ChatGPT

# IST Trip Actions: Validations and Prerequisites

## 1. Loading – `validateISTForLoading` Checks

The method `ISTHandler.prototype.validateISTForLoading` performs several **pre-loading validations** before an Inter Stock Transfer (IST) can be loaded:

- **Support Admin Bypass:** If the action is being performed by a support admin user, all operational validations are skipped (the function returns immediately) [1] .
- **Hub Existence:** It requires a `hubId` parameter and ensures that the hub exists. If `hubId` is missing or the hub is not found in the database, it throws errors (“Hub Id should be present” or “Hub not found”) [2] [3] .
- **Destination Hub & Route Validity:** If a `routeDestinationHubId` is provided (i.e. the waybill/consignment is supposed to go to a specific hub on the route), the code checks that:
- The destination hub exists (throws “Destination Hub not found” if it doesn’t) [4] .
- The IST’s route contains that destination hub. It fetches the route details for the IST’s `route_id`; if no route details are found, it errors (“No route details found for the given IST”) [5] . If the route exists but the specified destination hub is not one of the route’s stops, it throws “Invalid Route Destination Hub chosen” [6] .
- **App-Only Operation:** If the target hub is configured to allow load/unload actions **only via the Hub Ops mobile app** (hub’s `extra_details.app_based_load_unload_only` flag), then the validation ensures the request is coming from that app. If not (e.g. an attempt from web), it rejects the loading action (“Selected operation is only allowed from Hub Ops App. Please use Hub Ops App”) [7] .
- **Sequence of Events Constraints:** The organization-level IST configuration (`ist_config_master`) may impose rules on the sequence of events. The code checks three flags and ensures none are violated [8] :
- *“No Loading without Receive”:* If `donot_allow_loading_unloading_without_receive` is true, and the current hub is **not** the origin hub of the IST, then an **“IST Received”** event must exist for this IST at the hub before loading. If the IST hasn’t been received at this hub yet, loading is not allowed (“Loading not allowed before IST Receive” is thrown) [9] .
- *“No Loading after Depart”:* If `donot_allow_loading_after_depart` is true, the presence of an **“IST Departed”** event at this hub will block any new loading. In other words, once a vehicle has departed the hub, you cannot load additional consignments there (“Loading not allowed after IST Depart”) [9] .
- *“No Multiple Loading” (Freeze once):* If `donot_allow_multiple_loading_unloading` is true, the IST cannot be loaded again if it’s already been marked as loaded (frozen) at this hub before. The code checks for an existing **“IST Loaded”** event; if found, it throws “Loading not allowed after IST Freeze” [10] . This prevents multiple load->unload cycles on the same IST at one hub.

These checks must all pass (or be bypassed for support admins) before the actual loading operation can proceed.

## 2. Freezing the Trip – Validations in `ISTHandler.prototype.freeze`

The **freeze** action finalizes the loading process, marking the IST as fully loaded (no further loading allowed). The `freeze` method performs its own validations on top of the loading checks:

- **Basic Trip State Checks:** The IST must exist and not already be completed (closed) [11] . Also, the IST's `current_hub_id` must match the hub where freeze is called – you can only freeze at the hub that currently holds the IST. If the IDs differ, it errors "IST does not belong to given hub currently" [12] . Furthermore, the IST **cannot be frozen at its destination hub** (since loading at the destination makes no sense), so if `hubId` equals the IST's destination_hub_id, it throws "IST loading not allowed at destination hub" [13] .
- **Vehicle Assignment:** For surface transport ISTs, a vehicle must be assigned before freezing. If the IST has no `vehicle_id` (and its mode is not AIR), the system blocks the freeze ("Vehicle should be associated with IST") [14] . This ensures the trip has a conveyance ready.
- **Child Consignment Completeness:** The freeze logic then verifies that all **child consignments** (if any) of loaded parent consignments have been scanned in. It compiles the current "loading draft" (items scanned for loading) and checks for partially loaded consignments. Any parent consignment in the draft whose some child pieces are missing will be flagged.
- If the org config disallows "shortage loading" (`allow_shortage_loading` is false), the presence of **any** partially loaded consignment causes an immediate failure. The code gathers the IDs of such consignments and throws an error like: "Please load all child consignments for X, Y" (listing the ones not fully loaded) [15] . In other words, **all pieces of a multi-piece consignment must be loaded** unless shortage loading is enabled.
- If shortage loading *is* allowed, the freeze will proceed but still handle the missing pieces: it calls `validateISTForLoading` again at this point to re-run all the loading validations (from step 1) one more time [16] , ensuring nothing has changed (e.g. no late depart event, etc.). Then it continues to assign the loaded items to the IST. Any child consignment that was not scanned is automatically marked with an exception. Specifically, for each missing child consignment, the system creates a `SHORTAGE_LOADING` **exception** via `TaskConsignment.applyConsignmentException`, flagging that piece as not loaded [17] . This happens in a loop over all child consignments of each parent: if a child was not in the loaded list, it's marked as a shortage during loading [18] [19] . The parent consignment itself is then marked as added to the IST with a partial load, and it too gets a `SHORTAGE_LOADING` exception noted (indicating it departed with some pieces short) [20] [21] .
- **Partial Loading Rules & Edge Cases:** Even when shortage loading is allowed, the code enforces some edge-case rules:
- **No Partial for Dummy HU:** If any consignment uses a *dummy handling unit* (a special flag `is_dummy_hu` on child items) and the config `restrict_partial_loading_for_dummy_hu` is true, then **partial loading is completely disallowed for that consignment**. Encountering a dummy HU with missing children causes an immediate error: *"Partial Loading for Component Article <consignment_ref> not Allowed"* [17] . This prevents leaving behind pieces when using dummy packages.
- **Restricted Delivery Types:** The configuration may list certain delivery types that cannot be partially loaded (`restrict_partial_loading_delivery_types`). For any parent consignment of those types, if not all child pieces are loaded, the code collects those cases and aborts the freeze. It throws an error like *"Partial Loading Not Allowed for Consignments: [{ParentRef: Pieces Not Scanned(...)}]"* listing each affected parent and its missing pieces [22] .
- **Customer Allowances (PGI):** There's a check related to customer settings: if a parent consignment's customer has `allow_partial_loading` enabled *and* a certain PGI (Proof of

Goods Issue) process is not completed (`pgi_details` not set), the system *skips* strict enforcement for that consignment [23] [24]. In other words, if the customer permits partial loads before PGI, it won't throw an error for that particular case – it will allow freeze to continue (treating it as an expected partial scenario).
- **Finalizing the Load:** If all the above checks pass (and any necessary exceptions on short-loaded items are applied), the `freeze` method records the event and updates the IST status. It creates an **ISTLoaded event** and updates the IST's status to `'loaded'`, setting the `last_event_time` to the freeze time. It also clears out the `extra_details.loading_draft` (since those items are now finalized on the trip) and records the total loaded weight [25]. After `freeze`, the IST is effectively sealed for dispatch – no further loading can occur until it's unloaded at the destination.

## 3. Unloading – `validateISTForUnloading` and Usage in Start/Finish Unloading

**Unloading** an IST at the destination (or an intermediate hub, in multi-hop scenarios) also has guard conditions. The method `ISTHandler.prototype.validateISTForUnloading` is invoked at the start of unloading and at the finish to ensure everything is in order:

- **Pre-unloading Validation (`validateISTForUnloading`):** This function parallels the loading validator, with checks appropriate for the unload phase:
- It requires a valid IST object in context and a transaction. If the IST isn't loaded or is already completed, it throws ("IST not found" or "IST already Completed") [26].
- A `hubId` must be provided; if missing, it throws "Hub Id should be present" [27]. It then looks up that hub in the DB; if not found, "Hub not found" is thrown [28].
- If the IST's `current_hub_id` doesn't match the provided hub (meaning the trip isn't currently at this location), it aborts unloading ("IST Hub mismatch") [29]. This ensures you unload at the correct hub.
- It honors the *app-based operations only* setting similar to loading: if the hub is restricted to app-only and the call isn't flagged as coming from the Hub Ops app, it throws the same app-only error [30].
- It then checks organisation settings for unloading order:
  - If **no unloading without receive** is enforced (`donot_allow_loading_unloading_without_receive`), the IST must have a prior "received" event. Without an IST Receive event, unloading is blocked ("Unloading not allowed before IST Receive") [31]. This typically means the inbound vehicle should be marked as arrived (received) at the hub before you start unloading it.
  - If **no unloading after depart** is set (`donot_allow_unloading_after_depart`), then if an "IST Departed" event is already recorded at this hub, it won't allow unloading ("Unloading not allowed after IST Depart") [31]. (In practice, this prevents unloading after the vehicle has been dispatched onward from this hub.)
  - If **no multiple unloading** (`donot_allow_multiple_loading_unloading`) is true, the system forbids a second unload event. If an "unloaded" event for this IST at this hub already exists, it throws "Unloading not allowed after IST Unloaded" [32]. (This ensures you don't unload the same IST twice at the same location.)

- Like with loading, support admins are exempted – if `isSupportAdmin` is true, the function returns early without enforcing these validations [33].

- **During `startUnloading`:** When the unloading process begins, the system calls `validateISTForUnloading` to enforce all the above rules before changing any state [34] [35]. Assuming validation passes, `ISTHandler.prototype.startUnloading` will create an **ISTUnloading event** (recording that unloading has started at the hub, along with metrics like how many consignments/bags to unload) and update the IST's status to `'unloading'` [35] [36]. At this point the trip's state is in-progress unloading. (If any of the validations fail, the unloading start is aborted and the appropriate error is returned to the user.)

- **During `finishUnloading`:** This is the step that finalizes the unloading. At the beginning of `ISTHandler.prototype.finishUnloading`, the code again invokes `validateISTForUnloading` to ensure nothing has violated the rules in the interim [37] [38]. It also double-checks basic state: the IST must exist, not be already completed, and still be at the same hub (similar to `startUnloading` checks) [39]. Once validation passes, the finishUnloading logic proceeds to:

- **Determine Unloaded vs. Missing Items:** It looks at the IST's `unloading_draft` (the list of waybills actually scanned/unloaded) and compares it to the expected `way_bills_to_unload` (all items that *should* be unloaded at this hub) [40]. Any waybills present in the plan but *not* found in the unloaded draft are considered not unloaded (`wayBillsNotUnloaded`).
- **Auto Exceptions for Shortage:** If the org has enabled auto-application of exceptions for missing waybills (`waybill_exception_config_master.auto_apply_exception` is true) and the caller did not explicitly skip this, the code will iterate over each not-unloaded item and mark it as a shortage. For example: for every bag not unloaded, it calls `BagHandler.applyBagException` with internal code `'SHORTAGE'` [41]; for each consignment not unloaded, it initializes a TaskConsignment and calls `applyConsignmentException('SHORTAGE')` to flag that consignment as not received [42]. These exceptions indicate that those items were supposed to arrive but did not – effectively marking them lost or pending investigation. (If a parameter `skipUnloading` is true, the function can bypass automatically applying these exceptions [43] – presumably to handle such cases manually.)
- **Mark Items as Unloaded:** Next, for items that *were* scanned in the `unloading_draft`, the system updates their status. If the config `mark_item_inscan_at_unloading_scan` is **false** (meaning the act of unloading should also mark the item as arrived/inscanned into the hub's inventory), the code explicitly processes each unloaded item now. It calls `unloadBagFromIST` for each bag in the draft and `unloadConsignmentFromIST` for each consignment [44] [45]. This will update the status of those consignments/bags (e.g. marking consignments as arrived at hub). It also does a bulk update to mark all unloaded consignments as "Inscanned at Hub" in one go [46].
- **Finalize Unload Event:** Finally, an **ISTUnloaded event** is created to log the completion of unloading, and the IST's status is updated to `'unloaded'` [47] [48]. In this update, `extra_details.unloading_draft` is cleared (since unloading is done). Notably, any items that were not unloaded remain listed in `extra_details.way_bills_to_unload` (now essentially a record of shortages/missed waybills) [48]. After this, the IST at this hub is considered fully unloaded (even if some consignments were missing), and the event is submitted to record the state change. The return value typically includes `numWayBillsUnloaded` (the count of waybills successfully unloaded) [49] [50].

In summary, **both** `startUnloading` and `finishUnloading` rely on `validateISTForUnloading` to enforce prerequisites (correct hub, no rule violations like missing "Receive" event or prior depart). The `finishUnloading` step additionally handles the outcome for each expected item – updating statuses for unloaded consignments and marking missing ones with exceptions – before marking the IST as unloaded.

## 4. Trip Close/Completion – Validations in `ISTHandler.prototype.complete`

Closing out an IST trip (marking it **Completed**) also has strict validations, implemented in `ist-inbound.js` within `ISTHandler.prototype.complete`. This is the final step after an IST is fully unloaded (in most cases). Key validations include:

- **Trip State & Identity:** The IST must exist in context (`requiredIST`) and not already be completed [51]. If it's already marked complete, the call is invalid ("IST already Completed").
- **Correct Hub for Completion:** Generally, an IST trip should be closed at its final destination hub. The code checks that the `hubId` provided for completion matches the IST's current hub *and* the IST's `destination_hub_id`, unless the IST is of a special type. If the IST's type is not PTL (explained below), and the current hub or destination hub don't match the provided `hubId`, it will throw an error ("IST Hub mismatch" if current hub is different, or "IST Destination Hub mismatch" if the hub is not the destination) [52]. This effectively ensures a standard IST can only be completed at the intended destination where it ended up.
- **PTL Trip Exception:** If the IST's type is **PTL** (presumably "Partial Truck Load" or a special workflow), the above hub matching rules are relaxed. Instead, there is a configurable rule: if `ist_config_master.close_ptl_trip_only_at_origin_hub` is true, the system requires that a PTL-type trip be closed **at the origin hub**. In that case, completing it at any other hub throws "PTL Trips can only be closed at Origin Hub." [53]. (If that config is false or not set, the implication is a PTL trip *could* be closed at a non-destination hub, which might be the case for certain return or cancel scenarios. The code explicitly skips the normal dest/current hub checks for PTL trips [54].)
- **KM Reading (Odometer) Validation:** For organizations that track vehicle odometer readings on trip actions (`midmile_ops.use_km_reading_in_ist_actions` flag), the completion step expects an `endKMReading` in the input. If this feature is enabled:
- The `endKMReading` must be provided and cannot be null/undefined (or it throws "KM reading should be present") [55].
- The value must be a valid non-negative number (if it's not a number or is negative, "Invalid KM reading" is thrown) [56].
- It must also be **greater than or equal to** the last recorded KM reading for the trip. The code compares against `requiredIST.extra_details.km_reading` (presumably the start reading); if the end reading is smaller, it errors: "End KM reading cannot be less than Previous KM reading; Previous Reading: X" [57]. This prevents bogus odometer rollbacks.
- **Ensure Unloading is Done (or Not Required):** Perhaps the most important check for trip completion is verifying that all consignments have been unloaded (for standard trips). The `complete` method checks if there are any waybills still marked as `'loaded'` on this IST – it does so by looking up any `InterStockTransferRelation` with status `'loaded'` for this IST [58]. If it finds one, that means some item was never unloaded. In such a case, **and if the IST is not a PTL-type**, it will **prevent completion**. The code looks for an ISTEvent of type `'unloaded'` at this hub as evidence that an unload occurred; if no such event is found, it throws: *"IST must be unloaded before completion."* [59]. This effectively forces the normal workflow: you **cannot close a regular IST trip without performing the unload step for all consignments** (or at least recording an unload event for them). If the unload event exists (meaning `finishUnloading` was called, even if some items were marked shortage), then this check passes. For PTL trips, this entire check is skipped (the condition is gated by `if`

`(wayBillLoaded && istType !== PTL)` [60] ), implying that for PTL-type ISTs the system may allow completion even if not all items went through a standard unload process.

After all validations pass, the completion logic proceeds to create an **ISTCompletion event** and update the IST's status to `'completed'`. It timestamps the completion ( `last_status_change_time` etc.), and updates `extra_details` with final information like the `end_km_reading`, completion comments, and possibly distance calculations (if chargeable distance is computed at trip end) [61] [62]. The IST is then officially closed out.

## Closing a Trip Without Unloading All Consignments?

In general, the system is designed **not** to allow closing an IST trip without unloading its consignments – for a typical IST, you must perform the unload (or mark items as unable to unload) before calling complete. The check for any `'loaded'` waybills ensures that if anything is still in transit, the trip can't be marked complete [59]. However, there are scenarios and configurations that effectively allow an IST to be closed without every consignment being unloaded:

- **PTL Trips:** As noted, PTL-type ISTs can bypass the usual unload requirement. This could mean a PTL trip might be closed at origin or mid-route (depending on config) without a formal unload at destination. In such cases, consignments that were never unloaded would remain associated with the trip. What happens to them? Internally, those consignments would still be in a "loaded" state but the trip is closed – this is an unusual scenario typically gated by business rules. The expectation is that PTL trips are handled differently (e.g. perhaps they represent leg-wise transfers rather than a whole vehicle going hub-to-hub). If closed early, those consignments might be re-routed or handled outside the scope of this IST. (The code does not explicitly remove them; it simply doesn't require the unload event for PTL, implying it's an accepted outcome for that flow.)

- **Forced/Skipped Unloading:** The system provides a `skipUnloading` flag in the finishUnloading method, and also will mark missing items as shortages. If, for example, an IST arrived with **none of the expected consignments** (hypothetically, an empty vehicle or all packages lost), an operator could call `finishUnloading` with `skipUnloading=true` to essentially finalize the trip without scanning anything. In that process, any expected waybills would end up in the **not unloaded list** (`way_bills_to_unload`). If auto-exception is on, the system would normally mark them as SHORTAGE during a regular finish (when not skipping) [41] [42]. With skipUnloading, the code bypasses the scanning and potentially the auto-exception application, but it still creates the ISTUnloaded event and updates the IST status to 'unloaded' with those items remaining in the `way_bills_to_unload` array [40] [48]. Essentially, the trip is closed out as unloaded/complete, but those consignments are flagged as not unloaded. They would carry the **"SHORTAGE" exception status** if applied (or could be manually marked later). The presence of an unload event (even if everything was skipped) satisfies the completion check, so the IST can be completed. Once completed, those consignments are left in an unresolved state – effectively marked as missing in transit. They are not magically delivered; rather, they remain in the system for reconciliation (the trip's record shows they were never unloaded).

In summary, **under normal configurations the IST cannot be closed without unloading all consignments**, as the software insists on an unload step (or at least an unload event with exceptions for missing items). Only with special trip types (like PTL) or admin overrides can this be done. In those exceptional cases, any consignments not unloaded are **not simply ignored** – the system will tag them with appropriate exception codes (such as 'SHORTAGE') and leave them in the IST's data (`way_bills_to_unload`) to indicate they were not received [48]. This way, even though the trip is marked complete, there is a record that certain consignments were never unloaded and they would likely require further action (investigation or reallocation in the system).

**Sources:**

1. Shipsy ProjectX code – *ISTHandler.validateISTForLoading* in **ist-outbound.js** [1] [63]
2. Shipsy ProjectX code – *ISTHandler.freeze* (loading finalization) in **ist-outbound.js** [13] [64] [65] and partial-load handling [17] [22]
3. Shipsy ProjectX code – *ISTHandler.validateISTForUnloading* in **ist-inbound.js** [66] [31]
4. Shipsy ProjectX code – *ISTHandler.startUnloading* in **ist-inbound.js** [35] [36]
5. Shipsy ProjectX code – *ISTHandler.finishUnloading* in **ist-inbound.js** [40] [41] [48]
6. Shipsy ProjectX code – *ISTHandler.complete* (trip completion) in **ist-inbound.js** [52] [55] [59]

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [63] [64] [65] ist-outbound.js
https://github.com/shipsy/projectx/blob/57f41ecf13cdf1921b26e31dc931db4bcf9119cf/common/domain-models/ist-handler/ist-handler-parts/ist-outbound.js

[26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [66] ist-inbound.js
https://github.com/shipsy/projectx/blob/57f41ecf13cdf1921b26e31dc931db4bcf9119cf/common/domain-models/ist-handler/ist-handler-parts/ist-inbound.js