

---

# **Ichnaea**

***Release 1.5***

**Mar 05, 2022**



---

## Contents

---

<b>1</b>	<b>About the name</b>	<b>3</b>
<b>2</b>	<b>Table of contents</b>	<b>5</b>
<b>3</b>	<b>Development</b>	<b>75</b>
<b>4</b>	<b>License</b>	<b>77</b>
	<b>Index</b>	<b>79</b>



Ichnaea is a service to provide geolocation coordinates from other sources of data (Bluetooth, cell or WiFi networks, GeoIP, etc.). It is using both *Cell-ID* and Wi-Fi based positioning (*WPS*) approaches.

Mozilla hosts an instance of this service, called the *Mozilla Location Service (MLS)*.

You can interact with the service in two ways:

- If you know where you are, submit information about the radio environment to the service to increase its quality.
- or locate yourself, based on the radio environment around you.



# CHAPTER 1

---

## About the name

---

In Greek mythology, Ichnaea (Iknaia) means “the tracker”.





## 2.1 Using the Service

You can either use the service directly via the HTTP service APIs or you can use one of the existing applications or libraries.

### 2.1.1 HTTP API

You can use the *Services API* directly via a command line client like curl or via any programming language allowing you to do HTTP calls.

### 2.1.2 Applications / Libraries

A number of applications and libraries with different capabilities exist which allow you to interact with the service. Please have a look at the [Software listing](#) on the Mozilla Wiki to learn more.

## 2.2 Services API

The service APIs accept data submission for geolocation stumbling as well as reporting a location based on IP addresses, cell or WiFi networks.

New client developments should use the *Region*, *Geolocate* or *Geosubmit Version 2* APIs.

### 2.2.1 API Access Keys

You can anonymously submit data to the service without an API key via any of the submission APIs.

You must identify your client to the service using an API key when using one of the *Region*, *Geolocate* or *Transfer (private)* APIs.

If you want or need to specify an API key, you need to provide it as a query argument in the request URI in the form:

```
https://location.services.mozilla.com/<API>?key=<API_KEY>
```

Each API key can be rate limited per calendar day, but the default is to allow an unlimited number of requests per day.

## 2.2.2 Errors

Each of the supported APIs can return specific error responses. In addition there are some general error responses.

### Invalid API Key

If an API key was required but none or no valid key was given, the service responds with a *keyInvalid* message with a HTTP 400 error code:

```
{
  "error": {
    "errors": [{
      "domain": "usageLimits",
      "reason": "keyInvalid",
      "message": "Missing or invalid API key."
    }],
    "code": 400,
    "message": "Invalid API key"
  }
}
```

### API Key Limit

API keys can optionally be rate limited. If the limit for a specific API key is exceeded, the service responds with a *dailyLimitExceeded* message with a HTTP 403 error code:

```
{
  "error": {
    "errors": [{
      "domain": "usageLimits",
      "reason": "dailyLimitExceeded",
      "message": "You have exceeded your daily limit."
    }],
    "code": 403,
    "message": "You have exceeded your daily limit."
  }
}
```

### Parse Error

If the client sends a malformed request, typically sending malformed or invalid JSON, the service will respond with a *parseError* message with a HTTP 400 error code:

```
{
  "error": {
    "errors": [{
      "domain": "global",
      "reason": "parseError",
      "message": "Parse Error"
    }],
    "code": 400,
    "message": "Parse Error"
  }
}
```

## Service Error

If there is a transient service side problem, the service might respond with HTTP 5xx error codes with unspecified HTTP bodies.

This might happen if part of the service is down or unreachable. If you encounter any 5xx responses, you should retry the request at a later time. As a service side problem is unlikely to be resolved immediately, you should wait a couple of minutes before retrying the request for the first time and a couple of hours later if there's still a problem.

## 2.2.3 APIs

Historically the service first offered the custom *Search (Deprecated)* and *Submit (Deprecated)* APIs. Later it was decided to also implement the *Geolocate* API to lessen the burden on clients that want to support multiple location services. As an extension to this the *Geosubmit (Deprecated)* API was added to offer a consistent way to contribute back data to the service. Afterwards the *Region* API was added and *Geosubmit Version 2* superseded its version 1 counterpart. Finally the *Transfer (private)* API was added.

## Geolocate

**Purpose** Determine the current location based on data provided about nearby Bluetooth, cell or WiFi networks and based on the IP address used to access the service.

## Request

Geolocate requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v1/geolocate?key=<API_KEY>
```

This implements almost the same interface as the [Google Maps Geolocation API](#) endpoint, hence referred to as *GLS* or Google Location Service API. Our service implements all of the standard GLS API with a couple of additions.

Geolocate requests are submitted using a POST request with a JSON body.

A minimal example using only WiFi networks:

```
{
  "wifiAccessPoints": [{
    "macAddress": "01:23:45:67:89:ab",
    "signalStrength": -51
  }, {
```

(continues on next page)

(continued from previous page)

```
    "macAddress": "01:23:45:67:89:cd"
  }}
}
```

A minimal example using a cell network:

```
{
  "cellTowers": [{
    "radioType": "wcdma",
    "mobileCountryCode": 208,
    "mobileNetworkCode": 1,
    "locationAreaCode": 2,
    "cellId": 1234567,
    "signalStrength": -60
  }]
}
```

A complete example including all possible fields:

```
{
  "carrier": "Telecom",
  "considerIp": true,
  "homeMobileCountryCode": 208,
  "homeMobileNetworkCode": 1,
  "bluetoothBeacons": [
    {
      "macAddress": "ff:23:45:67:89:ab",
      "age": 2000,
      "name": "beacon",
      "signalStrength": -110
    }
  ],
  "cellTowers": [{
    "radioType": "wcdma",
    "mobileCountryCode": 208,
    "mobileNetworkCode": 1,
    "locationAreaCode": 2,
    "cellId": 1234567,
    "age": 1,
    "psc": 3,
    "signalStrength": -60,
    "timingAdvance": 1
  }],
  "wifiAccessPoints": [{
    "macAddress": "01:23:45:67:89:ab",
    "age": 3,
    "channel": 11,
    "frequency": 2412,
    "signalStrength": -51,
    "signalToNoiseRatio": 13
  }, {
    "macAddress": "01:23:45:67:89:cd"
  }],
  "fallbacks": {
    "lacf": true,
    "ipf": true
  }
}
```

(continues on next page)

(continued from previous page)

}

## Field Definition

All of the fields are optional. Though in order to get a Bluetooth or WiFi based position estimate at least two networks need to be provided and for each the *macAddress* needs to be known. The minimum of two networks is a mandatory privacy restriction for Bluetooth and WiFi based location services.

Cell based position estimates require each cell record to contain at least the five *radioType*, *mobileCountryCode*, *mobileNetworkCode*, *locationAreaCode* and *cellId* values.

Position estimates do get a lot more precise if in addition to these unique identifiers at least *signalStrength* data can be provided for each entry.

Note that all the cell JSON keys use the same names for all radio types, generally using the official GSM name to denote similar concepts, even though the actual client side API's might use different names for each radio type and thus must be mapped to the JSON keys.

## Global Fields

**carrier** The clear text name of the cell carrier / operator.

**considerIp** Should the clients IP address be used to locate it, defaults to true.

**homeMobileCountryCode** The mobile country code stored on the SIM card.

**homeMobileNetworkCode** The mobile network code stored on the SIM card.

**radioType** Same as the *radioType* entry in each cell record. If all the cell entries have the same *radioType*, it can be provided at the top level instead.

## Bluetooth Beacon Fields

**macAddress** The address of the Bluetooth Low Energy (BLE) beacon.

**name** The name of the BLE beacon.

**age** The number of milliseconds since this BLE beacon was last seen.

**signalStrength** The measured signal strength of the BLE beacon in dBm.

## Cell Tower Fields

**radioType** The type of radio network. One of *gsm*, *wcdma* or *lte*. This is a custom extension to the GLS API, which only defines the top-level *radioType* field.

**mobileCountryCode** The mobile country code.

**mobileNetworkCode** The mobile network code.

**locationAreaCode** The location area code for GSM and WCDMA networks. The tracking area code for LTE networks.

**cellId** The cell id or cell identity.

**age** The number of milliseconds since this networks was last detected.

**psc** The primary scrambling code for WCDMA and physical cell id for LTE. This is an addition to the GLS API.

**signalStrength** The signal strength for this cell network, either the RSSI or RSCP.

**timingAdvance** The timing advance value for this cell network.

### WiFi Access Point Fields

---

**Note:** Hidden WiFi networks and those whose SSID (clear text name) ends with the string *\_nomap* must NOT be used for privacy reasons. It is the responsibility of the client code to filter these out.

---

**macAddress** The BSSID of the WiFi network.

**age** The number of milliseconds since this network was last detected.

**channel** The WiFi channel, often 1 - 13 for networks in the 2.4GHz range.

**frequency** The frequency in MHz of the channel over which the client is communicating with the access point. This is an addition to the GLS API and can be used instead of the channel field.

**signalStrength** The received signal strength (RSSI) in dBm.

**signalToNoiseRatio** The current signal to noise ratio measured in dB.

**ssid** The SSID of the Wifi network. Wifi networks with a SSID ending in *\_nomap* must not be collected.

### Fallback Fields

The fallback section is a custom addition to the GLS API.

By default both a GeoIP based position fallback and a fallback based on cell location areas (lac's) are enabled. Simply omit the *fallbacks* section if you want to use the defaults. Change the values to *false* if you want to disable either of the fallbacks.

**lacf** If no exact cell match can be found, fall back from exact cell position estimates to more coarse grained cell location area estimates, rather than going directly to an even worse GeoIP based estimate.

**ipf** If no position can be estimated based on any of the provided data points, fall back to an estimate based on a GeoIP database based on the senders IP address at the time of the query.

### Deviations From GLS API

As mentioned in the specific fields, our API has a couple of extensions.

- The entire Bluetooth section is a custom addition.
- Cell entries allow to specify the *radioType* per cell network instead of globally. This allows for example doing queries with data from multiple active SIM cards where one of them is on a GSM connection while the other uses a WCDMA connection.
- Cell entries take an extra *psc* field.
- The WiFi *macAddress* field takes both upper- and lower-case characters. It also tolerates *;*, *-* or no separator and internally strips them.
- WiFi entries take an extra *frequency* field.

- The *fallbacks* section allows some control over the more coarse grained position sources. If no exact match can be found, these can be used to return a *404 Not Found* rather than a coarse grained estimate with a large accuracy value.
- If either the GeoIP or location area fallbacks were used to determine the response, an additional fallback key will be returned in the response.
- The *considerIp* field has the same purpose as the *fallbacks/ipf* field. It was introduced into the GLS API later on and we continue to support both, with the *fallbacks* section taking precedence.

## Response

A successful response returns a position estimate and an accuracy field. Combined these two describe the center and radius of a circle. The user's true position should be inside the circle with a 95th percentile confidence value. The accuracy is measured in meters.

If the position is to be shown on a map and the returned accuracy is large, it may be advisable to zoom out the map, so that all of the circle can be seen, even if the circle itself is not shown graphically. That way a user should still see his true position on the map and can further zoom in.

If instead the returned position is shown highly zoomed in, the user may just see an arbitrary location that they don't recognize at all. This typically happens when GeoIP based results are returned and the returned position is the center of a city or the center of a region.

A successful response will be:

```
{
  "location": {
    "lat": -22.7539192,
    "lng": -43.4371081
  },
  "accuracy": 100.0
}
```

Should the response be based on a GeoIP estimate:

```
{
  "location": {
    "lat": 51.0,
    "lng": -0.1
  },
  "accuracy": 600000.0,
  "fallback": "ipf"
}
```

Alternatively the fallback field can also state *lacf* for an estimate based on a cell location area.

If no position information could be determined, a HTTP status code 404 will be returned:

```
{
  "error": {
    "errors": [{
      "domain": "geolocation",
      "reason": "notFound",
      "message": "Not found",
    }],
    "code": 404,
    "message": "Not found",
  }
}
```

(continues on next page)

(continued from previous page)

```
}  
}
```

## Region

**Purpose** Determine the current region based on data provided about nearby Bluetooth, cell or WiFi networks and based on the IP address used to access the service.

The responses use region codes and names from the [GENC dataset](#), which is for the most part compatible with the ISO 3166 standard. While the API endpoint and JSON payload refers to *country*, no claim about the political status of any region is made by this service.

## Request

Requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v1/country?key=<API_KEY>
```

This implements the same interface as the [Geolocate](#) API.

The simplest request contains no extra information and simply relies on the IP address to provide a response:

```
{}
```

## Response

A successful response will be:

```
{  
  "country_code": "US",  
  "country_name": "United States"  
}
```

Should the response be based on a GeoIP estimate:

```
{  
  "country_code": "US",  
  "country_name": "United States",  
  "fallback": "ipf"  
}
```

If no region could be determined, a HTTP status code 404 will be returned:

```
{  
  "error": {  
    "errors": [{  
      "domain": "geolocation",  
      "reason": "notFound",  
      "message": "Not found",  
    }],  
    "code": 404,  
    "message": "Not found",  
  }  
}
```

(continues on next page)



(continued from previous page)

```
}
}
```

## Geosubmit Version 2

**Purpose** Submit data about nearby Bluetooth beacons, cell or WiFi networks.

### Request

Geosubmit requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v2/geosubmit?key=<API_KEY>
```

There is an earlier *Geosubmit (Deprecated)* version one API, with a slightly different and less extensive field list.

Geosubmit requests are submitted using a POST request with a JSON body:

```
{
  "items": [
    {
      "timestamp": 1405602028568,
      "position": {
        "latitude": -22.7539192,
        "longitude": -43.4371081,
        "accuracy": 10.0,
        "age": 1000,
        "altitude": 100.0,
        "altitudeAccuracy": 50.0,
        "heading": 45.0,
        "pressure": 1013.25,
        "speed": 3.6,
        "source": "gps"
      },
    },
    {
      "bluetoothBeacons": [
        {
          "macAddress": "ff:23:45:67:89:ab",
          "age": 2000,
          "name": "beacon",
          "signalStrength": -110
        }
      ],
    },
    {
      "cellTowers": [
        {
          "radioType": "lte",
          "mobileCountryCode": 208,
          "mobileNetworkCode": 1,
          "locationAreaCode": 2,
          "cellId": 12345,
          "age": 3000,
          "asu": 31,
          "primaryScramblingCode": 5,
          "serving": 1,
          "signalStrength": -51,
          "timingAdvance": 1
        }
      ],
    }
  ],
}
```

(continues on next page)

(continued from previous page)

```
"wifiAccessPoints": [  
  {  
    "macAddress": "01:23:45:67:89:ab",  
    "age": 5000,  
    "channel": 6,  
    "frequency": 2437,  
    "radioType": "802.11n",  
    "signalToNoiseRatio": 13,  
    "signalStrength": -77  
  },  
  {  
    "macAddress": "23:45:67:89:ab:cd"  
  }  
]  
}]}
```

## Field Definition

Requests always need to contain a batch of *reports*. Each *report* must contain at least one entry in the *bluetoothBeacons* or *cellTowers* array or two entries in the *wifiAccessPoints* array.

Almost all of the fields are optional. For Bluetooth and WiFi records only the *macAddress* field is required.

## Global Fields

**timestamp** The time of observation of the data, measured in milliseconds since the UNIX epoch. Can be omitted if the observation time is very recent. The age values in each section are relative to this timestamp.

## Position Fields

The position block contains information about where and when the data was observed.

**latitude** The latitude of the observation (WSG 84).

**longitude** The longitude of the observation (WSG 84).

**accuracy** The accuracy of the observed position in meters.

**altitude** The altitude at which the data was observed in meters above sea-level.

**altitudeAccuracy** The accuracy of the altitude estimate in meters.

**age** The age of the position data (in milliseconds).

**heading** The heading field denotes the direction of travel of the device and is specified in degrees, where 0° heading < 360°, counting clockwise relative to the true north.

**pressure** The air pressure in hPa (millibar).

**speed** The speed field denotes the magnitude of the horizontal component of the device's current velocity and is specified in meters per second.

**source** The source of the position information. If the field is omitted, "gps" is assumed. The term *gps* is used to cover all types of satellite based positioning systems incl. Galileo and Glonass. Other possible values are *manual* for a position entered manually into the system and *fused* for a position obtained from a combination of other sensors or outside service queries.

### Bluetooth Beacon Fields

**macAddress** The address of the Bluetooth Low Energy (BLE) beacon.

**name** The name of the BLE beacon.

**age** The number of milliseconds since this BLE beacon was last seen.

**signalStrength** The measured signal strength of the BLE beacon in dBm.

### Cell Tower Fields

**radioType** The type of radio network. One of *gsm*, *wcdma* or *lte*.

**mobileCountryCode** The mobile country code.

**mobileNetworkCode** The mobile network code.

**locationAreaCode** The location area code for GSM and WCDMA networks. The tracking area code for LTE networks.

**cellId** The cell id or cell identity.

**age** The number of milliseconds since this cell was last seen.

**asu** The arbitrary strength unit indicating the signal strength if a direct signal strength reading is not available.

**primaryScramblingCode** The primary scrambling code for WCDMA and physical cell id for LTE.

**serving** A value of *1* indicates this as the serving cell, a value of *0* indicates a neighboring cell.

**signalStrength** The signal strength for this cell network, either the RSSI or RSCP.

**timingAdvance** The timing advance value for this cell tower when available.

### Wifi Access Point Fields

**macAddress** The BSSID of the Wifi network. Hidden Wifi networks must not be collected.

**radioType** The Wifi radio type, one of *802.11a*, *802.11b*, *802.11g*, *802.11n*, *802.11ac*.

**age** The number of milliseconds since this Wifi network was detected.

**channel** The channel is a number specified by the IEEE which represents a small band of frequencies.

**frequency** The frequency in MHz of the channel over which the client is communicating with the access point.

**signalStrength** The received signal strength (RSSI) in dBm.

**signalToNoiseRatio** The current signal to noise ratio measured in dB.

**ssid** The SSID of the Wifi network. Wifi networks with a SSID ending in *\_nomap* must not be collected.

### Response

Successful requests return a HTTP 200 response with a body of an empty JSON object.

## Geosubmit (Deprecated)

---

**Note:** Please use the *Geosubmit Version 2* API instead.

---

**Purpose** Submit data about nearby Bluetooth, cell and WiFi networks.

### Request

Geosubmit requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v1/geosubmit?key=<API_KEY>
```

Geosubmit requests are submitted using a POST request with a JSON body:

```
{ "items": [
  {
    "latitude": -22.7539192,
    "longitude": -43.4371081,
    "accuracy": 10.0,
    "altitude": 100.0,
    "altitudeAccuracy": 50.0,
    "timestamp": 1405602028568,
    "heading": 45.0,
    "speed": 3.6,
    "bluetoothBeacons": [
      {
        "macAddress": "ff:74:27:89:5a:77",
        "age": 2000,
        "name": "beacon",
        "signalStrength": -110
      }
    ],
    "cellTowers": [
      {
        "radioType": "gsm",
        "cellId": 12345,
        "locationAreaCode": 2,
        "mobileCountryCode": 208,
        "mobileNetworkCode": 1,
        "age": 3,
        "asu": 31,
        "signalStrength": -51,
        "timingAdvance": 1
      }
    ],
    "wifiAccessPoints": [
      {
        "macAddress": "01:23:45:67:89:ab",
        "age": 3,
        "channel": 6,
        "frequency": 2437,
        "signalToNoiseRatio": 13,
        "signalStrength": -77
      }
    ]
  }
],
```

(continues on next page)

(continued from previous page)

```

    {
      "macAddress": "23:45:67:89:ab:cd"
    }
  ]
}
}]

```

## Field Definition

Requests always need to contain a batch of *reports*. Each *report* must contain at least one entry in the *cellTowers* array or two entries in the *wifiAccessPoints* array.

Most of the fields are optional. For Bluetooth and WiFi records only the *macAddress* field is required. For cell records the *radioType*, *mobileCountryCode*, *mobileNetworkCode*, *locationAreaCode* and *cellId* fields are required.

The Bluetooth array is an extension and can contain four different fields for each Bluetooth network:

**macAddress** The address of the Bluetooth Low Energy (BLE) beacon.

**name** The name of the BLE beacon.

**age** The number of milliseconds since this Bluetooth beacon was last seen.

**signalStrength** The measured signal strength of the BLE beacon in dBm.

The cell record has been extended over the geolocate schema to include three more optional fields:

**age** The number of milliseconds since this cell was primary. If age is 0, the cell id represents a current observation.

**asu** The arbitrary strength unit. An integer in the range of 0 to 95 (optional).

**psc** The physical cell id as an integer in the range of 0 to 503 (optional).

The WiFi record has been extended with one extra optional field *frequency*. Either *frequency* or *channel* may be submitted to the geosubmit API as they are functionally equivalent.

**frequency** The frequency in MHz of the channel over which the client is communicating with the access point.

The top level schema is identical to the geolocate schema with the following additional fields:

**latitude** The latitude of the observation (WSG 84).

**longitude** The longitude of the observation (WSG 84).

**timestamp** The time of observation of the data, measured in milliseconds since the UNIX epoch. Should be omitted if the observation time is very recent.

**accuracy** The accuracy of the observed position in meters.

**altitude** The altitude at which the data was observed in meters above sea-level.

**altitudeAccuracy** The accuracy of the altitude estimate in meters.

**heading** The heading field denotes the direction of travel of the device and is specified in degrees, where 0° heading < 360°, counting clockwise relative to the true north. If the device cannot provide heading information or the device is stationary, the field should be omitted.

**speed** The speed field denotes the magnitude of the horizontal component of the device's current velocity and is specified in meters per second. If the device cannot provide speed information, the field should be omitted.

### Response

Successful requests return a HTTP 200 response with a body of an empty JSON object.

### Search (Deprecated)

---

**Note:** Please use the *Geolocate* API instead.

---

**Purpose** Determine the current location based on data provided about nearby Bluetooth, cell or WiFi networks and based on the IP address used to access the service.

### Request

Search requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v1/search?key=<API_KEY>
```

A search record can contain a list of Bluetooth, cell and WiFi records.

A example of a well formed JSON search request :

```
{
  "radio": "gsm",
  "blue": [
    {
      "key": "ff:23:45:67:89:ab",
      "age": 1000,
      "signal": -110
    },
    {
      "key": "ff:23:45:67:89:cd",
      "signal": -105
    }
  ],
  "cell": [
    {
      "radio": "umts",
      "mcc": 123,
      "mnc": 123,
      "lac": 12345,
      "cid": 12345,
      "signal": -61,
      "age": 1500,
      "asu": 26
    }
  ],
  "wifi": [
    {
      "key": "01:23:45:67:89:ab",
      "age": 3000,
      "channel": 11,
      "frequency": 2412,
      "signal": -50
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    },
    {
      "key": "01:23:45:67:ab:cd"
    },
    {
      "key": "01:23:45:67:cd:ef"
    }
  ]
}

```

## Field Definition

### Bluetooth Fields

For *blue* entries, the *key* field is required.

**key (required)** The *key* is the mac address of the Bluetooth network. So for example a valid key would look similar to *ff:23:45:67:89:ab*.

**age** The number of milliseconds since this BLE beacon was last seen.

**signal** The received signal strength (RSSI) in dBm, typically in the range of -10 to -127.

**name** The name of the Bluetooth network.

### Cell Fields

**radio** The type of radio network. One of *gsm*, *umts* or *lte*.

**mcc** The mobile country code.

**mnc** The mobile network code.

**lac** The location area code for GSM and WCDMA networks. The tracking area code for LTE networks.

**cid** The cell id or cell identity.

**age** The number of milliseconds since this networks was last detected.

**psc** The primary scrambling code for WCDMA and physical cell id for LTE.

**signal** The signal strength for this cell network, either the RSSI or RSCP.

**ta** The timing advance value for this cell network.

### WiFi Fields

For *wifi* entries, the *key* field is required. The client must check the Wifi SSID for a *\_nomap* suffix. Wifi networks with such a suffix must not be submitted to the server.

Most devices will only report the WiFi frequency or the WiFi channel, but not both. The service will accept both if they are provided, but you can include only one or omit both fields.

**key (required)** The client must check the WiFi SSID for a *\_nomap* suffix. WiFi networks with such a suffix must not be submitted to the server. WiFi networks with a hidden SSID should not be submitted to the server either.

The *key* is the BSSID of the WiFi network. So for example a valid key would look similar to *01:23:45:67:89:ab*.

**age** The number of milliseconds since this network was last detected.

**frequency** The frequency in MHz of the channel over which the client is communicating with the access point.

**channel** The channel is a number specified by the IEEE which represents a small band of frequencies.

**signal** The received signal strength (RSSI) in dBm, typically in the range of -51 to -113.

**signalToNoiseRatio** The current signal to noise ratio measured in dB.

**ssid** The SSID of the Wifi network. Wifi networks with a SSID ending in *\_nomap* must not be collected.

An example of a valid WiFi record is below:

```
{
  "key": "01:23:45:67:89:ab",
  "age": 1500,
  "channel": 11,
  "frequency": 2412,
  "signal": -51,
  "signalToNoiseRatio": 37
}
```

### Mapping records into a search request

The mapping can contain zero or more Bluetooth records, zero or more WiFi records and zero or more cell records. If any list of records is empty, it can be omitted entirely.

For Bluetooth and WiFi lookups at least two keys of nearby networks need to be provided. This is an industry standard that is meant to prevent you from looking up the position of a single network over time.

### Response

A successful response will be:

```
{
  "status": "ok",
  "lat": -22.7539192,
  "lon": -43.4371081,
  "accuracy": 100.0
}
```

The latitude and longitude are numbers, with seven decimal places of actual precision. The coordinate reference system is WGS 84. The accuracy is an integer measured in meters and defines a circle around the location.

Should the response be based on a GeoIP estimate:

```
{
  "status": "ok",
  "lat": 51.0,
  "lon": -0.1,
  "accuracy": 600000.0,
  "fallback": "ipf"
}
```

Alternatively the fallback field can also state *lacf* for an estimate based on a cell location area.

If no position can be determined, you instead get:



```
{
  "status": "not_found"
}
```

## Submit (Deprecated)

**Note:** Please use the *Geosubmit Version 2* API instead.

**Purpose** Submit data about nearby cell and WiFi networks.

## Request

Submit requests are submitted using a POST request to the following URL:

```
https://location.services.mozilla.com/v1/submit?key=<API_KEY>
```

with a JSON body:

```
{ "items": [
  {
    "lat": -22.7539192,
    "lon": -43.4371081,
    "time": "2012-03-01T00:00:00.000Z",
    "accuracy": 10.0,
    "altitude": 100.0,
    "altitude_accuracy": 1.0,
    "heading": 45.0,
    "speed": 13.88,
    "radio": "gsm",
    "blue": [
      {
        "key": "ff:74:27:89:5a:77",
        "age": 2000,
        "name": "beacon",
        "signal": -110
      }
    ],
    "cell": [
      {
        "radio": "umts",
        "mcc": 123,
        "mnc": 123,
        "lac": 12345,
        "cid": 12345,
        "age": 3000,
        "signal": -60
      }
    ],
    "wifi": [
      {
        "key": "01:23:45:67:89:ab",
        "age": 2000,
        "channel": 11,

```

(continues on next page)

(continued from previous page)

```
        "frequency": 2412,  
        "signal": -51  
      }  
    ]  
  }  
]
```

### Field Definition

The record fields have the same meaning and requirements as explained in the *Search (Deprecated)*.

The only required fields are *lat* and *lon* and at least one Bluetooth, cell or WiFi entry. If neither *lat* nor *lon* are included, the record will be discarded.

The altitude, accuracy and altitude\_accuracy fields are all measured in meters. Altitude measures the height above or below the mean sea level, as defined by WGS84.

The heading field specifies the direction of travel in  $0 \leq \text{heading} \leq 360$  degrees, counting clockwise relative to the true north.

The speed field specifies the current horizontal velocity and is measured in meters per second.

The heading and speed fields should be omitted from the report, if the speed and heading cannot be determined or the device was stationary while observing the environment.

The time has to be in UTC time, encoded in ISO 8601. If not provided, the server time will be used.

### Response

On successful submission, you will get a 204 status code back without any data in the body.

### Transfer (private)

---

**Note:** This is a private API for synchronizing multiple instances of the service and not open to the public.

---

**Purpose** Transfer aggregated area and station data from one instance of ichnaea to another.

### Request

Transfer requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v1/transfer?key=<API_KEY>
```

The requests must contain a JSON body:

```
{ "items": [  
  ] }
```

## Field Definition

Requests always need to contain a batch of items.

## Response

Successful requests return a HTTP 200 response with a body of an empty JSON object.

## 2.3 Data Export

Ichnaea supports automatic, periodic CSV (comma separated values) export of aggregate cell data (position estimates).

The data exchange format was created in collaboration with the [OpenCellID](#) project.

Records should be written one record to a line, with CRLF (0x0D 0x0A) as line separator.

A value should be written as an empty field – two adjacent commas, for example – rather than being omitted.

The first five fields (radio to cell) jointly identify a unique logical cell network. The remaining fields contain information about this network.

The data format does not specify the means and exact algorithms by which the position estimate or range calculation was done. The algorithms might be unique and changing for each source of the data, though both ichnaea and [OpenCellID](#) currently use similar and comparable techniques.

The fields in the CSV file are as follows:

### 2.3.1 Cell Fields

radio

Network type. One of the strings *GSM*, *UMTS* (for WCDMA networks) or *LTE*.

mcc

Mobile Country Code. An integer, for example *505*, the code for Australia.

net

For GSM, UMTS and LTE networks, this is the mobile network code (MNC). An integer, for example *4*, the MNC used by Vodafone in the Netherlands.

area

For GSM and UMTS networks, this is the location area code (LAC). For LTE networks, this is the tracking area code (TAC). An integer, for example *2035*.

cell

For GSM and LTE networks, this is the cell id or cell identity (CID). For UMTS networks this is the UTRAN cell id, which is the concatenation of 2 bytes of radio network controller (RNC) code and 2 bytes of cell id. An integer, for example *32345*.

unit

For UMTS networks, this is the primary scrambling code (PSC). For LTE networks, this is the physical cell id (PCI). For GSM networks, this is empty. An integer, for example *312*.

lon

Longitude in degrees between -180.0 and 180.0 using the WSG 84 reference system. A floating point number, for example *52.3456789*.

lat

Latitude in degrees between -90.0 and 90.0 using the WSG 84 reference system. A floating point number, for example *-10.034*.

range

Estimate of radio range, in meters. This is an estimate on how large each cell area is, as a radius around the estimated position and is based on the *observations* or a knowledgeable source. An integer, for example *2500*.

samples

Total number of *observations* used to calculate the estimated position, range and averageSignal. An integer, for example *1200*.

changeable

Whether or not this cell is a position estimate based on *observations*, and therefore subject to change in the future, or is an exact location entered from a knowledgeable source. A boolean value, encoded as either *1* (for “changeable”) or *0* (for “exact”).

created

Timestamp of the time when this record was first created. An integer, counting seconds since the UTC Unix Epoch of 1970-01-01T00:00:00Z. For example, *1406204196*, which is the timestamp for 2014-07-24T12:16:36Z.

updated

Timestamp of the time when this record was most recently modified. An integer, counting seconds since the UTC Unix Epoch of 1970-01-01T00:00:00Z. For example, *1406204196*, which is the timestamp for 2014-07-24T12:16:36Z.

averageSignal

Average signal strength from all observations for the cell network. An integer value, in dBm. For example, *-72*.

This field is only used by the *OpenCellID* project and has been used historically as a hint towards the quality of the position estimate.

## 2.4 Installation

If you want to run your own version of the service, you need to install and set up its various parts. There are two different ways to set up the service, depending on whether you want to simply run it or if you want to contribute back to the development of it.

### 2.4.1 Configuration

As part of deploying the application, you need to create an application configuration file, commonly called `location.ini` and insert a couple of rows into various database tables.

## Configuration File

As explained in the *the deployment documentation* the processes find this configuration file via the `ICHNAEA_CFG` environment variable. The variable should contain an absolute path, for example `/etc/location.ini`.

The configuration file is an ini-style file and contains a number of different sections.

## Required Sections

### Cache

The cache section contains a `cache_url` pointing to a Redis server.

The cache is used as a classic cache by the webapp code, as a backend to store rate-limiting counters, as a custom and a celery queuing backend.

```
[cache]
cache_url = redis://localhost:6379/0
```

### Database

The database section contains settings for accessing the MySQL database.

The web application only requires and uses the read-only connection, while the asynchronous celery workers only use the read-write connection.

Both of them can be restricted to only DML (data-manipulation) permissions, as neither needs DDL (data-definition) rights.

DDL changes are only done via the alembic database migration system, which has a separate `alembic.ini` configuration file.

```
[database]
rw_url = mysql+pymysql://rw_user:password@localhost/location
ro_url = mysql+pymysql://ro_user:password@localhost/location
```

### GeoIP

The `geoip` section contains settings related to the maxmind GeoIP database.

The `db_path` setting needs to point to a maxmind GeoIP city database in version 2 format. Both GeoLite and commercial databases will work.

```
[geoip]
db_path = /path/to/GeoIP2-City.mmdb
```

## Optional Sections

### Assets

The assets section contains settings for a static file repository (Amazon S3) and a public DNS to access those files via HTTPS (Amazon CloudFront).

These are used to store and serve both the image tiles generated for the data map and the public export files available via the downloads section of the website.

```
[assets]
bucket = amazon_s3_bucket_name
url = https://some_distribution_id.cloudfront.net
```

### Sentry

The sentry section contains settings related to a Sentry server.

The dsn setting needs to contain a valid DSN project entry.

```
[sentry]
dsn = https://public_key:secret_key@localhost/project_id
```

### StatsD

The statsd section contains settings related to a StatsD service. The project uses a lot of metrics as further detailed in *the metrics documentation*.

The host and port settings determine how to connect to the service via UDP.

Since a single StatsD service usually supports multiple different projects, the metric\_prefix setting can be used to prefix all metrics emitted by this project with a unique name.

The tag\_support setting can either be false or true and declares whether or not the StatsD service supports metric tags. *Datadog* is an example of a service that supports tags. If tag\_support is false, the tags will be emitted as part of the standard metric name.

```
[statsd]
host = localhost
port = 8125
metric_prefix = location
tag_support = true
```

For initial testing it can be useful to simply capture the statsd metrics without running an actual statsd daemon. To do so you can use the `nc -lku localhost 8125` command to run a UDP service and print out all incoming data on the console.

### Web

The web section contains settings related to the non-API website content.

The web functionality by default is limited to the public HTTP API. If the enabled setting is set to true the website content pages are also made available.

The map\_id\_base and map\_id\_labels settings specify Mapbox map ids for a base map and a map containing only labels. The map\_token specifies a Mapbox access token.

```
[web]
enabled = true
map_id_base = example_base.map-123
map_id_labels = example_labels.map-234
map_token = pk.example_public_access_token
```

## Database Configuration

### API Keys

The project requires API keys to access the locate APIs. You need to add API keys manually to the database by direct SQL inserts.

API keys can be any string of up to 40 characters, though random UUID4s in hex representation are commonly used, for example `329694ac-a337-4856-af30-66162bc8187a`.

But to start off, you can add a simple literal *test* API key:

```
INSERT INTO api_key
(`valid_key`, `allow_locate`) VALUES ("test", 1);
```

### Export Configuration

The project supports exporting all data that its gets via the submit-style APIs to different backends. This configuration lives in the *export\_config* database table.

Currently three different kinds of backends are supported:

- Amazon S3 buckets
- The projects own internal data processing pipeline
- A HTTPS POST endpoint accepting the geosubmit v2 format

The type of the target is determined by the *schema* column of each entry.

All export targets can be configured with a *batch* setting that determines how many reports have to be available before data is submitted to the backend.

All exports have an additional *skip\_keys* setting as a set of API keys. Data submitted using one of these API keys will not be exported to the target.

There can be multiple instances of the bucket and HTTP POST export targets, but only one instance of the internal export.

In the simplest case, you insert one row to send data to the internal data pipeline via:

```
INSERT INTO export_config
(`name`, `batch`, `schema`) VALUES ("internal", 1, "internal");
```

For a production setup you want to set the batch column to something like *100* or *1000* to get more efficiency. For initial testing its easier to set it to *1* so you immediately process any incoming data.

### Bucket Export

The Amazon S3 bucket export combines reports into a gzipped JSON file and uploads them to the specified bucket url, for example:

```
s3://amazon_s3_bucket_name/directory/{source}{api_key}/{year}/{month}/{day}
```

The schema column must be set to *s3*.

The url can contain any level of additional static directories under the bucket root. The `{api_key}/{year}/{month}/{day}` parts will be dynamically replaced by the *api\_key* used to upload the data, the source of the report (e.g. gnss) and the date when the backup took place. The files use a random UUID4 as the filename.





Specifically Amazon CloudFront and S3 are only used for backup and serving image tiles and public data downloads for the public website. Using Combain, Datadog, OpenCellID and Sentry is also optional. Finally there doesn't have to be a *admin* EC2 box, but it can be helpful for debug access and running database migrations.

## MySQL / Amazon RDS

The application is written and tested against MySQL 5.6.x or Amazon RDS of the same version. The default configuration works for the most part. There are just three changes you need to do. For example via the my.cnf:

```
[mysqld]
innodb_file_format=Barracuda
innodb_strict_mode=on
sql-mode="NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES"
```

The web app frontend role only needs access to a read-only read-replica of the database. The async worker backend role needs access to a read-write primary database. You can use load balancers like haproxy to distribute load from the web app frontends over multiple read-replicas. The diagram shows a final fallback connection, where the web app frontends fall back to talking to the primary database if all read-replicas are down. An example of such a haproxy configuration is included in the source code.

## Redis / Amazon ElastiCache

The application uses Redis as a queue for the asynchronous task workers and also uses it directly as a cache and to track API key rate limitations.

You can install a standard local Redis for development or production use. The application is also compatible with Amazon ElastiCache (Redis).

## Amazon S3

The application uses Amazon S3 for various tasks, including backup of *observations*, export of the aggregated cell table and hosting of the data map image tiles.

All of these are triggered by asynchronous jobs and you can disable them if you are not hosted in an Amazon environment.

If you use Amazon S3 you might want to configure a lifecycle policy to delete old export files after a couple of days and *observation* data after one year.

## Datadog / Statsd / Sentry

The application uses Statsd to aggregate stats and Sentry to log exception messages.

To use Statsd, you need to configure it in the config file, specifying a host and port, for example localhost port 8125.

To get the app to log exceptions to Sentry, you will need to obtain the DSN for your Sentry instance. Edit location.ini and in the *sentry* section put your real DSN into the *dsn* setting.

Installation of Statsd and Sentry are outside the scope of this documentation.

## Image Tiles

The code includes functionality to render out image tiles for a data map of places where observations have been made. This part of the code relies on two external projects. One is the [datamaps image tile generator](#) the other is [pngquant](#). Make sure to install both of them and make their binaries available on your system path. The datamaps package includes the *encode*, *enumerate* and *render* tools and the pngquant package includes a tool called *pngquant*. You can install these tools on the server running the celery scheduler (celerybeat) and trigger the *location\_map* script via a cron job.

## 2.4.3 Development

### Prerequisites

In order to install a development version of the service, you need to have a Linux or Mac OS machine and [install docker](#) and docker-compose.

On Linux you can use your OS level package manager to install them.

On Mac OS you need to install [Docker for Mac](#). Docker Toolbox or docker-machine based setups aren't supported by the documentation and Makefile.

### Docker

We use docker to run additional development dependencies like Redis and MySQL with the exact versions we want to test against.

We assume that you can run `docker` and `docker-compose` on your command line. Test this via:

```
docker --version
docker-compose --version
```

### Requirements

In order to run the code you need to have Python 2.6, 2.7 or 3.5 installed on your system. The default Makefile also assumes a *virtualenv* command is globally available. If this isn't true for your system, please create a virtualenv manually inside the ichnaea folder before continuing (`/path/to/virtualenv --python=python2.6 .`).

In the next step you are going to install a good number of Python libraries, which depend on various OS level C libraries. These C libraries are best installed via the OS level package management system. We list the CentOS/Redhat names, but they should be similar on other OS.

Runtime requirements:

```
openssl, python, libmaxminddb, libffi, atlas-sse3, geos, spatialindex-devel
```

Build requirements:

```
openssl-devel, gcc, gcc-c++, gcc-gfortran, make, python, python-pip,
python-virtualenv, git, libmaxminddb, libffi-devel, atlas-devel,
geos-devel, spatialindex-devel
```

## Code

Now run the following command to get the code:

```
git clone https://github.com/mozilla/ichnaea
cd ichnaea
```

Then run make, which is going to take quite a while the first time:

```
make
```

Now you can run the web app for example on port 7001:

```
ICHNAEA_CFG=location.ini bin/gunicorn -b 127.0.0.1:7001 \
  -c python:ichnaea.webapp.settings ichnaea.webapp.app:wsgi_app
```

The celery processes are started via:

```
ICHNAEA_CFG=location.ini bin/celery -A ichnaea.async.app:celery_app beat

ICHNAEA_CFG=location.ini bin/celery -A ichnaea.async.app:celery_app worker \
  -Ofair --no-execv --without-mingle --without-gossip
```

## Documentation

In order to create and test the documentation locally run:

```
make docs
```

The documentation will be available in docs/build/html/index.html.

## Python Dependencies

The project uses [requires.io](#) to track whether or not the Python dependencies are outdated.

If they are, update the version pins in the various *requirements/\*.txt* files and rerun *make*, *make docs* or *make test*, depending on which requirements have changed.

## CSS / JS / Images

The project depends on a number of external web assets. Those dependencies are tracked via npm and bower in files under *docker/node*.

In order to install them, run:

```
make css
make js
```

This will install build tools and bower assets inside a docker container. It will also copy, compile and minify files in various folders under *ichnaea/content/static/*.

To check if the external assets are outdated run:

```
docker run --rm -it mozilla-ichnaea/node:latest bower list
```

### Cleanup

In case the local environment gets into a weird or broken state, it can be cleaned up by running:

```
make clean
```

Of course one can also delete the entire git repository and start from a fresh checkout.

### Release Build

The default *make / make build* target installs a local development version including database setup and testing tools. For a production environment or release pipeline one can instead use:

```
make release
```

This will not do any database setup and only install production dependencies. It will also create a virtualenv and install the ichnaea code itself via *bin/python setup.py install*, so that a copy will be installed into *lib/pythonX.Y/site-packages/*.

The step will also compile all py files to pyc files and remove any files from the tree which aren't compatible with the active Python version (blocklist in the *compile.py* script). The removal step ensures that any build tools (for example *rpmbuild / mock*) that typically call *compileall.compile\_dir* will work, without breaking on the incompatible files.

## 2.4.4 Metrics

As discussed in [the deployment document](#), Ichnaea emits metrics through the Statsd client library with the intent of aggregating and viewing them on a compatible dashboard.

This document describes the metrics collected.

The code emits most metrics using the statsd tags extension. A metric name of `task#name:function,version:old` therefor means a statsd metric called `task` will be emitted with two tags `name:function` and `version:old`. If the statsd backend does not support tags, the statsd client can be configured with a `tag_support = false` option. In this case the above metric would be emitted as: `task.name_function.version_old`.

### API Request Metrics

These are metrics that track how many times each specific public API is used and which clients identified by their API keys do so. They are grouped by the type of the API, where type is one of *locate*, *region* and *submit*, independent of the specific version of that API.

These metrics can help in deciding when to remove a deprecated API.

```
locate.request#path:v1.search,key:<apikey>,          locate.request#path:v1.
geolocate,key:<apikey>,    region.request#path:v1.country,key:<apikey>,    submit.
request#path:v1.submit,key:<apikey>,          submit.request#path:v1.geosubmit,
key:<apikey>,submit.request#path:v2.geosubmit,key:<apikey>: counters
```

These metrics count how many times a specific API was called by a specific API key expressed via the API keys short name. The API key is the actual API key, often a UUID.

Two special short names exist for tracking invalid (`invalid`) and no (`none`) provided API keys.

## API User Metrics

For all API requests including the submit-type APIs we gather metrics about the number of unique users based on the users IP addresses.

These metrics are gathered under the metric prefix:

```
<api_type>.user#key<apikey>: gauge
```

They have an additional tag to determine the time interval for which the unique users are aggregated for:

```
#interval:1d, interval:7d: tags
```

Unique users per day or last 7 days.

Technically these metrics are based on HyperLogLog cardinality numbers maintained in a Redis service. They should be accurate to about 1% of the actual number.

## API Query Metrics

For each incoming API query we log metrics about the data contained in the query with the metric name and tags:

```
<api_type>.query#key<apikey>, region:<region_code>: counter
```

*api\_type* describes the type of API being used, independent of the version number of the API. So *v1/country* gets logged as *region* and both *v1/search* and *v1/geolocate* get logged as *locate*.

*region\_code* is either a two-letter GENC region code like *de* or the special value *none* if the region of origin of the incoming request could not be determined.

We extend the metric with additional tags based on the data contained in it:

```
#geoip:false: tag
```

This tag only gets added if there was no valid client IP address for this query. Since almost all queries contain a client IP address we usually skip this tag.

```
#blue:none, #blue:one, #blue:many, #cell:none, #cell:one, #cell:many, #wifi:none, #wifi:one, #wifi:many: tags
```

If the query contained any Bluetooth, cell or WiFi networks, one blue, cell and wifi tag get added. The tags depend on the number of valid *stations* for each of the three.

## API Result Metrics

Similar to the API query metrics we also collect metrics about each result of an API query. This follows the same per API type and per region rules under the prefix / tag combination:

```
<api_type>.result#key:<apikey>, region:<region_code>
```

The result metrics measure if we satisfied the incoming API query in the best possible fashion. Incoming queries can generally contain an IP address, Bluetooth, cell, WiFi networks or any combination thereof. If the query contained only cell networks, we do not expect to get a high accuracy result, as there is too little data in the query to do so.

We express this by classifying each incoming query into one of four categories:

**High Accuracy (#accuracy:high)** A query containing at least two Bluetooth or WiFi networks.

**Medium Accuracy (#accuracy:medium)** A query containing neither Bluetooth nor WiFi networks but at least one cell network.

**Low Accuracy (#accuracy:low)** A query containing no networks but only the IP address of the client.

**No Accuracy (#accuracy:none)** A query containing no usable information, for example an IP-only query that explicitly disables the IP fallback.

A query containing multiple data types gets put into the best possible category, so for example any query containing cell data will at least be of medium accuracy.

Once we have determined the expected accuracy category for the query, we compare it to the accuracy category of the result we determined. If we can deliver an equal or better category we consider the status to be a *hit*. If we don't satisfy the expected category we consider the result to be a *miss*.

For each result we then log exactly one of the following tag combinations:

```
#accuracy:high,status:hit,      #accuracy:high,status:miss,      #accuracy:medium,
status:hit,#accuracy:medium,status:miss,#accuracy:low,status:hit,#accuracy:low,
status:miss:tags
```

We don't log metrics for the uncommon case of `none` or no expected accuracy.

One special case exists for cell networks. If we cannot find an exact cell match, we might fall back to a cell area based estimate. If the range of the cell area is fairly small we consider this to be a `#accuracy:medium,status:hit`. But if the size of the cell area is extremely large, in the order of tens of kilometers to hundreds of kilometers, we consider it to be a `#accuracy:medium,status:miss`.

In the past we only collected stats based on whether or not cell based data was used to answer a cell based query and counted it as a cell-based success, even if the provided accuracy was really bad.

In addition to the accuracy of the result, we also tag the result metric with the data source that got used to provide the result, but only for results that met the expected accuracy.

```
#source:<source_name>:tag
```

Data sources can be one of:

**internal** Data from our own crowd-sourcing effort.

**fallback** Data from the optional external fallback provider.

**geoip** Data from a GeoIP database.

And finally we add a tag to state whether or not the query was allowed to use the fallback source.

```
#fallback_allowed:<value>:tag
```

The value is either *true* or *false*.

## API Source Metrics

In addition to the final API result, we also collect metrics about each individual data source we use to answer queries under the `<api_type>.source#key:<apikey>,region:<region_code>` metric.

Each request may use one or multiple of these sources to deliver a result. We log the same metrics as mentioned above for the result.

All of this combined might lead to a tagged metric like:

```
locate.source#key:test,region:de,source:geoip,accuracy:low,status:hit
```

## API Fallback Source Metrics

The external fallback source has a couple extra metrics to observe the performance of outbound network calls and the effectiveness of its cache.

```
locate.fallback.cache#status:hit,    locate.fallback.cache#status:miss,    locate.fallback.cache#status:bypassed,    locate.fallback.cache#status:inconsistent,
locate.fallback.cache#status:failure: counter
```

Counts the number of hits and misses for the fallback cache. If the query should not be cached, a *bypassed* status is used. If the cached values couldn't be read, a *failure* status is used. If the cached values didn't agree on a consistent position, a *inconsistent* status is used.

```
locate.fallback.lookup#fallback_name:<fallback_name>: timer
```

Measures the time it takes to do each outbound network request. The fallback name tag specifies which fallback service is used.

```
locate.fallback.lookup#fallback_name:<fallback_name>, status:<code>: counter
```

Counts the HTTP response codes for all outbound requests per named fallback service. There is one counter per HTTP response code, for example *200*.

## Data Pipeline Metrics

When a batch of reports is accepted at one of the submission API endpoints, it is decomposed into a number of “items” – wifi or cell *observations* – each of which then works its way through a process of normalization, consistency-checking and eventually (possibly) integration into aggregate *station* estimates held in the main database tables. Along the way several counters measure the steps involved:

```
data.batch.upload, data.batch.upload#key:<apikey>: counters
```

Counts the number of “batches” of *reports* accepted to the data processing pipeline by an API endpoint. A batch generally corresponds to the set of *reports* uploaded in a single HTTP POST to one of the submit APIs. In other words this metric counts “submissions that make it past coarse-grained checks” such as API-key and JSON schema validity checking.

The metric is either emitted per tracked API key, or for everything else without a key tag.

```
data.report.upload, data.report.upload#key:<apikey>: counters
```

Counts the number of *reports* accepted into the data processing pipeline. The metric is either emitted per tracked API key, or for everything else without a key tag.

```
data.report.drop, data.report.drop#key:<apikey>: counter
```

Count incoming *reports* that were discarded due to some internal consistency, range or validity-condition error.

```
data.observation.upload#type:blue,    data.observation.upload#type:blue,
key:<apikey>,    data.observation.upload#type:cell,    data.observation.upload#type:cell,
key:<apikey>,    data.observation.upload#type:wifi,    data.observation.upload#type:wifi,
key:<apikey>: counters
```

Count the number of Bluetooth, cell or WiFi *observations* entering the data processing pipeline; before normalization and blacklist processing have been applied. In other words this metric counts “total Bluetooth, cell or WiFi *observations* inside each submitted batch”, as each batch is composed of individual *observations*.

The metrics are either emitted per tracked API key, or for everything else without a key tag.

```
data.observation.drop#type:blue,    data.observation.drop#type:blue, key:<apikey>,
data.observation.drop#type:cell,    data.observation.drop#type:cell, key:<apikey>,
data.observation.drop#type:wifi    data.observation.drop#type:wifi, key:<apikey> :
counters
```

Count incoming Bluetooth, cell or WiFi *observations* that were discarded before integration due to some internal consistency, range or validity-condition error encountered while attempting to normalize the *observation*.

```
data.observation.insert#type:blue,    data.observation.insert#type:cell,    data.
observation.insert#type:wifi: counters
```

Count Bluetooth, cell or WiFi *observations* that are successfully normalized, integrated and not discarded due to consistency errors.

```
data.station.blocklist#type:blue,    data.station.blocklist#type:cell,    data.
station.blocklist#type:wifi: counters
```

Count any Bluetooth, cell or WiFi network that is blocklisted due to the acceptance of multiple *observations* at sufficiently different locations. In these cases, we decide that the *station* is “moving” (such as a picocell or mobile hotspot on a public transit vehicle) and blocklist it, to avoid estimating query positions using the *station*.

```
data.station.confirm#type:blue,    data.station.confirm#type:cell,    data.station.
confirm#type:wifi: counters
```

Count the number of Bluetooth, cell or WiFi *station* that were successfully confirmed by any type of *observations*.

```
data.station.new#type:blue,          data.station.new#type:cell,          data.station.
new#type:wifi: counters
```

Count the number of Bluetooth, cell or WiFi *station* that were discovered for the first time.

## Data Pipeline Export Metrics

Incoming *reports* can also be sent to a number of different export targets. We keep metrics about how those individual export targets perform.

```
data.export.batch#key:<export_key>: counter
```

Count the number of batches sent to the export target.

```
data.export.upload#key:<export_key>: timer
```

Track how long the upload operation took per export target.

```
data.export.upload#key:<export_key>, status:<status>: counter
```

Track the upload status of the current job. One counter per status. A status can either be a simple *success* and *failure* or a HTTP response code like 200, 400, etc.

## Internal Monitoring

```
api.limit#key:<apikey>, #path:<path>: gauge
```

One gauge is created per API key and API path which has rate limiting enabled on it. This gauge measures how many requests have been done for each such API key and path combination for the current day.

```
queue#queue:celery_blue,    queue#queue:celery_cell,    queue#queue:celery_default,
queue#queue:celery_export, queue#queue:celery_incoming, queue#queue:celery_monitor,
queue#queue:celery_reports, queue#queue:celery_wifi: gauges
```

These gauges measure the number of tasks in each of the Redis queues. They are sampled at an approximate per-minute interval.



```
queue#queue:update_blue_0,queue#queue:update_blue_f,queue#queue:update_cell_gsm,
queue#queue:update_cell_wcdma,queue#queue:update_cell_lte,
queue#queue:update_cellarea,queue#queue:update_datamap_ne,
queue#queue:update_datamap_nw,queue#queue:update_datamap_se,
queue#queue:update_datamap_sw,queue#queue:update_wifi_0,
queue#queue:update_wifi_f: gauges
```

These gauges measure the number of items in the Redis update queues.

## HTTP Counters

Every legitimate, routed request to an API endpoint or to a content view increments a `request#path:<path>, method:<method>, status:<code>` counter.

The path of the counter is the based on the path of the HTTP request, with slashes replaced with periods. The method tag contains the lowercased HTTP method of the request. The status tag contains the response code produced by the request.

For example, a GET of `/stats/regions` that results in an HTTP 200 status code, will increment the counter `request#path:stats.regions, method:get, status:200`.

Response codes in the 400 range (eg. 404) are only generated for HTTP paths referring to API endpoints. Logging them for unknown and invalid paths would overwhelm the system with all the random paths the friendly Internet bot army sends along.

## HTTP Timers

In addition to the HTTP counters, every legitimate, routed request emits a `request#path:<path>, method:<method>` timer.

These timers have the same structure as the HTTP counters, except they do not have the response code tag.

## Task Timers

Our data ingress and data maintenance actions are managed by a Celery queue of tasks. These tasks are executed asynchronously, and each task emits a timer indicating its execution time.

For example:

- `task#task:data.export_reports`
- `task#task:data.update_statcounter`

## Datamaps Timers

We include a script to generate a data map from the gathered map statistics. This script includes a number of timers and pseudo-timers to monitor its operation.

```
datamaps#func:export, datamaps#func:encode, datamaps#func:merge,
datamaps#func:main, datamaps#func:render, datamaps#func:upload: timers
```

These timers track the individual functions of the generation process.

```
datamaps#count:csv_rows, datamaps#count:quadtrees, datamaps#count:tile_new,
datamaps#count:tile_changed, datamaps#count:tile_deleted,
datamaps#count:tile_unchanged: timers
```

Pseudo-timers to track the number of CSV rows, Quadtree files and image tiles.

## 2.4.5 Testing

---

**Note:** Since the tests use a real database and Redis connection, you cannot parallelize any tests.

---

### Unit Tests

If you have a local development environment setup, you can run all tests including coverage tests via:

```
make test
```

Or run individual test modules via for example:

```
make test TESTS=ichnaea.tests.test_geoip
```

### Testing Tasks

The project includes a good number of asynchronous tasks, executed by Celery in the code. The easiest way to test them is by writing unit tests and calling the task functions directly.

In order to do more integration testing, you can also manually queue a task by using the celery *call* command. For example:

```
ICHNAEA_CFG=location.ini bin/celery -A ichnaea.async.app:celery_app call \
    ichnaea.data.tasks.update_cell --args='[1000, ]'
```

You then need to run the celery worker process and it will pick up the task from the queue and execute it:

```
ICHNAEA_CFG=location.ini bin/celery \
    -A ichnaea.async.app:celery_app worker -c 1
```

### Testing Multiple Python Versions

The project supports multiple Python versions. In order to run the tests against all of them locally, we are using tox:

```
bin/tox
```

You can explicitly state what Python versions to test:

```
bin/tox -e=py{26,27,34}
```

You can also run a subset of all tests, the same way as via *make test*:

```
bin/tox TESTS=ichnaea.tests.test_util
```

If the tox environment got into a weird state, just recreate it:

```
bin/tox --recreate
```

Of course these options can be combined, for example:

```
bin/tox -e=py{34} TESTS=ichnaea.tests.test_util
```

Since the project relies on a number of non-Python dependencies, each tox environment is created from a full copy of the git repo. To ease testing the *ichnaea* code package is then removed from inside each tox environment and finally *bin/install -e ichnaea /path/to/parent/repo* is called. This means the code inside the top-level *ichnaea* code package is actually used from inside each tox environment.

If you set a pdb breakpoint in the normal main code, you'll thus get an easy way to have a pdb inside each tox environment.

One caveat is that this leads to the Cython based *geocalc.so* file to be recompiled with the last Python version you used for tox. You may need to run *make* again to recreate based on the Python version of the non-tox environment.

## 2.5 Algorithms

The project uses a couple of different approaches and algorithms.

There's two general approaches to calculate positions from signal sources, without the cooperation of the signal sources or mobile networks.

1. Determine the location of signal sources from *observations*, then compare / trilaterate user locations.
2. Generate signal fingerprints for a fine-grained grid of the world. Find best match for a observed fingerprint.

The second approach has a much better accuracy, but relies on a lot more available and constantly updated data. For most of the world this approach is not practical, so we currently focus on approach one.

In theory one would assume that one could use signal strength data to infer a distance measure from it. The further a device is away from the signal source, the weaker the signal should get.

Unfortunately the signal strength is more dependent on the device type, how a user holds a device and changing environmental factors like trucks in the way. Even worse modern networks adjust their signal strength to the number of devices inside their reception area. This makes this data highly unreliable while looking up a user's position via a single reading.

In aggregate over many data points this information can still be valuable in determining the actual position of the signal source. While observing multiple signals at once, their relative strengths can also be used, as this keeps some of the changing factors constant, like the device type.

One other approach is using time of flight data as a distance metric. While there are some reflection and multipath problems, it's a much more accurate distance predictor. Fine grained enough timing data is unfortunately almost never available to the application or operating system layer in client devices. Some LTE networks and really modern WiFi networks with support for 802.11v are the rare exception to this. But these are so rare, that we currently ignore timing data.

### 2.5.1 Accuracy

Depending on the signal standard, we can promise different sorts of accuracy.

Underlying this is the assumption that we have enough data about the area at all. With no or too little data we'll have to fallback to less accurate data sources. Bluetooth is the most accurate, followed by WiFi and then cell based estimation using single cells, multiple cells or cell location areas. If all else fails GeoIP serves as a general fallback.

## Bluetooth / WiFi

Bluetooth and WiFi networks have a fairly limited range. Bluetooth low-energy beacons typically just a couple meters and WiFi networks up to 100 meters. With obstacles like walls and people in the way, these get even lower.

But this data can be skewed when the device in question is moving. It takes some time to do a network scan and devices tend to cache this information heavily. So there can be a time delta of tens of seconds between when a network was actually seen and when it is reported to the application layer. With a fast moving device this can lead to inaccuracies of a couple kilometers. WiFi networks tend to show up in scan long after they are out of reach, if the the device was actually connected to these networks.

This means position estimates based on WiFi networks are usually accurate to 100 meters. If a lot of networks are available in the area accuracy tends to increase to about 10 or 20 meters. Bluetooth networks tend to be accurate to about 10 meters.

One challenge that's particular severe in Bluetooth and WiFi networks are all the constantly moving networks, for example those installed on buses or trains or in the form of hotspot enabled mobile phones or tablets. So movement detection and detecting inconsistencies between observed data and the database world view are important considerations.

## GSM Cells

In GSM networks one typically has only access to the unique cell id of the serving cell. In GSM networks the phone does not know the full cell ids of any neighboring cells, unless it associates with the new cell as part of a hand-over, forgetting the cell id of the old cell.

So we are limited to a basic *Cell-ID* approach, meaning we assume that the user is at the center of the current GSM cell area and use the cell radius as the accuracy.

GSM cells are restricted to a maximum range of 35km, but there are rare exceptions using the GSM extended range of 120km.

In more populated places the cell sizes are typically much smaller, but generally accuracy will be in the tens of kilometer range.

## WCDMA Cells

In WCDMA networks neighboring cell information can be available. But limitations in chipset drivers, the radio interface layer and the operating systems often hide this data from application code. Or only partially expose the cell identifiers, for example only exposing the carrier and primary scrambling code of the neighboring cells.

So in most cases we are limited to the same approach as for GSM cells. The typical cell sizes of WCDMA cells are much smaller, which practically leads to a better accuracy. But WCDMA cells in rural areas can have a larger radius than GSM cells and we observed cells sizes of 60-70km.

In urban areas we should typically see accuracy in the 1 to 10 kilometer range.

## LTE Cells

LTE networks are similar to WCDMA networks and the same restriction on neighboring cells applies. Instead of a primary scrambling code LTE uses a physical cell id, which for our purposes has similar characteristics.

LTE cells are again often even smaller than WCDMA cells which leads to better accuracies.

LTE networks also expose a time based distance metric in the form of the timing advance. While we currently don't use this information, in the future it has the potential to significantly improve position estimates based on multiple cells.

## GeoIP

The accuracy of GeoIP depends a lot on the region the user is in. In the US GeoIP can be fairly accurate and often places the user in the right city or metropolitan area. In many other parts of the world GeoIP is only accurate to the region level.

Typical GeoIP accuracies are either in the 25 km range for city based estimates or multiple hundred kilometers for region based estimates.

IP version 6 has the chance to improve this situation, as the need for private carrier networks and network address translation decreases. But so far this hasn't made any measurable impact and most traffic is still restricted to IP version 4.

## 2.6 Internal Code API

This section describes the internal Python API's used throughout the code base. For documentation on the public HTTP API please look at [Services API](#).

### 2.6.1 Overview

The code is structured around a couple of larger functional areas in addition to top-level helper modules.

There are two sub-packages *async* and *webapp* related to runtime configuration and settings of the two process classes. *async* deals with Celery and *webapp* deals with *gunicorn*.

The *api* sub-package deals with the implementation of the public HTTP API. The *content* sub-package includes the public website content. The *data* sub-package includes the data pipeline code, in the form of asynchronous Celery tasks. The *models* sub-package deals with database models and schema validation. *scripts* has the implementation of command line scripts.

### 2.6.2 Indices and Tables

- [genindex](#)
- [modindex](#)

`ichnaea.api`

`ichnaea.api.config`

`ichnaea.api.exceptions`

`ichnaea.api.locate`

`ichnaea.api.locate.blue`

`ichnaea.api.locate.cell`

`ichnaea.api.locate.constants`

`ichnaea.api.locate.fallback`

`ichnaea.api.locate.geoip`

`ichnaea.api.locate.internal`

`ichnaea.api.locate.mac`

`ichnaea.api.locate.query`

`ichnaea.api.locate.result`

`ichnaea.api.locate.schema`

`ichnaea.api.locate.schema_v0`

`ichnaea.api.locate.schema_v1`

`ichnaea.api.locate.searcher`

`ichnaea.api.locate.source`

`ichnaea.api.locate.views`

`ichnaea.api.locate.wifi`

`ichnaea.api.rate_limit`

`ichnaea.api.schema`

`ichnaea.api.submit`

`ichnaea.api.submit.schema`

`ichnaea.api.submit.schema_v0`

`ichnaea.api.submit.schema_v1`

`ichnaea.api.submit.schema_v2`

`ichnaea.api.submit.views`

`ichnaea.api.transfer`

`ichnaea.api.transfer.schema`

`ichnaea.api.transfer.views`

`ichnaea.api.views`

`ichnaea.async`

`ichnaea.async.app`

This is also the Celery app public endpoint, used on the command line via:

```
bin/celery -A ichnaea.async.app:celery_app <worker, beat>
```

`ichnaea.async.config`

`ichnaea.async.settings`

`ichnaea.async.task`

`ichnaea.cache`

`ichnaea.config`

`ichnaea.constants`

`ichnaea.content`

`ichnaea.content.views`

`ichnaea.data`

`ichnaea.data.tasks`

`ichnaea.db`

`ichnaea.exceptions`

`ichnaea.floatjson`

`ichnaea.geocalc`

`ichnaea.geocode`

`ichnaea.geoip`

`ichnaea.http`

`ichnaea.log`

`ichnaea.models`

`ichnaea.models.api`

`ichnaea.models.base`

`ichnaea.models.blue`

`ichnaea.models.cell`

## Cell Networks

As part of the public API and the internal models, we define unique keys and additional data for each cell network. Depending on the cell radio network type, these records contain different information.

The classification of radio types is based on the [Android TelephonyManager constants](#). A similar classification exists for Firefox OS devices with the [MozMobileConnectionInfo API](#).

## GSM

If the network is either GSM or any high-data-rate variant of it, the radio type should be specified as *gsm*. This includes *GSM*, *EDGE* and *GPRS*.

Example:

```
{
  "radio": "gsm",
  "mcc": 123,
  "mnc": 123,
  "lac": 12345,
  "cid": 12345,
  "signal": -61,
  "asu": 26,
  "ta": 10
}
```

**radio (required)** The string *gsm*.

**mcc (required)** The mobile country code. An integer in the range of 1 to 999.

**mnc (required)** The mobile network code. An integer in the range of 0 to 999.

**lac (required)** The location area code. An integer in the range of 1 to 65533. According to [TS 24.008 10.5.3](#) both 0 and 65534 are reserved values indicating a deleted state.

**cid (required)** The cell id. An integer in the range of 1 to 65535.

**signal** The received signal strength (RSSI) in dBm, typically in the range of -51 to -113 (optional).

**asu** The arbitrary strength unit. An integer in the range of 0 to 31 (optional). Unknown values can be marked with a value of 99. The formula:  $RSSI \text{ [dBm]} = (2 \times ASU) - 113$ .

**ta** The timing advance. An integer in the range of 0 to 63 (optional).

## WCDMA

If the network is either WCDMA or any high-data-rate variant of it, the radio field should be specified as *wcdma*. This includes *UMTS*, *HSPA*, *HSDPA*, *HSPA+* and *HSUPA*.



Example:

```
{
  "radio": "umts",
  "mcc": 123,
  "mnc": 123,
  "lac": 12345,
  "cid": 123456789,
  "psc": 123,
  "signal": -68,
  "asu": 48
}
```

**radio (required)** The string *wcdma*.

**mcc (required)** The mobile country code. An integer in the range of 1 to 999.

**mnc (required)** The mobile network code. An integer in the range of 0 to 999.

**lac (required)** The location area code. An integer in the range of 1 to 65533. According to TS 24.008 10.5.3 both 0 and 65534 are reserved values indicating a deleted state.

**cid (required)** The cell id. An integer in the range of 1 to 268435455.

**psc** The primary scrambling code as an integer in the range of 0 to 511 (optional).

**signal** The received signal code power (RSCP) in dBm, typically in the range of -25 to -121 (optional).

**asu** The arbitrary strength unit. An integer in the range of -5 to 91 (optional). Unknown values can be marked with a value of 255. The formula:  $RSCP [dBm] = ASU - 116$ .

A special case exists for WCDMA cells, to send data about neighboring cells. For these it is acceptable to omit the lac and cid fields if at the same time a valid psc field is submitted.

## LTE

Example:

```
{
  "radio": "lte",
  "mcc": 123,
  "mnc": 123,
  "lac": 12345,
  "cid": 12345,
  "psc": 123,
  "signal": -69,
  "asu": 71,
  "ta": 10
}
```

**radio (required)** The string *lte*.

**mcc (required)** The mobile country code. An integer in the range of 1 to 999.

**mnc (required)** The mobile network code. An integer in the range of 0 to 999.

**lac (required)** The tracking area code. An integer in the range of 1 to 65533. According to TS 24.301 9.9.3.32 both 0 and 65534 are reserved values indicating a deleted state.

**cid (required)** The cell identity. An integer in the range of 1 to 268435455.

**psc** The physical cell id as an integer in the range of 0 to 503 (optional).

**signal** The received signal strength (RSRP) in dBm, typically in the range of -43 to -140 (optional).

**asu** The arbitrary strength unit. An integer in the range of 0 to 97 (optional). The formula:  $\text{RSRP [dBm]} = \text{ASU} - 140$ .

**ta** The timing advance. An integer in the range of 0 to 63 (optional).

A special case exists for LTE cells, to send data about neighboring cells. For these it is acceptable to omit the lac and cid fields if at the same time a valid psc field is submitted.

## CDMA

We classify CDMA networks and their EVDO variants as *cdma*. This includes *1xRTT*, *CDMA*, *eHRPD*, *EVDO\_0*, *EVDO\_A*, *EVDO\_B*, *IS95A* and *IS95B*.

---

**Note:** Since CDMA networks are rare and only available in a minority of regions, this project does not support them.

---

`ichnaea.models.config`

`ichnaea.models.constants`

`ichnaea.models.content`

`ichnaea.models.mac`

`ichnaea.models.observation`

`ichnaea.models.sa_types`

`ichnaea.models.schema`

`ichnaea.models.station`

`ichnaea.models.wifi`

`ichnaea.queue`

`ichnaea.scripts`

`ichnaea.scripts.datamap`

`ichnaea.scripts.initdb`

`ichnaea.scripts.region_json`

`ichnaea.util`

`ichnaea.webapp``ichnaea.webapp.app``ichnaea.webapp.config``ichnaea.webapp.monitor``ichnaea.webapp.settings``ichnaea.webapp.view``ichnaea.webapp.worker`

## 2.7 Glossary

### Cell-ID

#### Cell ID

#### Cell IDs

**CGI** **Cell ID** describes both a globally unique identifier for any logical cell network, as well as an approach to locate devices based on prior knowledge of the positions of these cell networks. Sometimes the term cell global identity (CGI) can also be found. See also [WPS](#).

**CID** A term describing the GSM or WCDMA cell id, LTE cell identity or CDMA base station id. All these form the last part of the globally unique [Cell ID](#).

### decimal degree

**decimal degrees** [Decimal degrees](#) express latitude and longitude coordinates as decimal fractions, for example *10.1234567*.

**MLS** The [Mozilla Location Service](#) is an instance of the ichnaea software hosted by Mozilla.

### observation

**observations** An observation describes the data collected about a single [station](#) identified by its unique global identifier, additional data about the [station](#) like signal strength readings and [report](#) data about the position of the [station](#).

### OpenCellID

**OCID** [OpenCellID](#) is a collaborative project to create a free worldwide database of [Cell IDs](#).

### report

**reports** A report describes the data collected in a single reading, consisting of data about the position and movement at the time of taking the reading and data about multiple [stations](#) observable at the time. For example one report could contain information about one cell network and 10 WiFi networks.

### station

**stations** A term referring to any radio signal emitting stationary device, or the radio network it emits. Examples of what we call stations are WiFi access points / WiFi network, cell towers / cell networks and Bluetooth Beacons / Bluetooth LE networks.

### Web Mercator

**WSG84** [WSG 84 Web Mercator](#) refers to the geographic map projection used throughout this project. The latitude and longitude values use [WSG 84](#) as the coordinate reference system.

### WiPS

**WPS** [Wi-Fi based positioning system](#) describes a system of using prior knowledge about the location of WiFi networks, identified by their globally unique BSSID/MAC address, to position devices. See also [Cell ID](#).

## 2.8 Changelog

### 2.8.1 Changelogs

#### Changelog

##### 1.5 (unreleased)

#### Untagged

#### Migrations

#### Changes

##### 20170606213300

#### Changes

- Update Python dependencies.
- Adjust the text on the download and stats pages.

##### 20170314161700

#### Migrations

- d2d9ecb12edc: Add modified index on *datamap\_\** tables.

#### Changes

- No longer use secondary cell tables during lookups.
- Remove continous cell import functionality.
- Relax GeoIP database check to allow *GeoLite2-City* databases.
- Update region specific statistics once per day.

## 20170223205700

### Migrations

- cad2875fd8cb: Add *store\_sample\_\** columns to *api\_key* table.

### Changes

- Add in-memory API key cache.

## 20161123180300

### Changes

- Move numpy install into build step.

## 20161123162300

### Changes

- Downgraded pip and virtualenv.

## 20161123125700

### Changes

- Remove */leaders* HTTP redirects.
- Replace the */apps* page with a link to the Wiki.
- Add a workaround for a bug in the RTree library.

## 20160810110400

### Migrations

- 385f842b2526: Add *allow\_transfer* column to *api\_key* table.

### Changes

- Use a docker volume for the css/js build pipeline.

## 20160809152200

### Changes

- Add a new *location\_dump* command line script to dump/export networks.
- #491: Add an alternate homepage if the web content is disabled.
- Reduce datamap detail.
- No longer log sentry messages for client errors.
- Replace station data for old stations with new conflicting observations.
- Make statsd and sentry optional service dependencies.
- Disable the web site content by default.
- #490: Limit full cell export to cells which were modified in the last year.
- Use the accuracy\_radius field from the GeoIP database.
- Remove ipf fallback from responses for queries based only on GeoIP data.

## 20160525130100

### Migrations

- Remove the celery section from the config file, add the web section.

### Changes

- Lower maximum accuracy values returned from locate queries.
- Accept observations with slightly worse accuracy values.
- The website content can be disabled via a setting in the config file.
- Make map related settings configurable via the config file.
- Use the first of the month, to display stats for the entire month.
- Calculate and display stats for today.
- Preserve or derive both WiFi channel and frequency values.

## 20160429092200

### Changes

- Allow report submissions without any position data.
- #485: Fix development documentation.

## 20160427100000

### Migrations

- 1bdf1028a085: Extend export config table.
- 6ec824122610: Add export config table.
- 4255b858a37e: Remove user/score tables.
- In service checks, change anything that checked the `/__heartbeat__` view to check `/__lbheartbeat__` instead. Change `/__monitor__` to `/__heartbeat__`.

### Changes

- Be more explicit about closing socket connections.
- Use GNSS observations to replace purely query-based stations.
- Use query observations to confirm, blocklist and insert new stations.
- Configure release for raven/sentry client.
- Change heartbeat/monitor view to lbheartbeat/heartbeat views.
- Update last\_seen column on each station update.
- Use Vincenty formula for lat/lon additions.
- Use Vincenty instead of Haversine formula for distance calculations.
- Take age into account during locate lookups.
- Filter out observations with too little weight.
- Take age and speed into account in observation weights.
- Pass queries into internal data pipeline.
- Allow stations to be blocklisted once per 30 days of their lifespan.
- Normalize age fields for internal observations to GPS time difference.
- Add stricter validation of radio, source and timestamp fields.
- Pass pressure and source data into internal data pipeline.
- Read export config from database instead of ini file.

## 20160412083700

### Migrations

- 27400b0c8b42: Drop api\_key log columns.
- 88d1704f1aef: Drop cell\_ocid table.

### Changes

- Remove intermediate schedule\_export\_reports task.
- #456: Retire old leaderboard.
- Remove intermediate upload\_report task.

### 20160401185900

### Changes

- Downgrade numpy to 1.10.4 due to build failures.

### 20160401110200

### Migrations

- e23ba53ab89b: Add sharded OCID cell tables.
- fdd0b256cecc: Add fallback options to API key table.

### Changes

- Tag location fallback metrics with the fallback name.
- #484: Allow per API key fallback configuration.
- Document and forward age argument through all layers of abstraction.
- Limit the columns loaded for API keys.
- Prevent errors when receiving invalid timestamps.

### 20160323102800

### Changes

- #456: Deprecate weekly leaderboard.
- Remove the implied metadata setting from the config file.
- Enable extended metrics for all API keys.
- Speed up full cell export.
- Rename internal blue/wifi observation key to mac.
- Removed migrations before version 1.4.



## Changelog 1.4

### 1.4 (2016-03-03)

#### 20160303094100

##### Changes

- Deprecate hashkey and internaljson logic.
- Improve readability of downloads page.
- Restrict valid characters in API keys.
- Take signal strength into account for location queries.
- Decrease database session times in data tasks.
- Retry station updates on deadlocks and lock timeouts.
- Simplify and speed up InternalUploader.

#### 20160218132400

##### Changes

- Display region specific BLE statistics.
- Remove geodude compatibility API again.
- Avoid intermediate Redis task round-trip.
- Queue data for up to 24 hours.
- Simplify colander schemata.
- Update dependencies.

#### 20160202154300

##### Changes

- Avoid fancy syntax in build requirements.

#### 20160202101600

##### Migrations

- 4b11500c9014: Add Bluetooth region stat.
- b247526b9501: Add sharded Bluetooth tables.
- 0987336d9d63: Add weight and last\_seen columns to station tables.
- 44e1b53944ee: Remove old cell tables.

## Changes

- #476: Emit basic request metrics for region API.
- Remove internal API key human readable metric names.
- Accept and use Bluetooth networks in public HTTP APIs.
- Weight observations by their accuracy and signal strength values.
- Add stricter validation of asu, signal and ta values.
- Restrict observations to maximum accepted accuracy values.
- Allow queries to the fallback source if the combined score is too low.
- #151: Choose best position result based on highest combined score.
- #481: Fix broken cell export.
- #151: Choose best region result based on highest combined score.
- #371: Extend region API to use wifi data.
- #371: Extend region API to use cell area data.
- #151: Choose best region result based on highest score.
- Remove migrations and tests for 1.2 to 1.3 upgrade.
- Enable *shapely.speedups* to speed up GeoJSON parsing.
- Ship buffered region file with the code.
- Stop forwarding client IP address to data pipeline.

## 20160112143200

### Migrations

- 9743e7b8a17a: Add allow\_locate column to API key table.
- 5d245a440c6f: Remove unused user email field.
- d350610e27e: Shard cell table.
- 40d609897296: Add sharded cell tables.
- The command line for starting gunicorn has changed. The *-c* option now needs a *python:* prefix and has to be *-c python:ichnaea.webapp.settings*.

### Changes

- #478: Restrict some API keys from using the locate API.
- Register OCID import tasks based on the configuration file.
- #471: Remove sentence about Firefox OS.
- #477: Decrease cell maximum radius to 100 km.
- Use sharded cell tables.
- Keep separate rate limits per API version.

- Update to latest versions of dependencies.

## 20151118134500

### Migrations

- 91fb41d12c5: Drop mapstat table.

### Changes

- #469: Update to static tabzilla.
- #468: Add CORS headers and support OPTIONS requests.
- #467: Implement geodude compatibility API.
- #151: Choose best WiFi cluster based on a data quality score.
- Use up to top 10 WiFi networks in WiFi location.
- Use proper agglomerative clustering in WiFi clustering.
- Remove arithmetic/hamming distance analysis of BSSIDs.
- Accept and forward WiFi SSID's in public HTTP API's.

## 20151105120300

### Migrations

- 78e6322b4d9: Copy mapstat data to sharded datamap tables.
- 4e8635b0f4cf: Add sharded datamap tables.

### Changes

- Use new sharded datamap tables.
- Parallelize datamap CSV export, Quadtree generation and upload.
- Introduce upper bound for cell based accuracy numbers.
- Fix database lookup fallback in API key check.
- Switch randomness generator for data map, highlight more recent additions.
- Update to latest versions of lots of dependencies.

## 20151021143400

### Migrations

- 450f02b5e1ca: Update cell\_area regions.
- 582ef9419c6a: Add region stat table.

- 238aca86fe8d: Change cell\_area primary key.
- 3fd11bfaca02: Drop api\_key log column.
- 583a68296584: Drop old OCID cell/area tables.
- 2c709f81a660: Rename cell/area columns to radius/samples.

### Changes

- Maintain *block\_first* column.
- Introduce upper bound for Wifi based accuracy numbers.
- Provide better GeoIP accuracy numbers for cities and subdivisions.
- Fix cell queries containing invalid area codes but valid cids.
- #242: Add WiFi stats to region specific stats page.
- Add update\_statregion task to maintain region\_stat table.
- Update to latest versions of alembic, coverage, datadog, raven and requests.

## 20151013115000

### Migrations

- 33d0f7fb4da0: Add api\_type specific logging flags to api keys.
- 460ce3d4fe09: Rename columns to region.
- 339d19da63ee: Add new cell OCID tables.
- All OCID data has to be manually imported again into the new tables.

### Changes

- Add new *fallback\_allowed* tag to locate metrics.
- Calculate region radii based on precise shapefiles.
- Use subunits dataset to preserve smaller regions.
- Use GENC codes and names in GeoIP results.
- Consider more responses as high accuracy.
- Change internal names to refer to region.
- Change metric tag to region for region codes.
- Temporarily stop using cell/area range in locate logic.
- Discard too large cell networks during import.
- Use mcc in region determination for cells.
- Use new OCID tables in the entire code base.
- Use the intersection of region codes from GENC and our shapefile.
- Avoid base64/json overhead for simple queues containing byte values.

- Maintain a queue TTL value and process remaining data for inactive queues.
- Remove hashkey functionality from cell area models.
- Remove non-sharded update\_wifi queue.
- Merge scan\_areas/update\_area tasks into a single new update\_cellarea task.
- Remove backwards compatible tasks and area/mapstat task processing logic.
- Update to latest versions of bower, clean-css and uglify-js.
- Update to latest versions of cryptography, Cython, kombu, numpy, pyasn1, PyMySQL, requests, Shapely, six and WebOb.

## 20150928100200

### Migrations

- 26c4b3a7bc51: Add new datamap table.
- 47ed7a40413b: Add cell area id columns.

### Changes

- Improve locate accuracy by taking station circle radius into account.
- Split out OCID cell area updates to their own queue.
- Switch mapstat queue to compact binary queue values.
- Speed up update\_area task by only loading required cell columns.
- Validate all incoming reports against the region areas.
- Add a precision reverse geocoder for region lookups.
- Add a finer grained region border file in GeoJSON format.
- Shard update\_wifi queue/task by the underlying table shard id.
- Update datatables JS library and fix default column ordering.
- Switch to GENC dataset for region names.
- #372: Add geocoding / search control to map.
- Support the new *considerIp* field in the geolocate API.
- #389: Treat accuracy, altitude and altitudeAccuracy as floats.
- Speed up /stats/regions by using cell area table.
- Use cell area ids in update\_cellarea task queue.
- Enable country level result metrics.
- Removed migrations before version 1.2.
- Update to latest versions of numpy, pytz, raven, rtree and Shapely.

## Changelog 1.3

### 1.3 (2015-09-16)

#### 20150916130500

##### Migrations

- b24dbb9ccfe: Remove CDMA networks.
- 18d72822fe20: Remove wifi table.

##### Changes

- Stop importing and exporting CDMA networks.
- #222: Maintain a country/region code estimate for new wifi networks.
- Add new *location\_load* script to load cell dumps into a local db.
- Remove obsolete *remove\_wifi* task.
- Update to latest versions of certifi, cryptography, coverage and Cython.

#### 20150903095100

##### Migrations

- Manually run the wifi migration script in *scripts/migrate.py*.

##### Changes

- Stop using the wifi table.
- Update to latest versions of datadog, greenlet, mako and raven.

#### 20150828125400

##### Changes

- Fix bug in *block\_count* station update routine.

#### 20150827110300

##### Migrations

- c1efc747c9: Remove unused *api\_key* email/description columns.
- 4f12bf0c0828: Remove standalone wifi blocklist table.

## Changes

- Insert new wifi networks into sharded tables.
- Factor out more of the *aggregate position* logic.
- Optimize gzip compression levels.
- Add new celery queues (*celery\_cell*, *celery\_ocid*, *celery\_wifi*).
- Remove extra *internal\_dumps* call from insert task.
- Add a source tag for successful result metrics.
- Update data tables and stats/regions page.
- Setup Cython support and convert geocalc centroid and distance functions.
- Optimize OCID cell data export and import.
- Return multiple results from MCC based country source.
- Move best country result selection into searcher logic.
- Update to latest versions of alembic, cffi, cryptography, coverage, cython, hiredis, numpy, pip and scipy.

## 20150813105600

### Changes

- Use *data\_accuracy* as the criteria to decide if more locate sources should be consulted.
- Use both old and new wifi tables in locate logic.
- Add a new *\_\_version\_\_* route.
- Cache Wifi-only based fallback position results.
- Don't rate limit cache lookups for the fallback position source.
- Retry outbound connections once to counter expired keep alive connections.

## 20150806105100

### Migrations

- 2127f9dd0ed7: Move wifi blocklist entries into wifi shard tables.
- 4860cb8e54f5: Add new sharded wifi tables.
- The structure of the application ini file changed and the *ichnaea* section was replaced by a number of new more specific sections.

### Changes

- Enable SSL verification for outbound network requests.
- Add new metrics for counting unique IPs per API endpoint / API key.
- Enable locate source level metrics.

- #457: Fix cell export to again use UMTS as the radio type string.
- Optimize various tasks by doing batch queries and inserts.
- Avoid using a metric tag called *name* as it conflicts with a default tag.
- Deprecate *insert\_measure\_\** tasks.
- Move new station score bookkeeping into *insert\_measures* task.
- Update to latest version of datadog.

### 20150730143600

#### Changes

- Make report and observation drop metrics more consistent.

### 20150730111000

#### Migrations

- The statsd configuration moved from the *statsd\_host* option in the application ini file into its own section called *statsd*.

#### Changes

- Move blacklist and station creation logic into *update\_station* tasks.
- Add new *ratelimit\_interval* option to *locate:fallback* section.
- Set up a HTTPS connection pool used by the fallback source.
- Disable statsd request metrics for static assets.
- Let all internal data pipeline metrics use tags.
- Let all public API and fallback source metrics use tags.
- Let task, datamaps, monitor and HTTP counter/timer metrics use tags.
- Add support for statsd metric tagging.
- Use colander to map external to internal names in submit schemata.
- Add dependencies *pyopenssl*, *ndg-httpsclient* and *pyasn1*.
- Switch to datadog statsd client library.
- Consider Wifi based query results accurate enough to satisfy queries.
- Stop maintaining separate Python dependency list in *setup.py*.
- #433: Move GeoIP lookup onto the query object.
- #433: Add new detailed query metrics.
- Use a colander schema to describe the outbound fallback provider query.
- Set up and configure locate searchers and providers once during startup.



- Move all per-query state onto the locate query instance.
- Split customjson into internal and external pretty float version.
- Update to latest versions of alembic, setproctitle, simplejson and SQLAlchemy.

## Changelog 1.2

1.2 (2015-07-15)

20150716174000

### Changes

- Add a database migration test from a fresh SQL structure dump.

### Migrations

- 1a320a751cf: Remove observation tables.

### Changes

- #395: Move *incomplete\_observation* logic onto colander schema.
- #287: Replace observation models with non-db-model classes.
- #433: Move query data validation into Query class.
- #433: Introduce a new *api.locate.query.Query* class.
- Handle any RedisError, e.g. TimeoutError and not just ConnectionErrors.
- Update to latest raven release and update transport configuration.
- Explicitly limit the cell cache key to its unique id parts.
- Add *fallback* key to all locate responses.
- #451: Properly test and reject empty submit requests.
- #376: Document the added home mcc/mnc fields.
- #419: Update geolocate docs to mention all GLS fields.

20150707130400

### Migrations

- 2e0e620ebc92: Remove id column from content models.

### Changes

- Add workaround for andymccurdy/redis-py#633.
- Unify v1 and v2 parse error responses to v2 format.
- Batch key queries depending on a per-model batch size.
- #192: Suggest a observation data retention period.
- Optimize mapstat and station data tasks.
- Switch to using bower for CSS/JS dependency management.
- Update to latest versions of all CSS and JS dependencies.
- Update to latest versions of geoip2, SQLAlchemy and unittest2.

### 20150616104200

#### Migrations

- 55db289fa497: Add content model composite primary keys.
- 14dbafc4fec2: Remove new\_measures indices.
- 19d6d9fbdb6b: Increase stat table value column to biginteger.

### Changes

- Fix locate errors for incomplete cell keys.
- Remove backwards compatibility code.

### 20150610103900

#### Migrations

- 38fde2949750: Remove measure\_block table.

### Changes

- #287: Remove table based location\_update tasks and old backup code.
- Adjust batch sizes for new update\_station tasks.
- Bugzilla 1172833: Use apolitical names on country stats page.
- #443: Reorganize internal module/classes.
- Update to latest version of SQLAlchemy.

**20150604164500****Changes**

- #446: Filter out incomplete csv cell records.
- #287: Switch location\_update tasks to new queue based system.
- #438: Add explicit fallback choices to geolocate API.
- Replace the last daily stats task with a queue based one.
- #440: Allow search/locate queries without a cell id.
- Update to latest versions of nose, simplejson and SQLAlchemy.

**20150528085200****Changes**

- #394: Replace magic schema values by *None*.
- #423: Add new public *v2/geosubmit* API.
- #242: Pass through submission IP address into the data pipeline.
- #242: Expose geoip database to async tasks.
- Make sure there are no unexpected raven messages left after each test.
- #434: Internal test only changes to test base classes.
- Update to latest versions of gevent and simplejson.

**20150522094900****Changes**

- #421: Pass through additional lookup data into the fallback query.
- #421: Cache cell-only lookups for fallback provider queries.
- #421: Add rate limiting to fallback provider.
- #421: Reordered data sources to prefer fallback over geoip responses.
- Fix api-key specific report upload counter.
- Add workaround for raven issue #608.
- Enable new stat counter tasks.
- #433: Remove the wifi specific query stats.
- Updated to latest version of alembic, celery, greenlet, kombu and pytz.

## **20150507103300**

### **Changes**

- Correct handling for requests without API keys.
- #421: Fix encoding of radioType in fallback queries.

## **20150505113200**

### **Migrations**

- e9c1224f6bb: Add allow\_fallback column to api\_key table.

### **Changes**

- #287: Move mapstat and score processing to separate queues/tasks.
- #287: Keep track of uploaded data via Redis stat counters.
- #287: Add new backup to S3 export target.
- #421: Add fallback geolocation provider.
- Deal with nan/inf floating point numbers in data submissions.
- Fixed upload issues for cell entries without any radio field.
- Updated to latest versions of certifi, greenlet, pyramid, raven and requests.

## **20150423105800**

### **Changes**

- Allow anonymous data submission via the geosubmit API.
- #425: Refactor internal API key logic.
- Updated to latest raven version, requires a Sentry 7 server.
- Updated to latests versions of billiard, pyramid and WebOb.

## **20150416111700**

### **Migrations**

- The command line invocation for the services changed, please refer to the deploy docs for the new syntax.

## Changes

- #423: Add a first version of an export job.
- Expose all config file settings to the runtime services.
- Move runtime related code into async/webapp sub-packages.
- #385: Configure Python's logging module.
- #423: Add a new queue system using the new geosubmit v2 internal format.
- Updated to latest versions of boto and maxminddb.

## 20150409120500

## Changes

- Make radio an internally required field.
- Don't validate radio fields in request side schema.
- #418: Remove country API shortcut implementation.
- Removed BBB code for old tasks and pre-hashkey queued values.
- Updated to latest versions of alabaster, boto, factory\_boy and pytz.

## 20150320100800

## Changes

- Remove the circus docs and example ini file.
- Remove the vaurien/integration tests.
- #416: Accept radioType inside the cellTowers mapping in geolocate queries.
- Updated to latest version of Sphinx and its new dependencies.
- Updated to latest versions of pyramid, requests, SQLAlchemy and statsd.

## 20150309175500

- Fix unittest2 version pin.

## 20150305122500

## Migrations

- 1d549c1d6cfe: Drop total\_measures index on station tables.
- 230bbf3fe044: Add index on mapstat.time column.
- 6527bee5ac1: Remove auto-inc id columns from cell related tables.
- 3b8d52a9eac4: Change score, stat and measure\_block enum columns to tinyint.

## **Changes**

- Replace heka-py-raven with a direct raven client.
- #319: Remove the per station ingress filtering.
- Allow partial cell ids in geolocate/geosubmit APIs.
- Removed the mixed locate/submit mode from the geosubmit API.
- #402: Avoid multiple validation of common report data fields.
- Add a new CellCountryProvider to allow country searches based on cell data.
- #406: Allow access to the country API via empty GET requests.
- Massive internal code refactoring and cleanup.
- Updated to latest versions of iso3166, pyramid and requests.

## **20150211113000**

## **Changes**

- Reestablish database connections on connection failures.

## **20150209110000**

## **Changes**

- Backup/delete all observation data except for the current day.
- Updated to latest versions of boto, Chameleon, gunicorn, jaraco.util, Mako, psutil, Pygments, pyzmq and WebTest.

## **20150203093000**

## **Changes**

- Specify statsd prefix in application code instead of heka config.
- Fix geoip country lookup for entries without countries.
- #274: Extend monitor view to include geoip db status.

## **20150127130000**

## **Migrations**

- 10542c592089: Remove invalid lac values.
- fe2cfea89f5: Change cell/\_blacklist tables primary keys.

## Changes

- #367: Tighten lac filtering to exclude 65534 (gsm) and 65535 (all).
- Remove alembic migrations before the 1.0 PyPi release.
- #353: Remove auto-inc id column from cell/\_blacklist tables.
- Add additional stats to judge quality of WiFi based queries.
- #390: Remove command line importer script.

## 20150122140000

### Migrations

- 188e749e51ec: Change lac/cid columns to signed integers.

## Changes

- #352: Switch to new maxmind v2 database format and libraries.
- #274: Add a new `__monitor__` endpoint.
- #291: Allow 32bit UMTS cell ids, tighten checks for CDMA and LTE.
- #311: On station creation optionally use previous blacklist time.
- #378: Use colander for internal data validation.
- Remove explicit queue declaration from celery base task.
- Updated to latest versions of alembic, boto, Chameleon, jaraco.util, mobile-codes, psutil, requests-mock, WS-GIPProxy2 and zope.deprecation.

## 20150105140000

### Migrations

- 48ab8d41fb83: Move cell areas into separate table.

## Changes

- Prevent non-countries from being returned by the country API.
- #368: Add per API key metrics for uploaded batches, reports and observations.
- Clarify metric names related to batches/reports/observations, add new `items.uploaded.batch_size` pseudo-timer and `items.uploaded.reports` counter.
- Introduce new internal `GeoIPWrapper.country_lookup` API.
- #343: Fall back to GeoIP for incomplete search requests.
- #349/#350: Move cell areas into new table.
- Give all celery queues a prefix to better distinguish them in Redis.

- #354: Remove scan\_lacs fallback code looking at new\_measures.
- Updated to latest versions of alembic, argparse, billiard, celery, colander, filechunkio, iso8601, kombu, PyMySQL, pytz, requests, six, WebTest and zope.interface.

## 20141218093500

- #371: Add new country API.

## 20141120130000

- Add api key specific stats to count best data lookup hits/misses.
- Validate WiFi data in location lookups earlier in the process.
- #312: Add email field to User model.
- #287: Move lac update scheduling to Redis based queue.
- #304: Auto-correct radio field of GSM cells with large cid values.
- Move responsibility for lac entry deletion into update\_lac task.
- Accept more ASU values but tighten signal strength validation.
- #305: Stricter range check for mnc values for non-CDMA networks.
- Add a convenience *session.on\_post\_commit* helper method.
- #17: Remove the unused code for cell backfill.
- #41: Explicitly allow anonymous data submissions.
- #335: Omit incomplete cell records from exports.
- Delete measures in batches of 10k rows in backup tasks.
- Re-arrange backup tasks to avoid holding db session open for too long.
- Report errors for malformed data in submit call to sentry.
- Report errors during backup job to sentry.
- #332: Fix session handling in map tiles generation.
- Updated to latest versions of argparse, Chameleon, irc, Pygments, pyramid, translationstring and unittest2.

## 20141103125500

- #330: Expand api keys and download sections.
- Close database session early in map tiles generation.
- Close database session early in export task to avoid timeout errors while uploading data to S3.
- Optimize cell export task and avoid datetime/unixtime conversions.
- Add an index on cell.modified to speed up cell export task.
- Updated to latest versions of boto, irc, pygeoip, pytz, pyzmq, simplejson and unittest2.



**20141030113700**

- Add play store link for Mozilla Stumbler to apps page.
- Updated privacy notice style to match general Mozilla style.
- Switch gunicorn to use a gevent-based worker.
- Clean last database result from connections on pool checkin.
- Close the database connections even if exceptions occurred.

**Changelog 1.1****1.1 (2014-10-27)****20141027122000**

- Lower DB pool and overflow sizes.
- Update Mozilla Stumbler screenshot.
- Update to new privacy policy covering both Fennec and Mozilla Stumbler.

**20141023094000**

- Updated Fennec link to point to Aurora channel.
- Renamed MozStumbler to Mozilla Stumbler, added new screenshot.
- Increase batch size for *insert\_measures\_wifi* task.
- Extend queue maximum lifetime for incoming reports to six hours.
- Extend observation task batching logic to apply to cell observations.
- #328: Let gunicorn start without a valid geoip database file.
- Extend the *make release* step to deal with Python files with incompatible syntax.
- Update to latest versions of configparser, greenlet, irc and pyzmq.

**20141016123300**

- Log gunicorn errors to stderr.
- #327: Add an anchor to the leaderboard table.
- Move the measure tables gauges to an hourly task.
- Fix initdb script to explicitly import all models.

**20141014161400**

- #311: Filter out location areas from unique cell statistics.
- Introduce a 10 point minimum threshold to the leaderboard.

- Change download page to list files with kilobytes (kB) sizes.
- #326: Quantize data maps image tiles via pngquant.
- Optimize file size of static image assets.
- Remove task modules retained for backwards compatibility.
- Update to latest version of SQLAlchemy.

### 20141009121300

- Add a task to monitor the last import time of OCID cells.
- Change api\_key rate limitation monitoring task to use shortnames.
- Small improvements to the manual importer script.
- #276: Fix bug in batch processing, when receiving more than 100 observations in one submission.
- Refactor some internals and move code around.
- Create a new *lbcheck* MySQL user in the *location\_initdb* command.
- Fix *monitor\_api\_key\_limits* task to work without api limit entries.
- #301: Schedule hourly differential imports of OCID cell data.
- Update to latest versions of boto, celery, iso3166, jaraco.util, requests and simplejson.

### 20141002103900

- #301: Add OCID cell data to statistics page.
- Allow a radio type of *lte* for the geolocate API. Relates to [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1010284](https://bugzilla.mozilla.org/show_bug.cgi?id=1010284).
- #315: Add a *show my location* control to the map.
- Reverse ordering of download files to display latest files first.
- Extend db ping to retry connections for *2003 connection refused* errors.
- Ignore more exception types in API key check, to continue degraded service in case of database downtimes.
- Switch from d3.js/rickshaw to flot.js and prepare graphs to plot multiple lines in one graph.
- Make country statistics table sortable.
- Remove auto-increment column from *ocid\_cell* table and make the radio, mcc, mnc, lac, cid combination the primary key. Also optimize the column types of the lac and cid fields.
- Update to latest versions of alembic, amqp, celery, configparser, cornice, greenlet, jaraco.util, kombu, protobuf, psutil, pytz, requests, six, Sphinx and WebTest.
- #301: Add code to do continuous updates of the OpenCellID data and add license note for OCID data.

### 20140904094000

- #308: Fixed header row in cell export files.

**20140901114000**

- #283: Add manual logic to trigger OpenCellID imports.
- Add Redis based caching for SQL queries used in the website.
- #295: Add a downloads section to the website and enable cell export tasks.
- Clarify api usage policy.
- Monitor api key rate limits and graph them in graphite.
- Update to latest versions of nose and simplejson.
- #282: Add a header row to the exported CSV files.

**20140821114200**

- #296: Trust WiFi positions over GeoIP results.
- Optimized SQL types of mnc, psc, radio and ta columns in cell tables.
- Update to latest versions of country-bounding-boxes, gunicorn and redis.
- #282: Added code to do exports of cell data, both daily snapshots as well as hourly diffs. Currently the automatic schedule is still disabled. This also adds a new modified column to the cell and wifi tables.

**20140812120000**

- Include links to blog and new @MozGeo twitter account.
- Update to latest version of alembic, boto, redis, simplejson and statsd.
- Add a monitoring task to record Redis queue length.
- Make a Redis client available in Celery tasks.
- #285: Update favicon, add touch icon and tile image.
- Only retain two days of observation data inside the DB.
- Fixed image tiles generation to generate up to zoom level 13 again.
- #279: Offer degraded service if Redis is unavailable.
- #72: Always log sentry messages for exceptions inside tasks.
- #53: Document testing approaches.
- #130: Add a test for syntactic correctness of the beat schedule.
- #27: Require sufficiently different BSSIDs in WiFi lookups. This reduces the chance of being able to look up a single device with multiple logical networks.

**20140730133000**

- Avoid using *on\_duplicate* for common update tasks of tables.
- Remove GeoIP country submission filter, as GeoIP has shown to be too inaccurate.
- #280: Relax the GeoIP country restriction and also trust the mcc derived country codes.

- #269: Improve search logic when dealing with multiple location areas.
- Correctly deal with multiple country codes per mcc value and don't restrict lookups to one arbitrary of those countries.
- Fix requirement in WiFi lookups to really only require two networks.
- Added basic setup for documenting internal code API's and use the geocalc and service.locate modules as first examples.
- Initialize the application and outbound connections as part of the gunicorn worker startup process, instead of waiting for the first request and slowing it down.
- Switch pygeoip module to use memory caching, to prevent errors from changing the datafile from underneath the running process.
- Introduce 10% jitter into gunicorn's max\_requests setting, to prevent all worker processes from being recycled at once.
- Update gunicorn to 19.1.0 and use the new support for config settings based on a Python module. The gunicorn invocation needs to include `-c ichnaea.gunicorn_config` now and can drop various of the other arguments.
- Updated production Python dependencies to latest versions.
- Updated supporting Python libraries to latest versions.
- Update clean-css to 2.2.9 and uglify-js to 2.4.15.
- Update d3.js to 3.4.11 and jQuery 1.11.1.
- Changed graphs on the stats page to show a monthly count for the past year, closes [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1043386](https://bugzilla.mozilla.org/show_bug.cgi?id=1043386).
- Update rickshaw.js to 1.5.0 and tweak stats page layout.
- Add MLS logo and use higher resolution images where available.
- Always load cdn.mozilla.net resources over https.
- Updated deployment docs to more clearly mention the Redis dependency and clean up Heka / logging related docs.
- Split out circus and its dependencies into a separate requirements file.
- Remove non-local debug logging from map tiles generation script.
- Test all additional fields in geosubmit API and correctly retain new *signalToNoiseRatio* field for WiFi observations.
- Improve geosubmit API docs and make them independent of the submit docs.
- Update and tweak metrics docs.
- Adjust Fennec link to point to Fennec Nightly install instructions. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1039787](https://bugzilla.mozilla.org/show_bug.cgi?id=1039787)

### 20140715114000

- Adjust beat schedule to update more rows during each location update task.
- Let the backup tasks retain three full days of measures in the DB.
- Remove the database connectivity test from the heartbeat view.

## **Changelog 1.0**

### **1.0 (2014-07-14)**

- Initial production release.

### **0.1 (2013-11-22)**

- Initial prototype.



## CHAPTER 3

---

### Development

---

All source code is available on [github](#) under `ichnaea`.

The developers of `ichnaea` can frequently be found on the [Mozilla IRC network](#) in the `#geo` channel.





## CHAPTER 4

---

### License

---

The `ichnaea` source code is offered under the Apache License 2.0.



### C

Cell ID, [47](#)  
Cell IDs, [47](#)  
Cell-ID, [47](#)  
CGI, [47](#)  
CID, [47](#)

### D

decimal degree, [47](#)  
decimal degrees, [47](#)

### M

MLS, [47](#)

### O

observation, [47](#)  
observations, [47](#)  
OCID, [47](#)  
OpenCellID, [47](#)

### R

report, [47](#)  
reports, [47](#)

### S

station, [47](#)  
stations, [47](#)

### W

Web Mercator, [47](#)  
WiPS, [48](#)  
WPS, [48](#)  
WSG84, [48](#)