the car authenticates via certificate.

- **Robustness in Remote Commands:** OCPP 2.0.1 allows more remote control. Implement **graceful handling** for things like:
  - `RequestStartTransaction` when the EV isn't plugged yet – station should reply Accepted and then wait until plug-in to actually start (with a reasonable timeout).
  - `RequestStopTransaction` – handle even if the transactionId is already stopping or gone (send the appropriate error if not valid, or accept and ignore if already stopped).
  - `Reset` – if a Reset is requested while charging, decide if you can **soft reset** subsystems without dropping the session, or send `Reject` / `Scheduled` appropriately. OCPP 2.0.1 allows a "Scheduled" response meaning reset will happen after current transaction (Command and Control | enua).
  - `ChangeAvailability` – if asked to go Inoperative immediately but a car is charging, typically respond `Scheduled` (will go Inoperative after session done) (Command and Control | enua).

- **Concurrent Transactions and EVSEs:** If your hardware has multiple EVSEs (like a charging station with two ports that can charge two cars simultaneously), ensure your implementation supports **parallel transactions** cleanly. Each EVSE has its own transaction messages. Use separate identifiers or threads for each EVSE's state to avoid mixing up events. This was possible in 1.6 (multiple connectors), but 2.0.1's EVSE model formalizes it – take advantage by structuring code to handle each EVSE independently under one station process.

- **Logging and Debugging:** OCPP 2.0.1 is complex, so implement extensive logging on the station for development and debugging. Log all messages sent/received (at least in debug builds) – this will help track the sequence of events and debug issues with the backend. Because many events are asynchronous, logging helps to trace, for example, why a `RequestStartTransaction` didn't result in a TransactionEvent if something failed in between.

- **Maintain backward compatibility (if needed):** Some networks might still run OCPP 1.6. If you want your device to support both (e.g., switchable via configuration), isolate the protocol-handling portion so you could run a 1.6 mode. However, due to fundamental differences, this can be non-trivial. Many choose to run separate firmware for 1.6 vs 2.0.1. If upgrading fully to 2.0.1, clearly communicate to clients that it requires a 2.0.1 compatible CSMS.

- **Leverage New Profiles Judiciously:** OCPP 2.0.1 has many optional features (Display, ISO15118, etc.). It's not required (or sometimes feasible) to implement everything. Choose the profiles that make sense for your product:
  - If your charger has a display, implement the *Display Messages* profile.
  - If it will support Plug & Charge, implement the *ISO 15118 Certificate Management* profile.
  - If your clients need rich transaction notifications, implement *Customer Information* profile to allow sending driver messages.

- Each profile is defined in the spec; you should handle at least all Core messages and ideally Smart Charging and Security as those are widely needed.

By following these practices, you'll create a robust and secure OCPP 2.0.1 charging station implementation that can handle the demands of modern EV charging networks.

Glossary of OCPP 2.0.1 Commands

Below is a glossary of all primary OCPP 2.0.1 message types (commands), organized by category. Each entry includes the message name and a brief description of its purpose. This serves as a reference for developers to quickly recall what each message does in OCPP 2.0.1.

Core Operations

- **BootNotification** *(Charge Point -> CSMS)*: Initial handshake from a Charging Station to the CSMS when it boots or reboots. Contains station identity, model, firmware version, etc. CSMS responds with acceptance and heartbeat interval. Establishes the station as online (OCPP 2.0.1 vs OCPP 1.6 — What's the Difference and What's Coming Next? | by mikebolshakov | Apr, 2025 | Medium).
- **Heartbeat** *(Charge Point -> CSMS)*: Periodic ping from the station to indicate it's still connected and to get the current time from CSMS. CSMS responds with current timestamp. Used as a keep-alive (interval configurable by HeartbeatInterval).
- **Authorize** *(Charge Point -> CSMS)*: Sent by station to ask CSMS to validate an IdToken (RFID card, etc.) for a charging session. CSMS responds with IdTokenInfo (status of IdToken and authorization status). In 2.0.1, supports multiple token types (e.g., "Central", "Local", "ISO14443" RFID, "KeyCode") and can carry a certificate for Plug & Charge.
- **TransactionEvent** *(Charge Point -> CSMS)*: **Key new message** that reports transaction state changes. Used for starting, updating, and stopping charging sessions. Replaces StartTransaction/StopTransaction/MeterValues (What is new in OCPP 2.0.1). Payload includes transactionId, timestamp, eventType (Started, Updated, Ended), trigger reason, meter values, and more. CSMS responds with basic status (and may echo some info).
- **StatusNotification** *(Charge Point -> CSMS)*: Station informs CSMS of a change in availability or status of a connector/EVSE. In 2.0.1, primarily used for non-transaction events (e.g., a fault occurs, or a connector becomes available/unavailable) (What is new in OCPP 2.0.1). Contains EVSE/connector id, current status (Available, Occupied, Faulted, etc.), and error code if any. CSMS uses this to update station status on its side.
- **DataTransfer** *(Either direction)*: For proprietary extensions. Contains a vendorId and any custom data. Used to implement non-standard features if needed (both station and CSMS must understand the content). Unchanged from 1.6.
- **MeterValues** *(CSMS -> Charge Point)*: *Rare in 2.0.1.* This is now an optional message where the CSMS requests a one-time meter value reading from the station (for a given EVSE). Not typically used because TransactionEvent covers periodic data. (In OCPP 1.6, MeterValues was Charge Point -> CSMS; in 2.0.1 it's redefined as a CSMS request to trigger a read.)
- **TransactionEventResponse** *(CSMS -> Charge Point)*: Response to `TransactionEvent`. Indicates if event was accepted. Can also convey things like updated `chargingPriority` or other operational messages (though usually just an OK).
- **Response messages (General):** In OCPP, every request has a corresponding response (e.g. `BootNotificationResponse`, `AuthorizeResponse`, etc.), typically echoing status of processing and sometimes additional info. In this glossary, we focus on requests since responses are usually straightforward.

Remote Control & Administration (CSMS -> Charge Point)

- **RequestStartTransaction** *(CSMS -> Charge Point)*: Instructs the station to start a transaction for a given IdToken (and possibly EVSE). Contains IdToken, optional EVSE id, and a remoteStartId. Station will (if possible) start a session (authorize internally using the token or pre-authorized info) (Command and Control | enua). Response is Accepted or Rejected (Command and Control | enua). If Accepted, a TransactionEvent(Started) should follow when conditions are met.
- **RequestStopTransaction** *(CSMS -> Charge Point)*: Requests an ongoing transaction (identified by transactionId) to stop. Station responds Accepted/Rejected (Command and Control | enua). If accepted, it should immediately (or as soon as safely possible) terminate the session and send TransactionEvent(Ended).
- **ReserveNow** *(CSMS -> Charge Point)*: Reserve an EVSE for future use by a specific IdToken (or e.g. an EV license via e.g. IdToken with type "MacAddress" for Autocharge). The request includes reservationId, IdToken, evseId, and reservation expiry time. Station, if able, will lock that EVSE for that user (no one else can start a session on it until either reservation times out or the reserved user starts a session). Responds with Accepted/Rejected/Faulted. Station will later send `ReservationStatusUpdate` to inform the final outcome (e.g. reservation started when the user arrived, or ended).
- **CancelReservation** *(CSMS -> Charge Point)*: Cancel an existing reservation (by reservationId). Station responds if it canceled it (Accepted) or if it was not found or already started (Rejected). If a reservation is cancelled, the station can free up the EVSE immediately.
- **TriggerMessage** *(CSMS -> Charge Point)*: Asks the station to immediately send a particular message. The request specifies a `requestedMessage` (enum of possible message types, e.g. BootNotification, StatusNotification, FirmwareStatusNotification, Heartbeat, etc.) and optionally an evseId if the message is specific to an EVSE. Station responds Accepted if it will trigger the message, then sends the requested message right after. This is used for on-demand updates instead of waiting for next scheduled report ([PDF] Improving Uptime Monitoring with OCPP - Open Charge Alliance).
- **ChangeAvailability** *(CSMS -> Charge Point)*: Change operational status of an EVSE or the whole station (Command and Control | enua) (Command and Control | enua). It has an `operationalStatus` = Operative or Inoperative, and an evseId (0 meaning all). *Operative* means the EVSE can carry out charging transactions. *Inoperative* means it should refuse new transactions (or even stop current if instructed). Station responds Accepted if it will comply, Scheduled if it will defer until current session ends, or Rejected if it cannot comply. The station should also reflect the new availability via StatusNotifications or component status (e.g., evse component variable "AvailabilityState").
- **Reset** *(CSMS -> Charge Point)*: Remotely reboot the charging station or a specific subsystem. The request can specify a type: *Soft* or *Hard*. Some implementations allow an evseId or section (though in core, it's station-wide). Station responds Accepted (will reset immediately), Scheduled (will reset after a running transaction), or Rejected (if not allowed). On Accepted, the station should send a final message (like a disconnect, then BootNotification after reboot). Use Soft reset to restart application software, Hard to power-cycle the device.
- **UpdateFirmware** *(CSMS -> Charge Point)*: Initiate a firmware update. Contains a URL to download the firmware, optionally a retrieveDate (start time), a checksum, and possibly a signing certificate. Station responds Accepted/Rejected, then later downloads and installs the firmware. Progress is reported via `FirmwareStatusNotification` events (which include statuses like Downloading, Installing, etc.).
- **SendLocalList** *(CSMS -> Charge Point)*: Update the local authorization list in the station. Contains a list version, update type (Full or Differential), and a list of IdTokens with their authorization status. Station stores this list (for offline auth) and responds. After processing a full update, the station will use the list for offline auth decisions.
- **GetLocalListVersion** *(CSMS -> Charge Point)*: Query the station for its current local auth list version. Station returns a number (or 0 if no list). Used by CSMS to decide if it needs to send updates.

- **UnlockConnector** *(CSMS -> Charge Point)*: Command the station to unlock the cable/connector on a given EVSE (often if a user's cable is stuck). The request typically provides an evseId (and possibly connector if needed). Station should physically unlock (if design supports it) and respond with: Unlocked (success), UnlockFailed (couldn't), OngoingTransaction (refused because a session is active on that connector), or NotSupported.
- **ClearCache** *(CSMS -> Charge Point)*: Instruct station to clear its local IdTag authorization cache. Station should drop all cached authorizations, forcing fresh Authorize checks next time. Response is usually Accepted if it had a cache to clear.

Device & Configuration Management

- **SetVariables** *(CSMS -> Charge Point)*: Primary way to remotely change configuration values on the station ([PDF] Alternative Fuels Infrastructure Regulation - Alfen). Carries a list of Component-Variable pairs with the intended value. Station will attempt to apply each and return a status per entry (in `SetVariablesResponse`), such as Accepted, Rejected, UnknownComponent, etc. This replaces 1.6's ChangeConfiguration. The variety of variables is huge (covering network settings, operational flags, etc.), so stations often implement a subset relevant to their features.
- **GetVariables** *(CSMS -> Charge Point)*: Read configuration variables. The request can list specific Component + Variable names of interest. The station replies with the values or error statuses for each. E.g. CSMS can read current "ChargingScheduleAllowedChargingRateUnit" or "HeartbeatInterval" using this.
- **GetBaseReport** *(CSMS -> Charge Point)*: Requests a standard report from the station about its setup. It can ask for *FullInventory* (all components and all variables and their values), *SummaryInventory*, or *ConfigurationInventory* depending on what info is needed (Command and Control | enua) (Command and Control | enua). The station will respond Accepted and then start sending one or multiple `NotifyReport` messages containing the data (it chunks if needed). This is often used right after BootNotification to get a full description of the station (especially multi-EVSE stations).
- **GetReport** *(CSMS -> Charge Point)*: A more flexible report request where the CSMS can specify criteria like specific component, specific variable, or specific monitoring criteria to report. For example, "report all variables of component X" or "report all variables with a certain attribute". The station responds similarly with one or more `NotifyReport` messages.
- **NotifyReport** *(Charge Point -> CSMS)*: Sent by station to deliver report data requested by `GetBaseReport` / `GetReport`. It can also be used periodically if configured (some stations may send scheduled reports). Each NotifyReport might have a list of ReportData entries, each describing a component, variable, and its value/attributes. The final message in a series will have a flag to indicate it's the last part (`tbc=false` meaning no more to come).
- **SetNetworkProfile** *(CSMS -> Charge Point)*: Configure network connection profiles (like WiFi or cellular APN settings) on the station. This is used if the charger supports OCPP switching networks or connecting via various networks. The payload includes network configurations (APN, username/password, etc.). Station responds with Accepted or NotSupported. (This is part of OCPP 2.0.1's device management but not all stations will implement it).
- **SetMonitoringBase** *(CSMS -> Charge Point)*: Sets the active set of monitoring criteria. Essentially, the CSMS can tell the station which kinds of events to monitor by default: for example, 'All' (enable all standard monitoring), or 'SafetyCritical' (only monitor critical metrics). The station acknowledges and will adjust which events trigger `NotifyEvent` or `NotifyMonitoringReport`.
- **SetMonitoringLevel** *(CSMS -> Charge Point)*: Similar concept, sets the severity level for which events should be reported. For instance, a level 3 might mean report even informative events, whereas level 1 might mean only critical events. Station applies this to filter what it sends.

- **SetVariableMonitoring** *(CSMS -> Charge Point)*: Instruct station to set up a custom monitor on a specific variable. For example, "monitor component=TemperatureSensor, variable=Temperature, and trigger an event when value > 80°C, with severity 5". The request contains one or more such monitor configurations (with a unique Id for each monitor). The station will set these up and respond with status per monitor (Accepted/NotSupported/etc.). If accepted, going forward the station will send `NotifyEvent` or `NotifyMonitoringReport` when conditions are met.
- **ClearVariableMonitoring** *(CSMS -> Charge Point)*: Remove previously set monitors by their IDs. The station stops monitoring those conditions.
- **NotifyEvent** *(Charge Point -> CSMS)*: When a notable event happens within the station (per configured monitoring or internal triggers), the station sends this with details. For example, if a hardware fault occurs on a component, or a monitored variable crosses a threshold, it can send an event with component, eventType, trigger, and severity. This is part of the improved error and state reporting enabled by the Device Model (What is new in OCPP 2.0.1).
- **NotifyMonitoringReport** *(Charge Point -> CSMS)*: Similar to NotifyEvent but can carry a batch of monitored values periodically. It's used to report the current value of all active monitors, perhaps on a schedule. For instance, every hour the station might send a NotifyMonitoringReport listing various sensor readings. This complements NotifyEvent (event is immediate on trigger, monitoring report is periodic summary).

Smart Charging & Energy Management

- **SetChargingProfile** *(CSMS -> Charge Point)*: Sends a charging profile (schedule with power or current limits over time) to the station for a particular EVSE or for the whole station. It includes:
  - An `evseId` (0 for station level or a specific one),
  - A `chargingProfile` with an `id`, a `profilePurpose` (TxProfile, TxDefaultProfile, or ChargePointMaxProfile etc.), a `stackLevel` (priority), and one or more `ChargingSchedule` entries (each with time intervals and limits in amps or watts).
    The station will apply this profile and respond Accepted/Rejected. This is used for **smart charging control**, like load balancing or implementing time-of-use tariffs. In 2.0.1, profiles also can specify energy or currency constraints and tie in with EV's schedule (if EV provides one).

- **ClearChargingProfile** *(CSMS -> Charge Point)*: Removes one or more charging profiles. The CSMS can specify criteria: by EVSE, by profile purpose, by stack level, or a specific profileId. Station will remove matching profiles and respond with success or not. This frees up any limitations set prior.

- **GetChargingProfiles** *(CSMS -> Charge Point)*: Requests the station to send currently active charging profiles. The request can filter by evseId and/or charging profile purpose (Tx, Default, etc.). The station replies with Accepted and then sends `ReportChargingProfiles` (one or more) containing the details of each profile active. This is new in 2.0.1 to allow the CSMS to retrieve what schedules are in effect on a charger.

- **ReportChargingProfiles** *(Charge Point -> CSMS)*: Sent in response to GetChargingProfiles. It contains one or more charging profiles (with all the schedule periods) for given EVSEs. This helps the CSMS audit the station's state in case profiles were set by another system or retained from offline instructions.

- **GetCompositeSchedule** *(CSMS -> Charge Point)*: As in 1.6, this asks the station for a composite view of its planned power draw. The station, considering all active profiles and constraints (and possibly EV's needs), returns a schedule of expected power/current over a requested duration. Useful for external systems (like a facility energy manager) to know the load. The station responds with a schedule (start time, duration, list of power/time values) or with Rejected if not available.

- **ChargingLimit-related (various):** OCPP 2.0.1 introduced finer control for limit enforcement:
  - `ClearedChargingLimit` *(Charge Point -> CSMS)*: The station sends this to inform the CSMS that a limit it was enforcing (perhaps a limit set by a profile or due to a grid issue) has been lifted. E.g., if a temporary current limit is no longer in effect. Not present in 1.6.
  - `NotifyChargingLimit` *(Charge Point -> CSMS)*: Station informs CSMS of an imposed charging limit on an EVSE, often due to an external reason. For example, if the station itself reduced current due to thermal reasons, it can notify the backend of the new limit.

- **CostUpdated** *(CSMS -> Charge Point)*: The CSMS can send this during a transaction to update the price or cost info. For instance, if the energy price changes at a certain time, the CSMS sends the new price per kWh or the total cost so far. The station may use this to update display or to account for cost in the end transaction details. In 1.6 there was no live cost update (price was handled out-of-band usually).

- **CustomerInformation** *(CSMS -> Charge Point)*: This can be used in different ways depending on the parameters – it can request the station to send certain information related to an ongoing transaction or customer (like energy usage, or id data), or even instruct the station to display information to the customer. There are modes in the request (e.g., "Send Personal Message"). The station responds and possibly triggers `NotifyCustomerInformation`. This is a flexible feature introduced in 2.0.1 to enhance how customer-specific data (like messages or transaction details) can be exchanged.

- **NotifyEVChargingNeeds** *(Charge Point -> CSMS)*: If the station supports ISO 15118 and gets info from the EV about its needs (like "I need 50 kWh by 7:00 PM" or "I can only charge at 32A max"), the station will send this message to the CSMS. It contains details like the EV's energy request, departure time, whether the EV is willing to pause charging, etc. The CSMS can use this to adjust charging plans across multiple stations. No equivalent in 1.6 (which wasn't aware of EV's intents).

- **NotifyEVChargingSchedule** *(Charge Point -> CSMS)*: Also related to ISO 15118 / smart charging. The EV might negotiate a charging schedule with the station (especially in V2G scenarios). The station then informs the CSMS of the agreed schedule via this message. This allows the CSMS to integrate the EV's plan into higher-level grid management.

Diagnostics & Maintenance

- **GetLog** *(CSMS -> Charge Point)*: Request the station to upload a log file. Parameters include log type (for instance, "DiagnosticsLog" for general debug logs or "SecurityLog"), a requested log duration or entries, and an optional URL where to send it (if the CSMS wants it to be uploaded to a server). Station responds and, if Accepted, proceeds to gather the log. It will then either send it via a separate secure channel or to the given URL, and finally send `LogStatusNotification`.
- **LogStatusNotification** *(Charge Point -> CSMS)*: Indicates the result of a log upload requested by `GetLog`. Status could be Uploading, Uploaded (success), UploadFailed, AccessDenied, etc. Informs the CSMS if the diagnostics log or security log was successfully delivered. Replaces `DiagnosticsStatusNotification` from 1.6 ([PDF] OCPP 2.0.1: Part 2 - Errata - Regulations.gov).
- **FirmwareStatusNotification** *(Charge Point -> CSMS)*: Station's notifications about firmware update status. Sent for events such as:
  - *Downloading*: started download from URL,
  - *Downloaded*: finished download (ready to install),
  - *DownloadFailed*: couldn't download (e.g., network error),

  - *Installing*: started applying the firmware,
  - *Installed*: firmware updated successfully (station will usually reboot after this),
  - *InstallationFailed*: update failed (possibly will revert).
    This allows the CSMS to track firmware updates in real time (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison). In 1.6, fewer states were defined. 2.0.1 expects robust use of this.
- **PublishFirmware** *(CSMS -> Charge Point)*: This is used in scenarios where a CSMS wants a station to share firmware with other stations (perhaps via a controller or when a station manages a cluster). It contains a download link and checksum similar to UpdateFirmware, but also an *Md5* of the firmware and a *target* (like "all stations of model X"). The station receiving this might act as a seed to distribute firmware. If your charger is not doing such forwarding, you might not implement this. After trying to distribute, it will send `PublishFirmwareStatusNotification`.
- **PublishFirmwareStatusNotification** *(Charge Point -> CSMS)*: Reports status of a PublishFirmware process (e.g., whether the station managed to distribute the firmware to others). Most chargers that are not hubs won't use this.

Security & Certificate Management

- **SignCertificate** *(Charge Point -> CSMS)*: The station sends this to request the CSMS (or a subordinate CA) to sign a Certificate Signing Request (CSR). It's typically used to get a new client certificate for the station. The request contains the CSR in PEM format and a type (what it's for – e.g. ChargingStationCertificate). The CSMS responds with `CertificateSigned` (or an error) asynchronously or via the response.
- **CertificateSigned** *(CSMS -> Charge Point)*: This can be a response or an unsolicited message to deliver the signed certificate (and possibly chain) to the station. If `SignCertificate` is used, the CSMS might respond immediately with the certificate chain in this message. The station then should install the certificates in its keystore and reply with status (Accepted/Rejected).
- **InstallCertificate** *(CSMS -> Charge Point)*: Instructs the station to install a certificate (CA or client cert). Contains the certificate type (one of: CA for CSMS, CA for MF (manufacturer), V2G root, MORoot for mobility operator, or ChargingStationCertificate) and the PEM certificate. Station stores it in appropriate store and returns status (What is new in OCPP 2.0.1). This is how a new trust anchor or a new local certificate can be provisioned remotely.
- **GetInstalledCertificateIds** *(CSMS -> Charge Point)*: CSMS queries which certificates (by type) the station currently has installed. The station replies with a list of certificate types and their identifiers (like thumbprints). This helps the CSMS know if a station already has the latest certs or if it needs updates.
- **DeleteCertificate** *(CSMS -> Charge Point)*: Tells the station to delete a certificate of a given type or with a given hash. E.g., remove a specific expired root CA from your trust list. Station attempts removal and responds with success or not.
- **SecurityEventNotification** *(Charge Point -> CSMS)*: Station notifies about a security-related event. For example, "Invalid Firmware Signature" event, or "TLS handshake failed", or physical tamper detection. The message includes a type (predefined event name), timestamp, tech info like UID if needed. The CSMS can log or alert these.

Display, Messaging & Customer Interaction

- **SetDisplayMessage** *(CSMS -> Charge Point)*: Allows the CSMS to post a message on the charging station's display. The request includes a message ID, priority, start/stop time for display, and the content (which can have multiple languages). The station stores it in a "message queue" and displays according to scheduling. E.g., display "Welcome " when a session starts, or show advertising. Responds with Accepted/Rejected.

- **GetDisplayMessages** *(CSMS -> Charge Point)*: CSMS requests the list of display messages currently queued on the station (with statuses). The station replies with one or more `NotifyDisplayMessages` containing the messages info (id, priority, status like "Waiting" or "Displaying").
- **ClearDisplayMessage** *(CSMS -> Charge Point)*: Remove a message from the station's display queue (by id). Station removes it if present and responds. If the message was currently showing, station may clear it off the screen.
- **NotifyDisplayMessages** *(Charge Point -> CSMS)*: Station's answer to `GetDisplayMessages`, or unsolicited update if a message's status changed. It lists message entries and their current state on the charger (e.g., message #3 is now displaying or expired).
- **CustomerInformation** *(CSMS -> Charge Point)*: (Mentioned earlier in smart charging too) – It has multiple uses:
  - If `requestId` and reportRequested are given, station should send `NotifyCustomerInformation` with the data (like customer consuption info).
  - If a message to display is included, station could display it to customer.
  - It can also be used to request ongoing reports of customer-related data.
    Essentially a multi-purpose message for any customer-centric feature not covered elsewhere.
- **NotifyCustomerInformation** *(Charge Point -> CSMS)*: Station's response or update to a CustomerInformation request. Could carry data like current energy consumption of a specific user's session, personal messages delivered, etc., depending on how it's used.

Reservation and Availability Updates

- **ReservationStatusUpdate** *(Charge Point -> CSMS)*: Station informs CSMS about the outcome of a reservation. For example, after a `ReserveNow`, when the reservation start time arrives, the station will send ReservationStatusUpdate with status "Expired" if the driver didn't show up, or perhaps "Accepted/Active" when the EV plugs in and uses the reservation. Notifies the backend to free the reservation slot in its system.
- **Heartbeat** *(Charge Point -> CSMS)*: (Already covered under Core, but relevant to availability if a station misses heartbeats, CSMS might mark it offline.)

This glossary is a quick rundown – for detailed structures and additional optional fields, refer to the official OCPP 2.0.1 Specification. Each message often has many sub-fields especially in 2.0.1. But this list should help identify what needs to be implemented and tested when moving from OCPP 1.6 to 2.0.1.

Conclusion

Migrating to OCPP 2.0.1 brings a wealth of new possibilities to EV charging stations: richer communication, better control, and future-ready features like Plug & Charge and V2G. It also introduces complexity that requires careful implementation. By understanding the core differences and following best practices outlined in this guide, engineers can methodically upgrade their charger firmware. Testing all scenarios (normal operation, error cases, offline, security events) is crucial given the broader scope of OCPP 2.0.1. The result will be a charging station that is interoperable with the latest CSMS platforms and prepared for the advanced ecosystem of EV charging.

References: Key points in this guide reference the OCPP specifications and expert analyses for accuracy, such as the unification of transaction messages (What is new in OCPP 2.0.1), the introduction of the EVSE device model (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison), new security features (OCPP 1.6 vs. OCPP 2.0: A Comprehensive Comparison), and others. These ensure that the descriptions and recommendations are aligned with official OCPP 2.0.1 documentation and industry understanding as of 2025.