Trip Planner APIs: Manual Open Data Production Technical Documentation

Mar 2017 | Version: 3.0

Table of Contents

Author:	Transport for NSW
Date:	Mar 2017
Version:	3

1. Transport for NSW API

This document describes the Transport for NSW Trip Planner APIs. It is intended to complement the TfNSW Open Data Trip Planner API Swagger Documentation.

1.1. Introduction

The API allows users to search for trips, stops, service alerts and places of interest using the following five request types:

1. **Stop Finder API**: Provides capability to return all NSW public transport stop, station, wharf, points of interest and known addresses to be used for auto-suggest/auto-complete (to be used with the Trip planner and Departure board APIs).
2. **Trip Planner API**: Provides capability to provide NSW public transport trip plan options, including walking and driving legs, real-time and Opal fare information.
3. **Departure API**: Provides capability to provide NSW public transport departure information from a stop, station or wharf including real-time.
4. **Service Alert API**: Provides capability to display all public transport service status and incident information (as published from the Incident Capture System).
5. **Coordinate Request API**: When given a specific geographical location, this API finds public transport stops, stations, wharfs and points of interest around that location.

Each of these calls performs complementary functions so that you can combine them to make a comprehensive trip planner for the New South Wales public transportation network.

This document discusses each of the API calls in greater depth than the TfNSW Open Data Swagger documentation and demonstrates realistic usage examples to help you make effective use of the API.

A Note about Usage Examples

In the Getting Started chapter there are two ways presented for accessing the API data: accessing JSON response data directly, or using the TfNSW Open Data Swagger documentation to build a client library using your development language.

The usage examples in this manual will use PHP code directly accessing the JSON response data. This makes the examples easier to comprehend for non-PHP users than if we worked with a client library generated for PHP using the Swagger documentation.

A Note about Caching

In this document we don't discuss caching strategies (as in, saving response data for frequently used requests to speed up improvement of your web site or app). Since trip planning data can update frequently (e.g. daily, depending on the route or its operator), you may not want to cache such responses at all.

However, other API calls - such as retrieving Opal resellers using the Coordinate Request API - won't update their data as frequently. You may want to cache such data for a short period of time, as this will improve the performance of your applications and improve the user experience.

It is important to consider the type of data you want to cache to ensure stale data is never presented to your users.

A Note about Response Data

In order to retrieve response data in the format covered by this manual and the corresponding Swagger documentation, you must include a request parameter called `outputFormat` with a value of `rapidJSON`. The Getting Started chapter discusses JSON in further detail.

In order to retrieve coordinates in the standard format (EPSG 4326) that services such as Google Maps, or many modern programming languages use, you must include a request parameter called coordOutputFormat with a value of EPSG:4326. This is true for all API calls in the TfNSW API.

Getting Started

The API returns responses in JSON format. There are primarily two ways you can programmatically handle these responses:

1. Manually build request URLs, then perform the HTTP request, then parse the returned JSON data manually in accordance with the TfNSW Open Data Swagger Documentation. This takes more work but gives you full control of how the data is handled.

2. Use `swagger-codegen` to generate usable code in your language of choice. This is a simpler way to get up and running with the API, especially for handling future versions of the API, where the response data format may have changed.

JSON

All API calls return JSON data. JSON (an acronym for JavaScript Object Notation - although it is not specific to JavaScript at all), is a lightweight data-interchange format that is made up of the following six types of data:

•   Strings (e.g. `"Hello"`)

•   Numbers (e.g. `1.5` or `67`)

•   Booleans (`true` or `false`)

•   Null (empty value)

•   Arrays (an ordered collection of zero or more of any of these six types, e.g. `[ "Hello", 1.5, false ]`)

•   Objects (a unordered collection of zero or more of any of these six types, each accessible using a unique string key, e.g. `{ name: 'Mary' }).`

This is an extremely simplified introduction to JSON, but you should be familiar with JSON if you are using this API.

All API calls to the TfNSW Trip Planner APIs must include a URL parameter called `outputFormat` with a value of `rapidJSON`. This is used to enable the JSON output mode.

Manually Performing Requests and Handling Responses

You can manually perform HTTP requests in the language of your choice and the handle the response manually. In this case, you would need to create the request URL manually and then parse the JSON response accordingly.

For example, in PHP you could perform a Stop Finder request as follows:

Note: the following example is very basic and doesn't handle errors such as network connectivity issues or validate response data. If you're using using PHP to perform API requests, it is recommended you use the cURL functions instead.

```php
<?php
  $apiEndpoint = 'https://api.transport.nsw.gov.au/v1/tp/';
  $apiCall    = 'stop_finder';

  $params = array(
    'outputFormat'     => 'rapidJSON',
    'type_sf'          => 'any',
    'name_sf'          => 'Circular Quay',
    'coordOutputFormat' => 'EPSG:4326',
    'anyMaxSizeHitList' => 10
  );

  $url = $apiEndpoint . $apiCall . '?' . http_build_query($params);

  $response = file_get_contents($url);
```

Assuming this response is a valid JSON response, you can access the data by converting it into a native JSON data structure then extracting the data you require. The following example shows how to output the name from each returned location from the above request.

```php
<?php
  // ... continue from above code

  $json = json_decode($response, true);

  $locations = $json['locations'];

  foreach ($locations as $location) {
    echo $location['name'] . "\n";
  }
```

You can follow a similar principle for each of the other API calls also.

Using swagger-codegen to Build a Client Library

Swagger-Codegen is a third-party tool available from https://github.com/swagger-api/swagger-codegen. It is used to convert a Swagger specification file (such as the one provided for the TfNSW API) into the native code of your app / web site.

One of the primary reasons why using Swagger-Codegen is useful instead is that you can regenerate the client library code when new versions of the API are released, and you will quickly be able to see where these changes impact your code.

A list of available languages that can be generated is available from the above URL also.

Once you have installed `swagger-codegen`, use the following command to retrieve help for generating code:

Note: Alternatively, there is a web-based code generator at https://generator.swagger.io.

To generate code, use the `swagger-codegen generate` command. For example, to generate PHP code to access the TfNSW API, use the following command:

```
$ swagger-codegen generate \
    -i https://
opendata.transport.nsw.gov.au/sites/default/files/swagger/TripPlanner.json \
    -l php \
    -o /path/to/generated/files
```

Note: the language code is being generated for is specified by the `-l` *parameter. For instance, if*

you wanted to generate Java code, you would use `-l java` *instead.*

For additional options that can be used with `swagger-codegen generate`, use the following command:

```
$ swagger-codegen help generate
```

When you have successfully generated a client library for the API in your chosen language, you can then import the generated files into your project.

For example, if you have generated PHP files, you can perform the same Stop Finder request as in the previous section as follows:

```
<?php

  $api = new Swagger\Client\Api\DefaultApi();

  try {
    $result = $api->tfnswStopfinderRequest(
      'rapidJSON',
      'any',
      'Circular Quay',
```

```
      'EPSG:4326',
      10
  );

} catch (Exception $e) {
  echo 'Exception: ', $e->getMessage(), "\n";
  exit(1);
}

$locations = $result->getLocations();

foreach ($locations as $location) {
  echo $location->getName() . "\n";
}
```

Note: In this particular case, the PHP client library improves error handling by using exception handlers. Additionally, you do not need to specify the API endpoints (URLs), since these are read from the Swagger specification.

If your development environment includes code auto completion, it is much easier to discover which response values are available, compared to manually parsing the JSON data yourself.

1.2. Stop Finder API

The Stop Finder API (also known as `stop_finder`) is used to search for stops, places of interest, or other locations you can travel between on the TfNSW network. Unlike `coord`, you search based on names or IDs instead of a single coordinate.

The primary use-case for this API call is so users can find a starting or finishing location for a trip planning request (performing trip planning requests is covered in the chapter about `trip`).

If multiple locations are returned, each location can be presented to the user so they can then manually choose (either on a map or by its title) the location they are looking for. Each location has a corresponding unique identifier that can be used in other API calls to reference the given location.

Example: Searching for a Stop

Consider the scenario where somebody flies from Melbourne to Sydney and wants to travel from the domestic airport to Circular Quay.

In this situation, two searches would be needed: one for the airport station and another for Circular Quay.

Assuming the user is presented a search form, into which they simply entered `Airport`, let's say their input is in the `$searchQuery` variable.

```php
<?php
  // User's search query
  $searchQuery = 'Airport';

  $apiEndpoint = 'https://api.transport.nsw.gov.au/v1/tp/';
  $apiCall    = 'stop_finder';

  // Build the request parameters
  $params = array(
    'outputFormat'     => 'rapidJSON',
    'odvSugMacro'      => 1
    'name_sf'          => $searchQuery,
    'coordOutputFormat' => 'EPSG:4326',
    'TfNSWSF'          => 'true'
  );

  $url = $apiEndpoint . $apiCall . '?' . http_build_query($params);

  // Perform the request and build the JSON response data
  $response = file_get_contents($url);
  $json = json_decode($response, true);

  // Extract locations from response
  $locations = $json['locations'];

  // This will hold the best match that is returned
  $bestMatch = null;

  foreach ($locations as $location) {
    // Only one returned location will have isBest set to true
    if ($location['isBest']) {
      $bestMatch = $location;
      break;
    }
  }

  if (is_null($bestMatch)) {
    // No best match found
  }
  else {
    // Location ID of 10101331
    $locationId = $location['id'];

    // Location Name of "Sydney Domestic Airport Station, Mascot"
    $name = $location['name'];
```

```
    // ... Do something with the location
  }
```

The API returns a value called `matchQuality` which indicates how well the location matches the search term. Additionally, the value for `isBest` is set to `true` for the returned location with the highest `matchQuality` score.

Since many locations were returned, you may instead want to allow the user to choose which location to use. In this case, you may want to sort the return locations by their `matchQuality` score (the higher the better).

You can then repeat this entire process again for the destination location (in this example, Circular Quay). Once this is complete, you will have a stop ID for both the origin and destination. You can subsequently use these IDs in a call to the Trip Planner API.

Location Types

Each location returned by `stop_finder` has a corresponding location type. It is important to handle each location type accordingly, as the data made available with the location is dependent upon its location type.

For example, if the type value is `stop`, then the location will include an array called `modes`, which indicates the modes of transport that service the stop.

```php
<?php
  // ... Perform request and extract a location

  if ($location['type'] == 'stop') {
    $modes = $location['modes'];

    if (in_array(4, $modes)) {
      // This stop is a light rail stop
    }
  }
```

Another type of location is `platform`, which indicates a specific platform inside (typically) a train station. When you encounter this `stop` type, the parent location will typically have a type of `stop`. This can be useful for display purposes, and you can also use this data to look up information about the corresponding stop.

```php
<?php
  // ... Perform request and extract a location

  if ($location['type'] == 'platform') {
    // Great, we have a platform, but what is its stop?

    $parent = $location['parent'];
```

```php
    if ($parent['type'] == 'stop') {
      $stopId = $parent['id'];
    }
  }
```

Finding a Stop by ID

If you know a stop's ID and want to look it up with `stop_finder` for its additional information, you can pass the stop ID in `name_sf`. additionally, if you set the `type_sf` value to `stop` (instead of `any`, as in previous examples), it will ensure only locations with a type of `stop` are returned.

In the following example, we have a stop ID of `10101331`, but want to find out more information:

```php
<?php
  $params = array(
    'outputFormat'    => 'rapidJSON',
    'type_sf'         => 'stop',
    'name_sf'         => '10101331',
    'coordOutputFormat' => 'EPSG:4326',
    'anyMaxSizeHitList' => 10,
    'TfNSWSF'         => 'true'
  );

  // Perform the request and handle the response
```

This query will subsequently return information about the domestic airport station.

Extracting Address Parts

The `stop_finder` API call can also return information about places of interest or specific Street addresses. In other words, users don't need to search for a specific stop.

In this instance, just update the `name_sf` request parameter accordingly.

Also, consider that many street names are used multiple suburbs, so it is especially important in this case to ask the user which of the matching locations they want. For example, a search of `1 Main Road` may return 10 or 20 locations.

When prompting a user which location they want, it's important to disambiguate between the results. You can use the `name` value to provide the suburb name, or you can manually build up the street address using `type`, `streetName`, `buildingNumber`, `parent`, and any other fields you think may be useful.

```php
<?php
  $searchQuery = '1 Main Road';
```

```php
// ... Perform request and handle response

foreach ($locations as $location) {
  if ($location['type'] == 'singlehouse') {
    // This result refers to a house

    $number = $location['buildingNumber'];
    $street = $location['streetName'];

    $parent = $location['parent'];

    if ($parent['type'] == 'locality') {
      $suburb = $parent['name'];
    }
  }
}
```

Note: Postcodes are not returned in calls to `stop_finder`.

Coordinates

Each returned location has a coordinate associated with it, which is useful for displaying on a map, or for finding nearby places of interest using the `coord` API call.

```php
<?php
// ... Perform request and handle response

foreach ($locations as $location) {
  $coord = $location['coord'];

  $latitude  = $coord[0];
  $longitude = $coord[1];

  // Use the coordinate.
}
```

As noted previously, remember to include a request parameter called `coordOutputFormat` with a value of `EPSG:4326` in order to retrieve coordinates in the standard format of many modern programming languages.

1.3. Trip Planner API

The Trip Planner API (also known as `trip`) is used to find available journeys between two locations.

The primary use-case for this API call is when a user wants to travel from one location to another at a certain time, but isn't sure of how to do so. Depending on the locations involved,

the `trip` API call will suggest a number of alternatives, each of which has different trade-offs (such as amount of walking or cost of trip).

The general algorithm for using `trip` is as follows:

1. **Determine origin and destination locations**. To do so, use the `stop_finder` API call. Note that this location may be determined well ahead of time (for example, if you offered"save favourite stops" type functionality).
2. **Determine departure or arrival time**. Generally speaking, you should assume that historical trip planning data is not available from the API. In other words, the search time shouldn't be in the past. Additionally, the distance into the future that is available may be variable. As a rule of thumb, assume you can plan trips no further than two weeks in advance.
3. **Perform a request to `trip`**.
4. **Handle response and display output**.

Data Returned in Trip Planner API

A call to `trip` returns a list of suggested journeys, each of which contains a large quantity of data. This includes:

- Information about each leg of the trip, including public transportation, walking - and in some cases - driving legs.
- Opal fare calculation information.
- Instructions for navigating between legs (such as paths, ramps, escalators and stairs that need to be traversed).
- Service alert information. This is the same information that is returned with the `add_info` API call, but related to the specific journey.
- Map display information. That is, a list of coordinates that represents the path the vehicle takes (or in the case of a walking leg, the path to walk).
- A list of every stop a vehicle makes while travelling from the leg's starting stop to its finishing stop.
- Wheelchair accessibility information.

Example: Performing a Trip Planner Request

To perform a request to the Trip Planner API, you need to know the origin and destination locations ahead of time. If you're using a coordinate as one of the locations, this can be used directly (see the "Directions from Current Location" example later this in this section). Otherwise, use the `stop_finder` to find the stop.

The following shows an example set of request parameters, using `Domestic Airport Station` and `Manly Wharf` (these two locations will be used for most of the examples in this section.

```php
<?php
  $apiEndpoint = 'https://api.transport.nsw.gov.au/v1/tp/';
  $apiCall    = 'trip';
```

```php
// Input parameters for the search
$when        = time();
$origin      = '10101331', // Domestic Airport
$destination = '10102027', // Manly Wharf

// Build the request parameters
$params = array(
    'outputFormat'      => 'rapidJSON',
    'coordOutputFormat' => 'EPSG:4326',
    'depArrMacro'       => 'dep',
    'itdDate'           => date('Ymd', $when),
    'itdTime'           => date('Hi', $when),
    'type_origin'       => 'stop',
    'name_origin'       => $origin,
    'type_destination'  => 'stop',
    'name_destination'  => $destination,
    'TfNSWTR'           => 'true'
);
```

There are four primary inputs that are necessary for a Trip Planner API request:

1. Origin location
2. Destination location
3. Date/time of search (this is set to "now" in this example)
4. Whether the query is "depart after" or "arrive before", (indicated by either `dep` or `arr` in the `depArrMacro` request parameter).

In this example, the native PHP `date()` function is used to format the timestamp into a valid date and time for the request (populated into `itdDate` and `itdTime` respectively).

The following code continues from the above, now performing the request and building a summary of results.

```php
<?php
$url = $apiEndpoint . $apiCall . '?' . http_build_query($params);

// Perform the request and build the JSON response data
$response = file_get_contents($url);
$json = json_decode($response, true);

// Returned journeys are in the `journeys` element
$journeys = $json['journeys'];

// Loop over each journey and display a summary
foreach ($journeys as $journey) {

    // Information about each leg is in `legs`.
```

```php
// This is sorted by leg sequence.
$legs = $journey['legs'];

// Fare information is in `fare`.
// See the later example for how to use this data.
$fares = $journey['fare'];

// The duration of each leg will be tallied in order
// to determine the total duration of the trip.
$totalDuration = 0;

// This array will hold a summary of route types for the journey
$summary = array();

$legNumber = 0;

foreach ($legs as $leg) {
    // Find the leg duration and add it to the total
    $totalDuration += $leg['duration'];

    // Extract origin and destination
    $origin       = $leg['origin'];
    $destination   = $leg['destination'];

    // Determine the trip departure time.
    // This is indicated by the departure time of the first leg.
    if ($legNumber == 0) {
        $depart = strtotime($origin['departureTimePlanned']);
    }

    // Determine the trip arrival time.
    // This is indicated by the arrival time of the final leg.
    if ($legNumber == count($legs) - 1) {
        $arrive = strtotime($origin['departureTimePlanned']);
    }

    // Extract the route information and determine the type of transport.
    $transportation = $leg['transportation'];

    $routeType = $transportation['product']['class'];

    switch ($routeType) {
        case 1: $summary[] = 'Train'; break;
        case 4: $summary[] = 'Light Rail'; break;
        case 5: $summary[] = 'Bus'; break;
        case 7: $summary[] = 'Coach'; break;
```

```
        case 9: $summary[] = 'Ferry'; break;
        case 11: $summary[] = 'School Bus'; break;
        case 99: $summary[] = 'Walk'; break;
        case 100: $summary[] = 'Walk'; break;
      }

      $legNumber += 1;
    }

    $minutes = $totalDuration / 60;

    // Output the departure, arrival and duration
    echo date("r", $depart) . " - " . date("r", $arrive) . " (" . $minutes . " mins)\n";

    // Output the summary of leg types used for the trip
    echo join(" -> ", $summary) . "\n\n";
  }
```

This example will produce output similar to the following:

```
Mon, 17 Oct 2016 15:14:00 +1100 - Mon, 17 Oct 2016 15:40:00
+1100 (39 mins)
Train -> Ferry

Mon, 17 Oct 2016 15:14:00 +1100 - Mon, 17 Oct 2016 15:43:00
+1100 (61 mins)
Train -> Walk -> Bus

Mon, 17 Oct 2016 15:20:00 +1100 - Mon, 17 Oct 2016 16:20:00
+1100 (55 mins)
Train -> Train -> Walk -> Bus -> School Bus -> Bus
```

Note that this example does not use real-time data. The next example shows how to make use of real-time data.

Example: Real-Time Data

By including the `TfNSWTR` parameter with a value of `true`, you enable the ability for real-time data to be returned.

Real-time data is contained within the `departureTimeEstimated` and `arrivalTimeEstimated` fields of a stop sequence element.

Each public transport leg (i.e. not walking or driving) includes an element called `stopSequence`. This contains an ordered list of stops where the vehicle stops. This includes the first and last stop, which are also available from the `origin` and `destination` values within a journey leg.

Note: The origin stop will typically only include departure times, while the destination stop will typically only include arrival times. Stops in-between may include both arrival and departure times, since they may differ if a vehicle has a layover at a stop.

The following example shows how to read the scheduled and estimated arrival from a journey leg. The same can be done for the arrival time using the `arrivalTimeEstimated` and `arrivalTimePlanned` values.

```php
<?php
  // ... Perform request and process the results into $journeys

  $journey = $journeys[0];

  // Each leg has its own origin and destination
  foreach ($journey['legs'] as $leg) {

    $origin     = $leg['origin'];
    $destination = $leg['destination'];

    // Extract the scheduled and real-time estimate
    $departureIsRealtime = false;
    $departureEstimate  = $origin['departureTimeEstimated'];
    $departurePlanned   = $origin['departureTimePlanned'];

    if (strlen($departureEstimate) > 0) {
      $departureIsRealtime = true;
      $departure = strtotime($departureEstimate);
    }
    else {
      $departure = strtotime($departurePlanned);
    }

    // ... Do something with the estimate
  }
```

Note: The method used here to ensure the departure time is somewhat crude. You may want perform your own validation to ensure it's a useable timestamp, or whether to fall back to the scheduled time.

Example: Arrive By

As noted in the above example, you can either search for trips either "departing after" or "arriving before" the given time.

- **Depart After**: If the user wants to leave right now, you would use a "depart after" query. All returned journeys would depart no earlier than this time. The results are sorted by their departure time (earliest first).
- **Arrive Before**: If the user wants to arrive at their destination by, say, 6 PM, you would use an "arrive before" query. All returned journeys would arrive no later than this time. The results are sorted by their arrival time (latest first - that is, closest to the arrival time first).

To make use of this in `trip`, it's just a matter of using `arr` in the `depArrMacro` request parameter.

```php
<?php

  // Timestamp for today at 6 PM
  $when = mktime(18, 0, 0, date("n"), date("j"), date("Y"));

  // Build the request parameters
  $params = array(

    ...
    'depArrMacro'    => 'arr',
    'itdDate'        => date('Ymd', $when),
    'itdTime'        => date('Hi', $when),

    ...
  );
```

Performing the request and handling the response data is identical to before, except that returned journeys are sorted in reverse order.

Example: Directions from Current Location

If you're developing a mobile app that uses the `trip` API call, a common use-case is to display directions from the user's current location (determined using GPS functionality on their device).

In this instance, you would use this coordinate as the origin location (the `name_origin` field). Additionally, specify the `type_origin` value as `coord`.

Note: Of course, you can also use a coordinate as the destination location - just use `name_destination` *and* `type_destination` *instead.*

The format for specifying a coordinate is `LONGITUDE:LATITUDE:EPSG:4326`. For example, if you have determined the user's location is `(-33.884080, 151.206290)`, then the `name_origin` value would be `151.206290:-33.884080:EPSG:4326`.

```php
<?php
  // User's search query
  $latitude  = -33.884080;
  $longitude = 151.206290;

  // Create the coordinate string. %01.6f means a floating-point
```

```php
   // number with up to 6 numbers after the decimal.
   $coord = sprintf('%01.6f:%01.6f:EPSG:4326', $longitude, $latitude);

   // Build the request parameters
   $params = array(
      …
      'name_origin' => $coord,
      'type_origin' => 'coord'
      …
   );
```

It is highly likely that for such a request that the first leg of the returned journeys is a walking leg, since the origin location isn't necessarily a stop or station.

Example: Calculating Trip Cost

The `trip` API call includes - when available - information about the cost of each returned journey. Since calculating an Opal fare is quite complex (there are many factors that determine a trip cost), using the data provided from this API call is very useful.

The following code shows how to extract the fare information from a journey. This example only keeps `ADULT` fares and discards the remaining fare information.

```php
<?php
   // … Perform request and process the results into $journeys

   $journey = $journeys[0];

   // Assume the user wants to see fares for only a particular fare level
   $fareType = 'ADULT';


   $fares = $journey['fare']['tickets'];

   // This will contain the summary of the total journey fare
   $journeyFare = null;

   // This will contain fares on a per-leg basis
   $legFares = array();

   foreach ($fares as $fare) {
      // Ensure this fare is for the chosen fare level
      if ($fare['person'] != $fareType) {
         continue;
      }

      $properties = $fare['properties'];
```

```php
    // Only the total summary contains `evaluationTicket` element
    if (array_key_exists('evaluationTicket', $properties)) {
        $journeyFare = $fare;
    }
    else {
        // Add this to the list of leg-specific fares
        $legFares[] = $fare;
    }
}
```

At this point, a summary of the total journey cost is in `$journeyFare`. A breakdown of the cost of each leg is available in `$legFares`.

In order to calculate and display the total cost of a journey, you need to use the `evaluationTicket` field mentioned above. It is used to determine what the total figure actually means. For example, if all legs in a journey are either TfNSW vehicles (ignoring walking legs, which are always free), then the returned fare will be the total cost of travel (a value of `nswFareEnabled`).

However, if a trip contains one TfNSW vehicle and a private ferry, then the value will be `nswFarePartiallyEnabled`, since the cost of the private ferry cannot be determined.

If the trip contains only a private ferry (in other words, so no legs have a calculated fare), then this value will be `nswFareNotAvailable`.

The `nswFareNotEnabled` was primarily used during the Opal rollout. It indicates that even though a vehicle may be part of the TfNSW network, a fare cannot be calculated. If one fare can be calculated and the other doesn't have Opal available, then `nswFarePartiallyEnabled` will also be used.

The following code demonstrates how to interpret `evaluationTicket` and to calculate and display the total journey fare.

```php
<?php
    // ... Continuing from above

    // Contains the total pricing and evaluationTicket value
    $properties = $journeyFare['properties'];

    $status = $properties['evaluationTicket'];

    // This is the cost of the trip, not including additional fees
    $tariff = $journeyFare['priceBrutto'];

    // This will typically be zero, but is used when
    // accessing the airport stations
    $stationAccessFee = $properties['priceStationAccessFee'];
```

```php
    // This total includes the station access fee, and is
    // subject to the evaluationTicket value. This should
    // be the sum of `$tariff` and `$stationAccessFee`.
    $total = $properties['priceTotalFare'];

    // A human-readable formatted title for the fare level
    $cardTitle = $properties['riderCategoryName'];


    echo $cardTitle . ":\n";
    echo "Tariff: $" . $tariff . "\n";
    echo "Station Access Fee: $" . $stationAccessFee . "\n";
    echo "Total: $" . $total . " (including station access fee)\n";

    switch ($status) {
        case 'nswFareEnabled':
            echo "This price is the full journey cost.\n";
            break;

        case 'nswFarePartiallyEnabled':
            echo "This price does not account for all journey legs.\n";
            break;

        case 'nswFareNotEnabled':
            echo "Opal is not available for this journey.\n";
            break;

        case 'nswFareNotAvailable':
            echo "Unable to determine cost for any legs in this journey.\n";
            break;
    }
```

This code will produce output similar to the following:

```
Adult:
Station Access Fee: $13.40
Total: $20.94 (including station access fee)
This price is the full journey cost.

Adult:
Station Access Fee: $13.40
Total: $19.50 (including station access fee)
This price does not account for all journey legs.
```

In addition to showing the entire journey cost, it is possible to break down the cost on a per-leg basis. The earlier code in this section built an array called $legFares. Each element in this array contains leg-specific information. Since a single part of a total fare may cover multiple legs, each fare contains a starting leg number and a finishing leg number.

For example, if a journey has 2 legs and it is covered by a single fare, then there will be one entry in $legFares, with a fromLeg value of 0 and a toLeg value of 1 (the fromLeg and toLeg values start from 0).

The tariff, access fee and total can be extracted in the same way as the journey fare.

Example: Stopping Pattern

Every public transport leg that is returned includes the stopping pattern, which is the list of stops that the vehicle visits on its way from the origin to the destination.

The stopping pattern can be retrieved from the stopSequence element within a journey leg. The list of stops is in stopping order, which means the first element in this array is the same as the leg's origin element, while the last element in this array is the same as the leg's destination element.

The first element typically only includes departure time information, while the final element only includes arrival time information. Stops in-between include both arrival and departure information (a vehicle may wait at a stop for a pre-determined amount of time before it continues the trip).

```php
// ... Perform request and process the results into $journeys

$journey = $journeys[0];

$legNumber = 1;
$legs = $journey['legs'];

foreach ($legs as $leg) {
    $transportation = $leg['transportation'];

    echo "Leg " . $legNumber . ". " . $transportation['number'] . "\n";

    $stopSequence = $leg['stopSequence'];

    $stopNumber = 1;

    foreach ($stopSequence as $stop) {
        echo $stopNumber . ". " . $stop['disassembledName'] . "\n";

        $stopNumber += 1;
    }

    echo "\n";
```

```
    $legNumber += 1;
  }
```

An example of the output this script might produce is as follows:

```
Leg 1. T2 Airport, Inner West & South Line
1. Domestic Airport Station, Platform 1
2. Mascot Station, Platform 1
3. Green Square Station, Platform 1
4. Central Station, Platform 21
5. Museum Station, Platform 1
6. St James Station, Platform 1
7. Circular Quay Station, Platform 1

Leg 2. Manly Fast Ferry
1. Circular Quay, Wharf 6
2. Manly Wharf
```

You can access departure/arrival times in the same way as shown earlier in this chapter. This includes both scheduled times and real-time estimates (when available).

Example: Journey Coordinates

Every journey leg returned in a request to `trip` includes coordinates of the path taken (whether that is a train, bus, or for a walking leg). This allows you to plot the path taken to a map.

Coordinates are returned in a leg's `coords` element as a list of latitude/longitude pairs in sequential order.

```php
<?php
  // ... Perform request and process the results into $journeys

  $journey = $journeys[0];

  $legNumber = 1;
  $legs = $journey['legs'];

  foreach ($legs as $leg) {
    $coords = $leg['coords'];

    foreach ($coords as $coord) {
      $lat = $coord[0];
      $lon = $coord[1];

      // Do something with the coordinate here
```

```
    }
  }
```

For example, if you wanted to plot a path onto Google Maps, you could combine PHP and JavaScript to something similar to the following:

```php
<?php
  $journey = $journeys[0];

  $legs = $journey['legs'];

  foreach ($legs as $leg) {

    // Build up the list of coordinates in the format
    // required for Google Maps JavaScript API.
    $coords = array();

    foreaech ($leg['coords'] as $coord) {
      $coords[] = array(
        'lat' => $coord[0],
        'lng' => $coord[1]
      );
    }
?>

// Output JavaScript code while looping over response legs

// This is a shortcut to output a PHP array to JavaScript code
var coords = <?php echo json_encode($coords) ?>;

var path = new google.maps.Polyline({
  path: coords,
  geodesic: true,
  strokeColor: '#FF0000', // Red line
  strokeOpacity: 1.0,
  strokeWeight: 2
});

// This assumes `map` has been configured
// previously to a `google.maps.Map` object.
path.setMap(map);

<?php
  } // Closes the legs loop
```

You can combine this technique with adding stops to the map also. For example, you might want to display the origin and destination of each leg on the map - or even all stops for a leg.

The following example shows how to turn the origin and destination locations into markers on a map built with the Google Maps JavaScript API.

```php
<?php
  $journey = $journeys[0];

  $legs = $journey['legs'];

  foreach ($legs as $leg) {

    $origin     = $leg['origin'];
    $destination = $leg['destination'];

    $originCoord    = array(
      'lat' => $origin['coord'][0],
      'lng' => $origin['coord'][1]
    );

    $destinationCoord = array(
      'lat' => $destination['coord'][0],
      'lng' => $destination['coord'][1]
    );
?>

// Output JavaScript code while looping over response legs

// This is a shortcut to output a PHP object to JavaScript code
var origin = <?php echo json_encode($originCoord) ?>;

var originMarker = new google.maps.Marker({
  position: origin,
  map: map,
  title: <?php echo json_encode($origin['name']) ?>
});

var destination = <?php echo json_encode($destinationCoord) ?>;

var destinationMarker = new google.maps.Marker({
  position: destination,
  map: map,
  title: <?php echo json_encode($destination['name']) ?>
});
```

```php
<?php
  } // Closes the legs loop
```

Example: Service Alerts

One of the API calls in the TfNSW Trip Planner API is `add_info`, which is used to retrieve service alerts. In addition to this API call, service alert information is also returned with journeys returned with `trip`. This makes it extremely easy to gather relevant, up-to-date service alert information when displaying a list of travel options.

Service alerts are returned on a per-leg basis, not on a per-journey basis, since alerts typically affect a specific route, stop, or mode of transport.

The following example shows how to read alerts from each leg:

```php
<?php
  $journey = $journeys[0];

  $legs = $journey['legs'];

  foreach ($legs as $leg) {

    // Per-leg alerts are stored in `infos`
    $alerts = $leg['infos'];

    foreach ($alerts as $alert) {
      // Extract and use the alert info
      echo $alert['subtitle'] . "\n";
    }

  }
```

In addition to service alert information being returned in the **infos** element of a journey leg, at times there are also useful travel hints available in the `hints` element.

```php
<?php
    $journey = $journeys[0];

    $legs = $journey['legs'];

    foreach ($legs as $leg) {

        // Per-leg hints are stored in `hints`
        $hints = $leg['hints'];

        foreach ($hints as $hint) {
            // Extract and use the hint text
            echo $hint['infoText'] . "\n";
```

```
        }
    }
```

Example: Wheelchair Accessibility

It is possible to determine the wheelchair accessibility status of returned journeys (specifically, the legs within each journey).

For a given journey leg, there are three accessibility-related things that need to be considered:

1. Accessibility at the origin stop.
2. Accessibility on the vehicle.
3. Accessibility at the destination stop.

Since the origin and destination stop data is the same type, it can be checked in the same way. Additionally, their data structure is the same as all records in the stopping pattern (`stopSequence`), meaning you can check the accessibility of any stop on a journey leg if necessary.

The wheelchair accessibility information for a stop is stored within the `properties` element, in `WheelchairAccess`. Note that due to legacy reasons, this value is a JSON `"string"` type, not a Boolean, so you must test for the string `"true"`.

```php
<?php
  $origin = $leg['origin'];
  $originWC = $origin['properties']['WheelchairAccess']
  $originIsAccessible = $originWC == "true";

  if ($originIsAccessible) {
    // Yes, wheelchair accessible stop
  }
```

To determine wheelchair accessibility for the vehicle, there are two values to check: `PlanLowFloorVehicle` and `PlanWheelChairAccess`. Both are contained within the `properties` element for the leg.

Note that due to legacy reasons, this value is a JSON `"string"` type, not a Boolean, so you must test for the string `"1"`. Additionally, note that the capitalised `C` in `WheelChair`, unlike when checking for a stop's accessibility.

```php
<?php
  $properties        = $leg['properties'];
  $lowFloorVehicle    = $properties['PlanLowFloorVehicle'] == "1";
  $wheelchairAccessible = $properties['PlanWheelChairAccess'] == "1";

  if ($lowFloorVehicle) {
    // This is a low floor vehicle
  }
```

```php
if ($wheelchairAccessible) {
    // This vehicle is wheelchair accessible
}
```

The following example shows how you can use all of these values together.

```php
// ... Perform request and process the results into $journeys

$journey = $journeys[0];

$legNumber = 1;
$legs = $journey['legs'];

foreach ($legs as $leg) {
    $properties        = $leg['properties'];
    $lowFloorVehicle     = $properties['PlanLowFloorVehicle'] == "1";
    $wheelchairAccessible = $properties['PlanWheelChairAccess'] == "1";

    $transportation = $leg['transportation'];

    echo sprintf("Leg %d. %s\n", $legNumber, $transportation['number']);
    echo "Low Floor Vehicle: " . ($lowFloorVehicle ? "yes" : "no") . "\n";
    echo sprintf(
        "Wheelchair Accessible Vehicle: %s\n\n",
        $wheelchairAccessible ? "yes" : "no")
    );

    $origin      = $leg['origin'];
    $originWC     = $origin['properties']['WheelchairAccess'];

    $destination   = $leg['destination'];
    $destinationWC = $destination['properties']['WheelchairAccess'];

    $originIsAccessible     = $originWC == "true";
    $destinationIsAccessible = $destinationWC == "true";

    echo "From: " . $origin['disassembledName'] . "\n";
    echo sprintf(
        "Accessible Stop: %s\n\n",
        $originIsAccessible ? "yes" : "no")
    );

    echo "To: " . $destination['disassembledName'] . "\n";
    echo sprintf(
        "Accessible Stop: %s\n\n",
        $destinationIsAccessible ? "yes" : "no")
```

```php
    );

    $legNumber += 1;
  }
```

Example: Querying For Only Wheelchair-Accessible Journeys

The above examples show how to check if a returned journey is wheelchair accessible, but for a user that requires a wheelchair-accessible trip, they probably want to avoid having to make this determination themselves.

Instead, by setting the `wheelchair` request value to `on`, you can ensure that only trips that are wheelchair-accessible will be returned.

```php
<?php
  // Build the request parameters
  $params = array(

    ...
    'wheelchair' => 'on'

    ...
  );
```

1.4.   Departure API

The Departure API (also known as `departure_mon`) is used to find upcoming departures at a given stop.

The primary use-case for this API call is to display a "departure board" type interface where you can quickly see which services are about to depart and where they are going to.

Example: Listing All Upcoming Departures

The following example shows how to request all upcoming departures for Domestic Airport Station (specified by stop ID `10101331`).

It uses "now" (indicated by `time()` in PHP) as the search date. The format string of `Ymd` in PHP will create a `YYYYMMDD` formatted date. Likewise, a format string of `Hi` creates a time in `HHMM` 24-hour format.

The comments in the code indicate how to use the returned data.

```php
<?php
  $apiEndpoint = 'https://api.transport.nsw.gov.au/v1/tp/';
  $apiCall    = 'departure_mon';

  // Set the location and time parameters
  $when = time();   // Now
```

```php
$stop = '10101331'; // Domestic Airport Station

// Build the request parameters
$params = array(
  'outputFormat'     => 'rapidJSON',
  'coordOutputFormat' => 'EPSG:4326',
  'mode'           => 'direct',
  'type_dm'        => 'stop',
  'name_dm'        => $stop,
  'depArrMacro'    => 'dep',
  'itdDate'        => date('Ymd', $when),
  'itdTime'        => date('Hi', $when),
  'TfNSWDM'        => 'true'
);

$url = $apiEndpoint . $apiCall . '?' . http_build_query($params);

// Perform the request and build the JSON response data
$response = file_get_contents($url);
$json = json_decode($response, true);

$stopEvents = $json['stopEvents'];

// Loop over returned stop events
foreach ($stopEvents as $stopEvent) {

  // Extract the route information
  $transportation = $stopEvent['transportation'];
  $routeNumber    = $transportation['number'];
  $destination    = $transportation['destination']['name'];

  // In the case of a train, the location includes platform information
  $location = $stopEvent['location'];

  // Determine how many minutes until departure
  $time     = strtotime($stopEvent['departureTimePlanned']);
  $countdown = $time - time();
  $minutes   = round($countdown / 60);

  // Output the stop event with a countdown timer
  echo $minutes . "m from " . $location['disassembledName'] . "\n";
  echo $routeNumber . " to " . $destination . "\n\n";
}
```

This example creates a crude departure board that lists all returned departure times. Additionally, it calculates and displays how many minutes until the departure is and includes that information.

Example: Departures from a Specific Platform

In the above example, all departures from Domestic Airport Station are included. If you only wanted to list departures from platform 1 of this station, you would change the value `name_dm` to the ID for the platform, and then set `nameKey_dm` to `$USEPOINT$`.

```php
<?php
  // ...

  $stop = '202091'; // Domestic Airport Station Platform 1

  // Build the request parameters
  $params = array(
    'outputFormat'     => 'rapidJSON',
    'coordOutputFormat' => 'EPSG:4326',
    'mode'             => 'direct',
    'type_dm'          => 'stop',
    'name_dm'          => $stop,
    'nameKey_dm'       => '$USEPOINT$',
    'depArrMacro'      => 'dep',
    'itdDate'          => date('Ymd', $when),
    'itdTime'          => date('Hi', $when),
    'TfNSWDM'          => 'true'
  );

  // ...
```

If you were to output the results of this query as above, you would notice the returned results are all from the same platform.

1.5. Service Alert API

The Service Alert API (also known as `add_info)` is used to retrieve service alert information.

The primary use-case for this API call is to determine any outages, delays, or other information that is pertinent to users of the public transportation network.

Example: Retrieving Active Alerts For Today

The following example shows how to retrieve active alerts for the current day.

This is achieved by specifying the `filterDateValid` value with a timestamp formatted in `DD-MM-YYYY` format. Additionally, by specifying the `filterPublicationStatus` with a

value of `current`, historical alerts are not returned (i.e. alerts that are no longer active), making the response data smaller.

```php
<?php
  $apiEndpoint = 'https://api.transport.nsw.gov.au/v1/tp/';
  $apiCall    = 'add_info';

  $when = time(); // Sets the date filter date to "now"

  // Build the request parameters
  $params = array(
    'outputFormat'         => 'rapidJSON',
    'coordOutputFormat'    => 'EPSG:4326',
    'filterDateValid'      => date('d-m-Y', $when),
    'filterPublicationStatus' => 'current'
  );

  $url = $apiEndpoint . $apiCall . '?' . http_build_query($params);

  // Perform the request and build the JSON response data
  $response = file_get_contents($url);
  $json    = json_decode($response, true);

  $infos   = $json['infos'];
  $current = $infos['current'];

  // Loop over found messages
  foreach ($current as $message) {

    // Extract heading and URL for alert
    $heading = $message['subtitle'];
    $url    = $message['url'];

    // Determine how long since alert modified
    $timestamps = $message['timestamps'];
    $modified   = strtotime($timestamps['lastModification']);
    $age      = time() - $modified;

    $hr  = round($age / 3600);
    $min = round($age / 60) % 60;

    echo strtoupper($heading) . "\n";
    echo sprintf("Last Modified: %s (%dh %dm ago)\n", date('r', $modified), $hr, $min);
    echo "More Info: " . $url . "\n";

    // Extract info about affected routes and stops
```

```php
    $affected = $message['affected'];
    $lines   = $affected['lines'];
    $stops   = $affected['stops'];

    echo "Affected Stops: " . count($stops) . "\n";
    echo "Affected Lines: " . count($lines) . "\n\n";
  }
```

This example shows how to extract various data from the alert. Refer to the Swagger documentation for a comprehensive list of all data available.

For instance, the stops and lines that are impacted by the alert are included also. This includes information about route / stop number and titles.

Example: Retrieving Alerts for a Specific Stop

It is possible with `add_info` to retrieve alerts only for a particular stop by specifying the `itdLPxx_selStop` request parameter.

The following example shows how to retrieve today's current alerts that are relevant to Central Station (which has a stop ID of `10111010`).

```php
<?php
  $when = time();

  // Build the request parameters
  $params = array(
    'outputFormat'         => 'rapidJSON',
    'coordOutputFormat'    => 'EPSG:4326',
    'filterDateValid'      => date('d-m-Y', $when),
    'filterPublicationStatus' => 'current',
    'itdLPxx_selStop'      => '10111010'
  );

  // Perform request and process response
```

1.6. Coordinate Request API

The Coordinate Request API (also known as `coord`) is used to find places of interest, stops or Opal resellers near a given location (specified using a latitude/longitude coordinate).

The primary use-case for this API call is for displaying such locations on a map when users are looking for particular locations. While the `stop_finder` API call finds locations when the user knows what to search for, `coord` suggests locations based on an input coordinate.

In addition to specifying a coordinate from which to base the search, the request must also include a radius (in metres), which restricts how far away from the source coordinate returned locations are.

Increasing the radius will exponentially increase the number of the returned results, so it's important to be mindful of your system resources. Increasing a search radius, say from 1000m to 2000m might overload a user's web browser or mobile device, since the number of returned locations could increase from, say, 100 to 10000.

Example: Finding Opal Ticket Resellers

Consider the scenario where you want to display a map of Opal resellers near the user's location on their iPhone or Android device.

The algorithm you would follow for doing so would be:

1. Determine the user's current location (latitude and longitude).
2. Determine how far you want to search from that location (the search radius). This may be based on the zoom level of the map being displayed to the user.
3. Perform a `coord` with a `type_1` value of `GIS_POINT` and the radius in the `radius_1` request parameter.
4. Parse results and display on map.

To specify the origin coordinate of the search, you must specify the `coord` request value. This uses a format of `LATITUDE:LONGITUDE:EPSG:4326`.

For example, if you have determined the user's location is `(-33.884080, 151.206290)`, then the `coord` value would be the string `151.206290:-33.884080:EPSG:4326`.

Note: Be aware that coordinates are commonly written with latitude first, but the ordering is reversed for TfNSW API request parameters such as `coord`, meaning longitude is first.

```php
<?php
  // User's location
  $latitude  = -33.884080;
  $longitude = 151.206290;

  $radius = 1000; // Metres

  $coord = sprintf('%01.6f:%01.6f:EPSG:4326', $longitude, $latitude);

  $apiEndpoint = 'https://api.transport.nsw.gov.au/v1/tp/';
  $apiCall    = 'coord';

  // Build the request parameters
  $params = array(
    'outputFormat'     => 'rapidJSON',
    'coord'           => $coord,
    'coordOutputFormat' => 'EPSG:4326',
```

```php
    'type_1'        => 'GIS_POINT',
    'radius_1'       => $radius,
    'inclFilter'     => 1,

    // Flag to ensure only Opal resellers are returned
    'inclDrawClasses_1' => 74
);

$url = $apiEndpoint . $apiCall . '?' . http_build_query($params);

// Perform the request and build the JSON response data
$response = file_get_contents($url);
$json = json_decode($response, true);

// Extract locations from response
$locations = $json['locations'];

foreach ($locations as $location) {
    // Read the name, coordinate and distance from the location

    $name = $location['name'];

    $coord = $location['coord'];
    $lat   = $coord[0];
    $lon   = $coord[1];

    $distance = $location['properties']['distance'];

    // Output information about the location
    echo $distance . "m away: " . $name . "\n";

    // Demonstrates how to use the coordinate.
    // You may want to display on a map instead
    echo sprintf(
        "https://maps.google.com/?q=%01.6f,%01.6f\n\n",
        $lat,
        $lon
    );
}
```