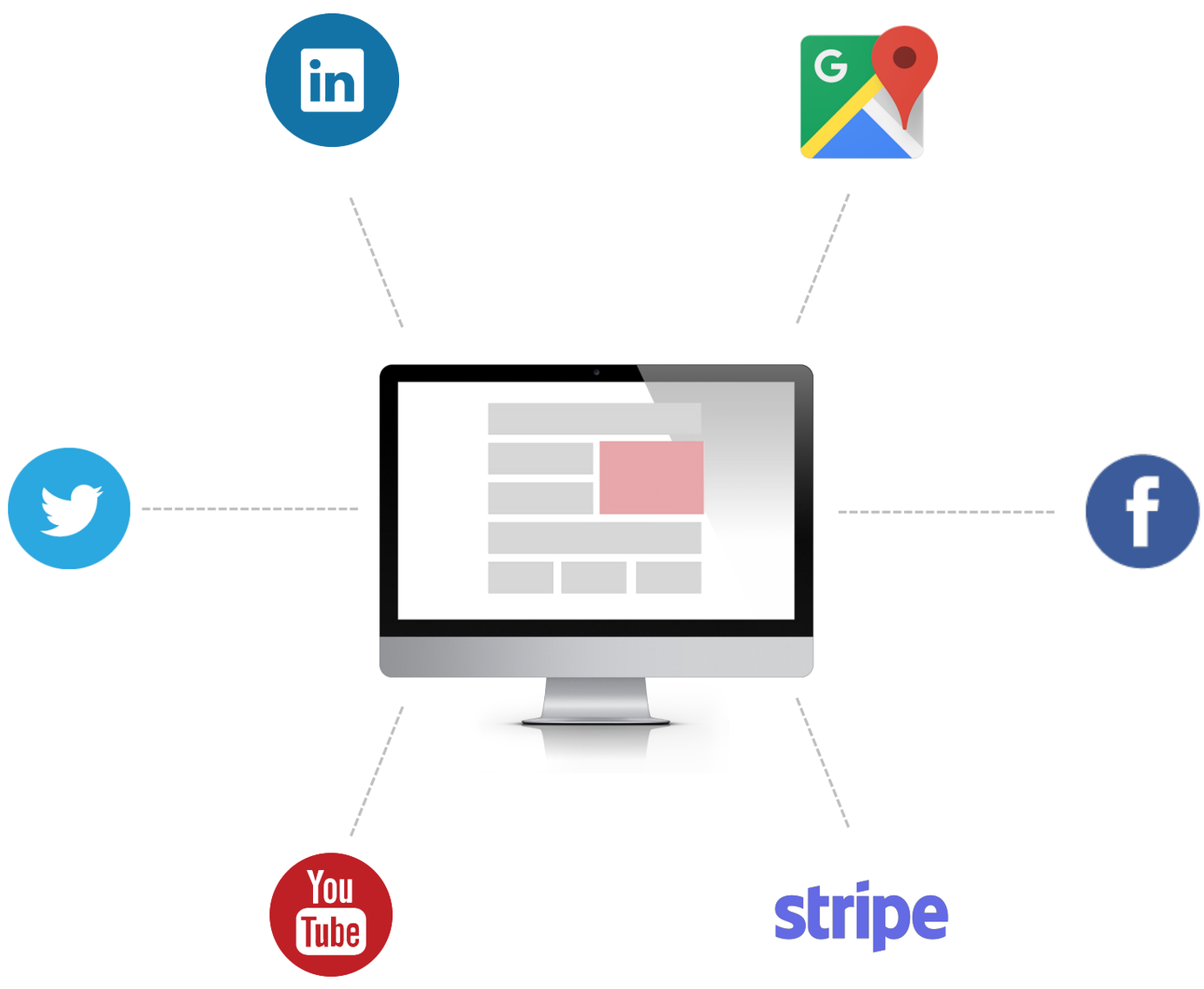
Complex Presence Constraints in Web Applications

Deliverable 4.1.1

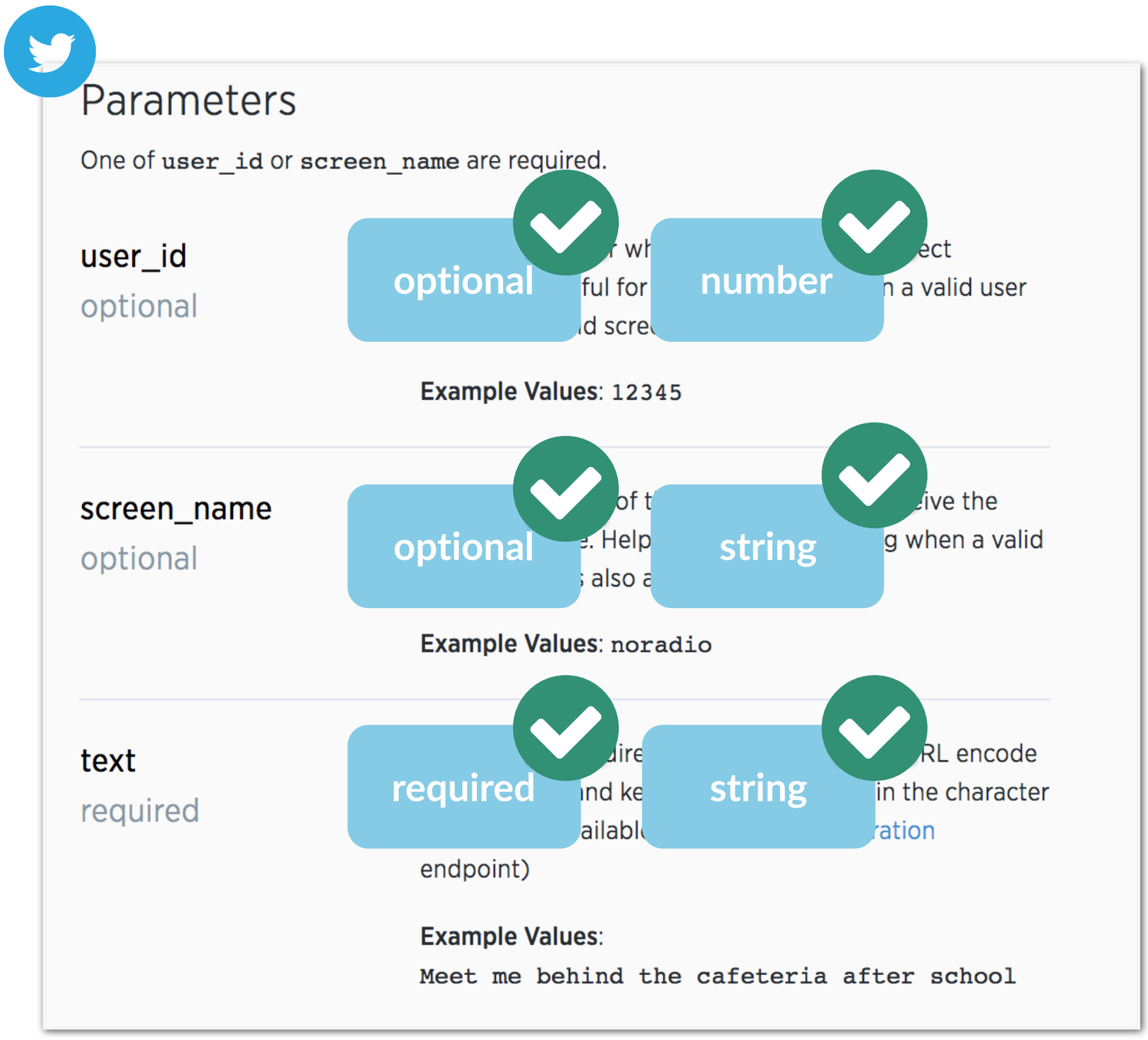
Nathalie Oostvogels

VRIJE UNIVERSITEIT BRUSSEL

Third party web services



Web API specifications

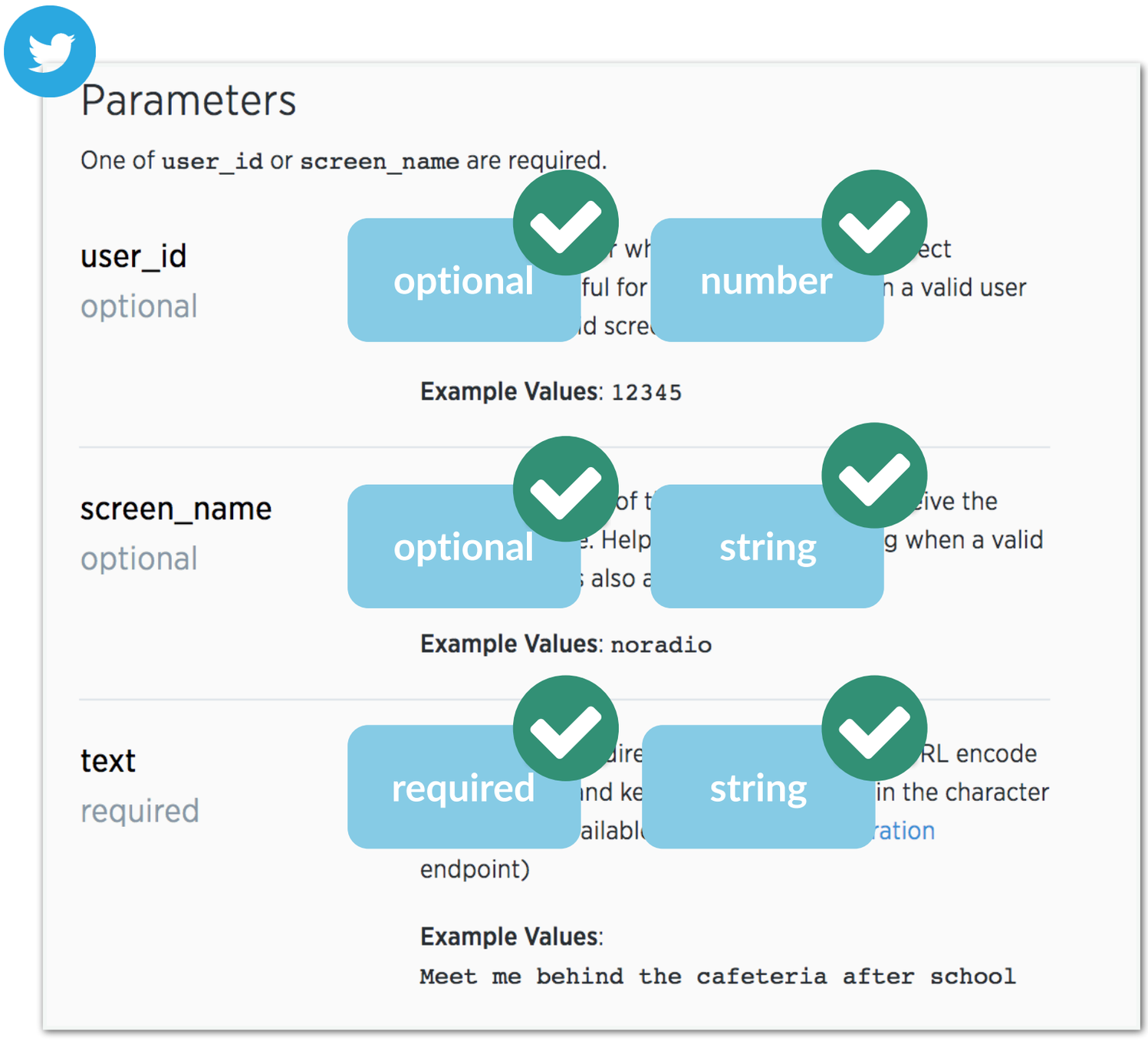


Verifying the constraints

```
export module Twitter{
    export interface PrivateMessage {
        user_id?: number;
         screen_name?: string;
         text: string;
   }

    export function sendPrivateMessage(body: PrivateMessage){
        // ...
    }

}
```

TypeScript

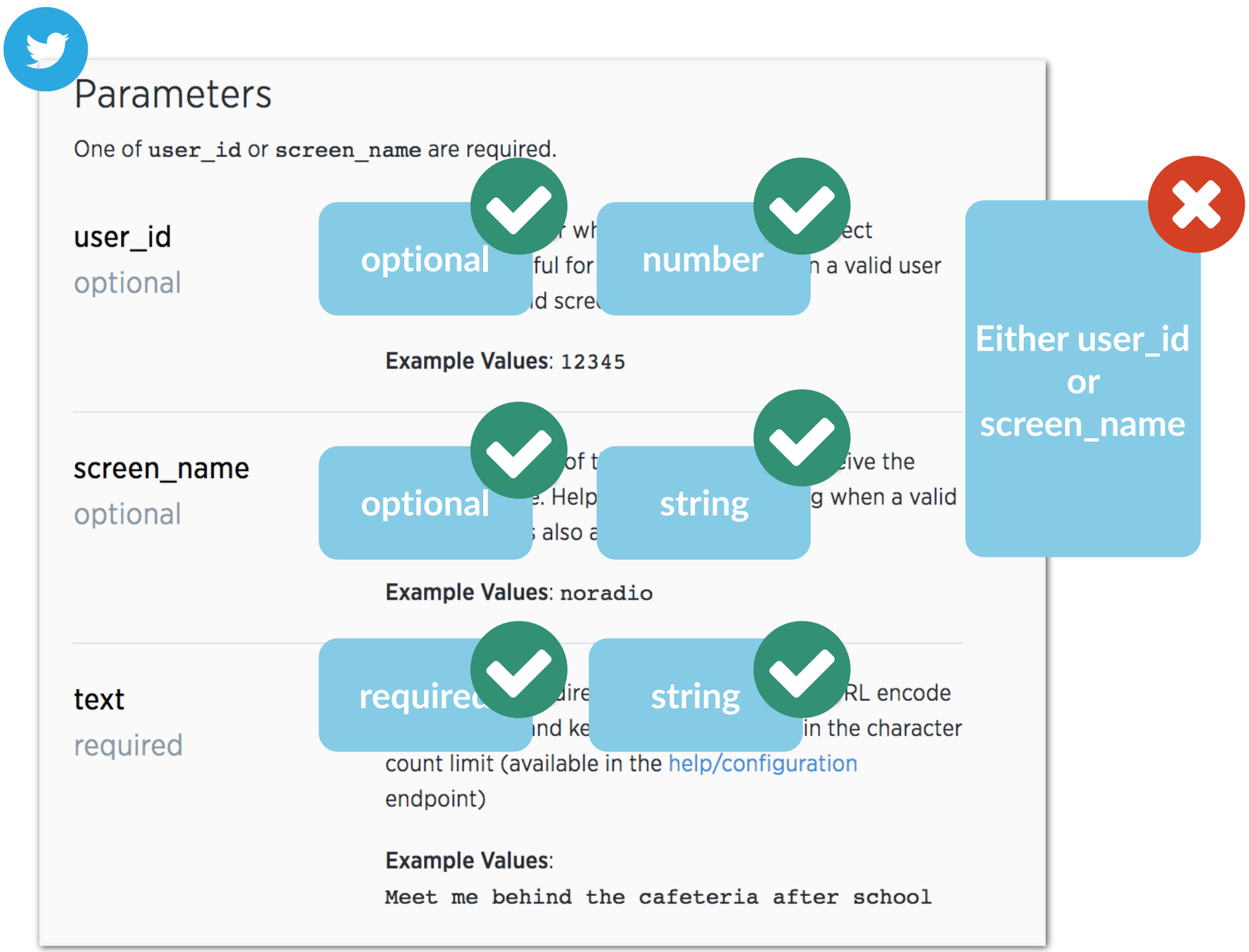
Web API specifications



Web API specifications

- “Incompatible parameters specified in the request”
- “Some co-ordinate parameters were blank”
- “You must specify either a list ID or a slug and owner”

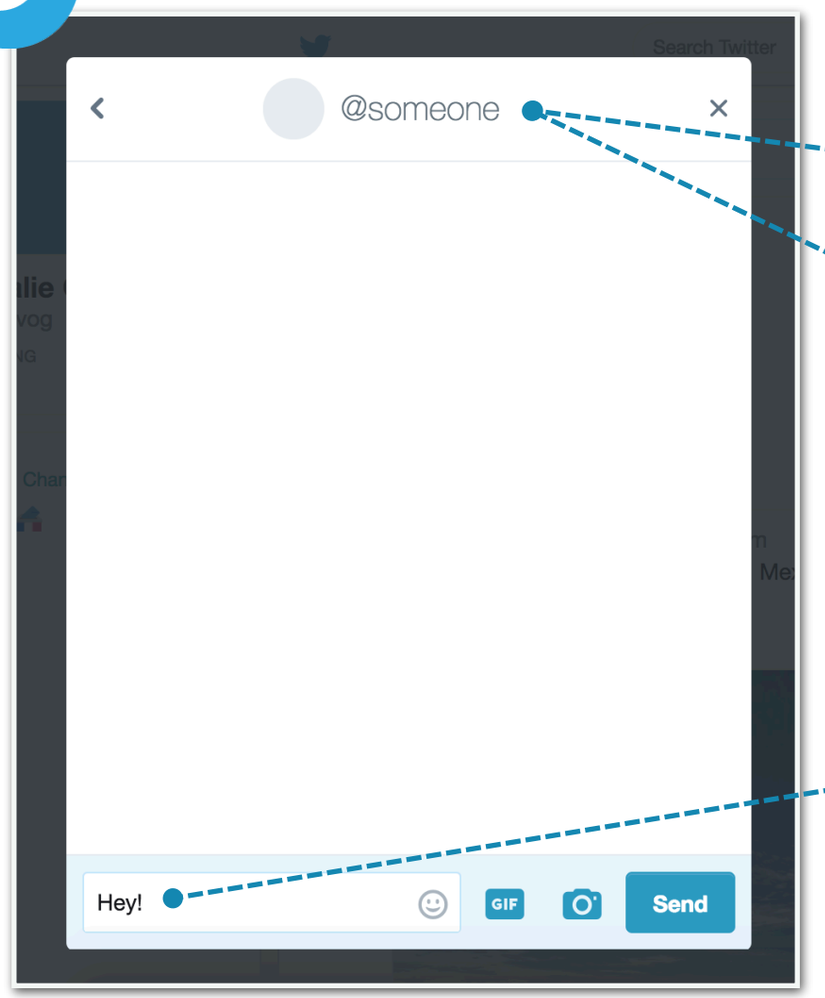
Web API specifications



Inter-parameter constraints

Exclusive constraints

- **exactly one** of a set of parameters is required



Parameters

One of `user_id` or `screen_name` are required.

user_id
optional

The ID of the user who should receive the direct message. Helpful for disambiguating when a valid user ID is also a valid screen name.

Example Values: `12345`

screen_name
optional

The screen name of the user who should receive the direct message. Helpful for disambiguating when a valid screen name is also a user ID.

Example Values: `noradio`

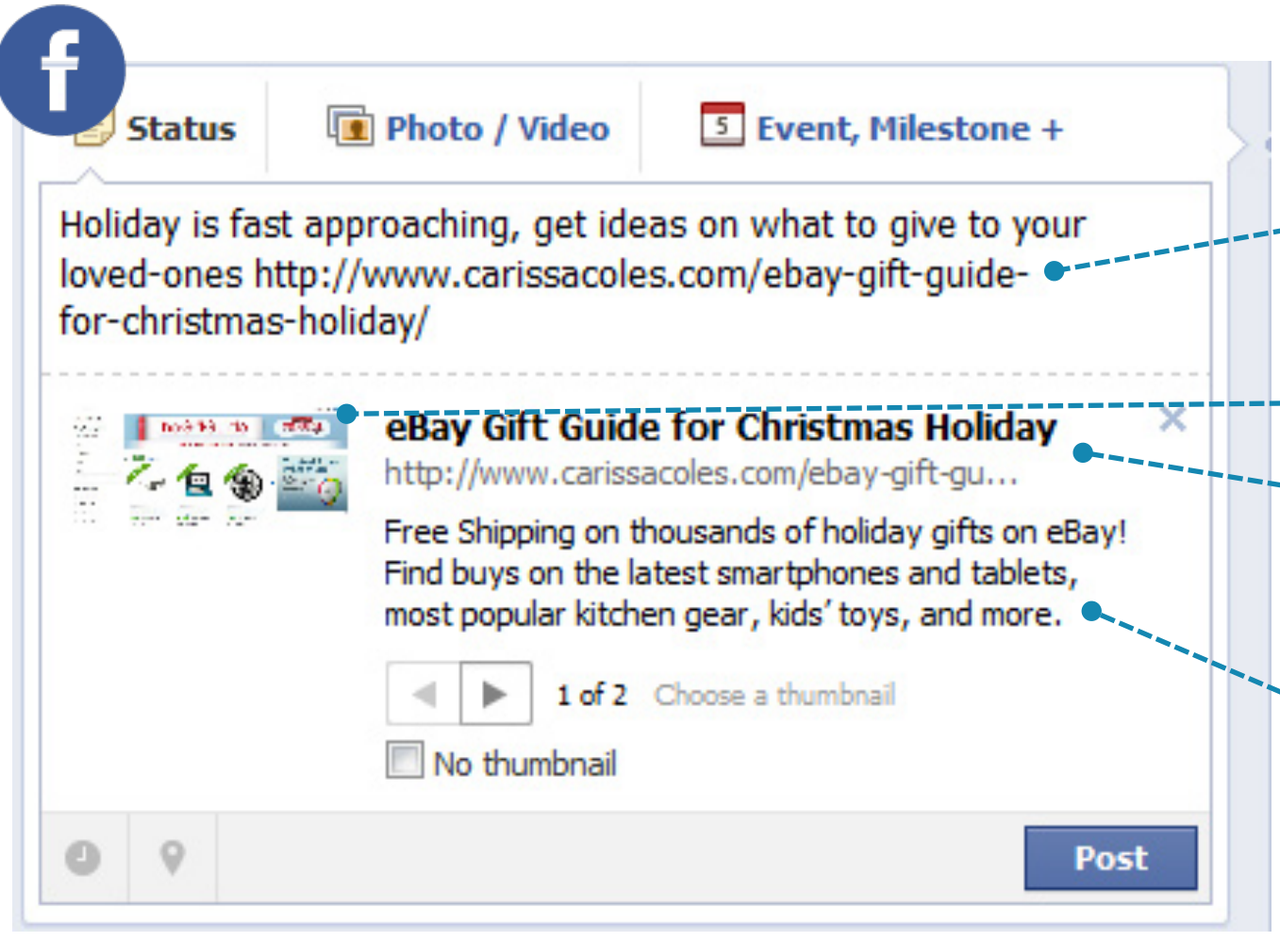
text
required

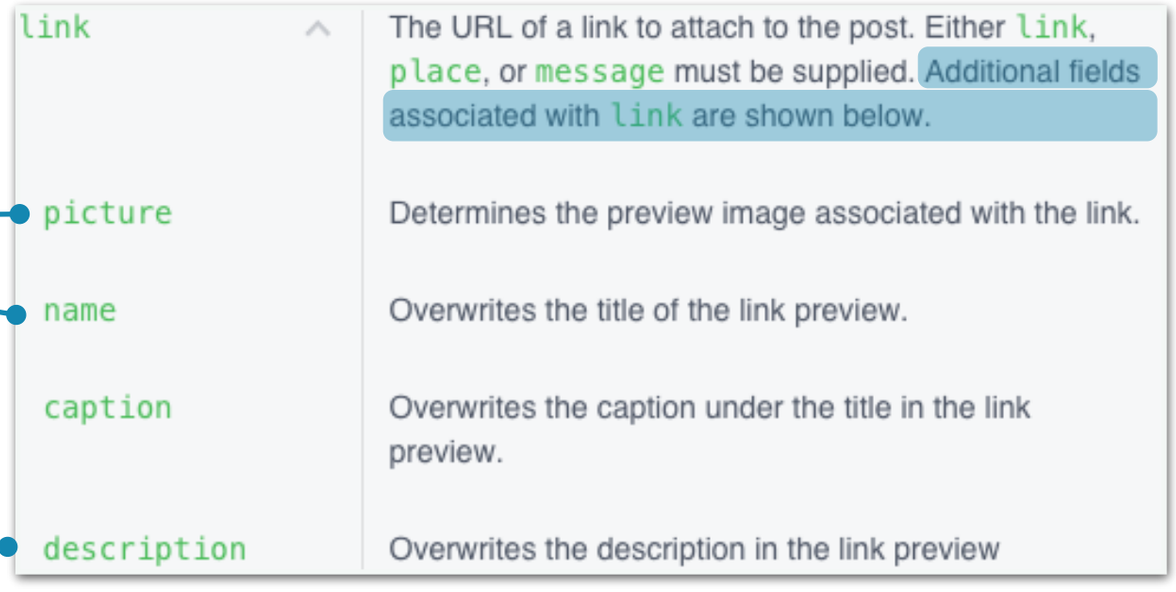
The text of your direct message. Be sure to URL encode as necessary, and keep the message within the character count limit (available in the help/configuration endpoint)

Example Values:
`Meet me behind the cafeteria after school`

Dependent constraints

- constraints on a parameter **depend** on a property of another parameter





Dependent constraints

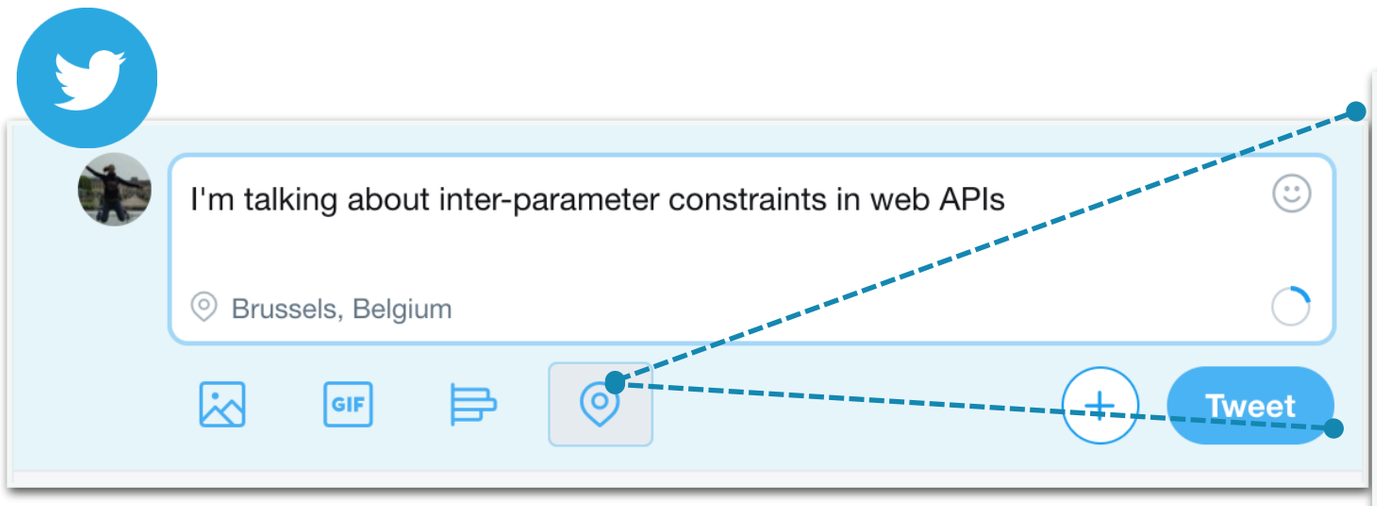
- constraints on a parameter **depend** on a property of another parameter

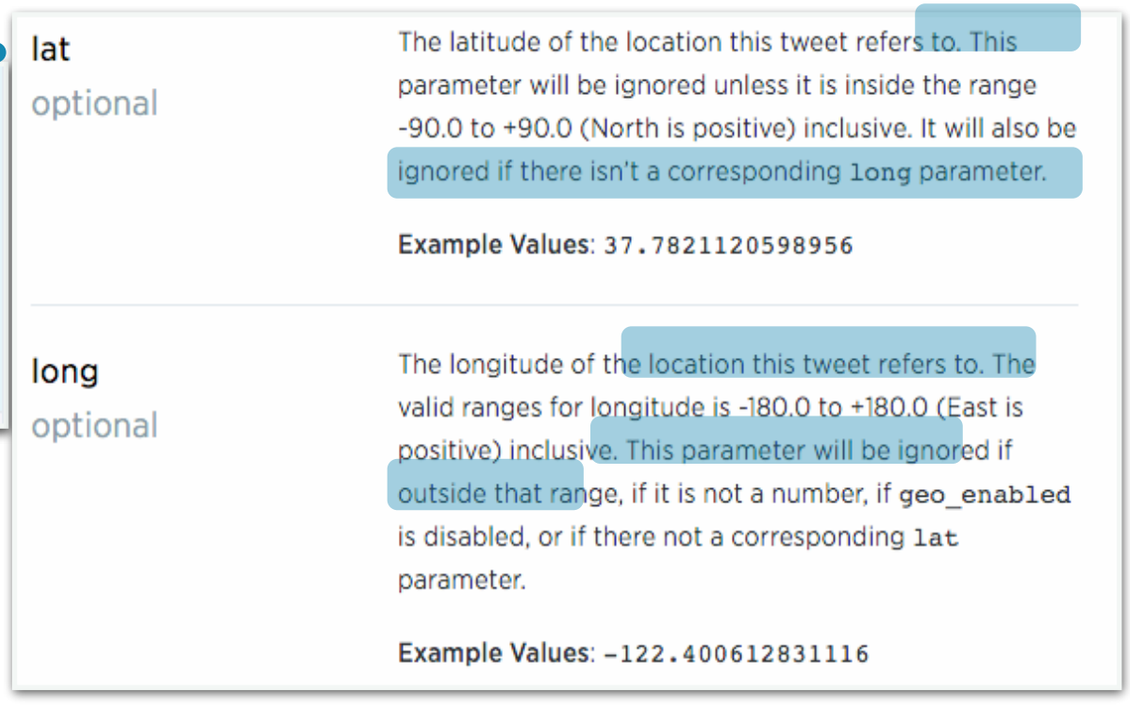
the property **infoWindow** is ignored, when **suppressInfoWindows** is “true”

when searching for an item, **condition** cannot be set to “new” when the **availability** parameter is set to “available”.

Group constraints

- a set of parameters should **either be all excluded** from a request **or all included**





Inter-parameter constraints are everywhere

exactly one of **user_id** and **screen_name** is required

the property **infoWindow** is ignored,
when **suppressInfoWindows** is “true”

latitude will be ignored when there is no
corresponding **longitude** parameter

it is an error to specify values for both **times** and **ns**

the **InterfaceId** or **InterfaceIndex** […] must be set
to the interface for which to retrieve information

inter-parameter constraints

in web applications

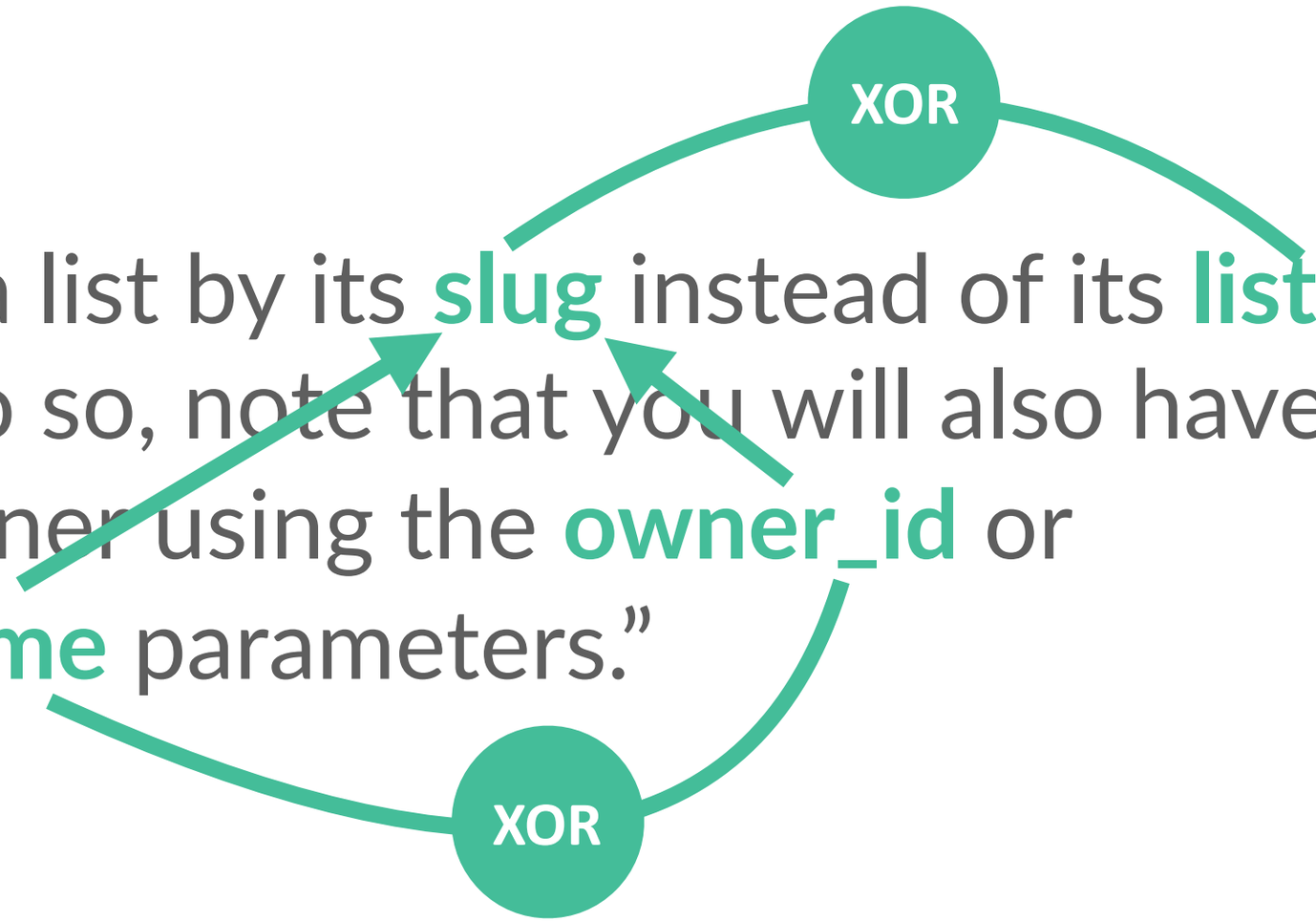
Verifying the constraints

```
export module Twitter{
    export interface PrivateMessage {
        user_id?: number;
        screen_name?: string;
        text: string;
    }

    export function sendPrivateMessage(body: PrivateMessage){
        // ...
    }

}
```

TypeScript

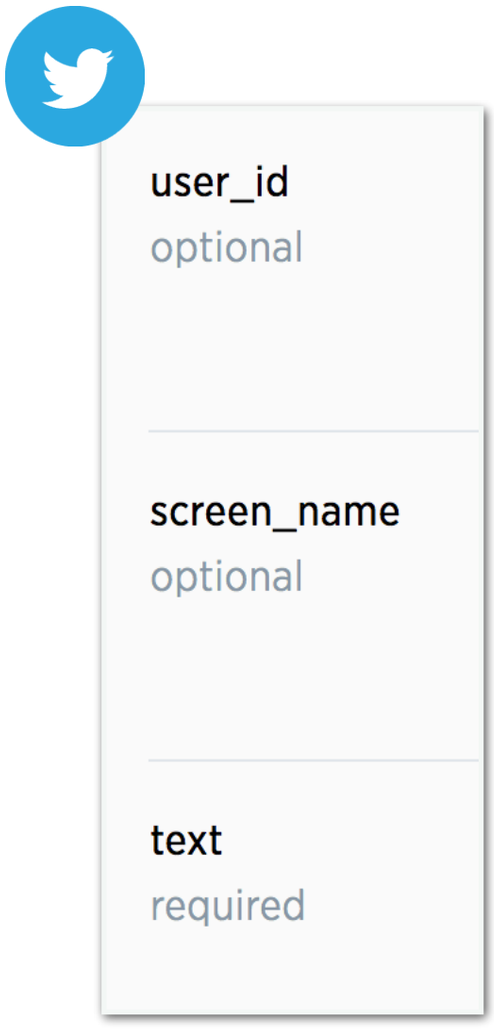
Complex inter-property constraints

“You can identify a list by its **slug** instead of its **list_id**. If you decide to do so, note that you will also have to specify the list owner using the **owner_id** or **owner_screen_name** parameters.”



IPCs in programming languages

exactly one of *user_id* and *screen_name* is required




```
interface PrivateMessage {
  user_id:      number;
  screen_name:  string;
  text:         string;
} constraining {
  present(text);
  present(user_id) XOR present(screen_name);
}
```

Programming with inter-property constraints

Initialisation

➔ object must satisfy constraints

```
present(text)
present(user_id) XOR present(screen_name)
```

```
var msg1: PrivateMessage = {text: "Hello",
                            user_id: 42};
```

```
var msg2: PrivateMessage = {text: "Hello"};
```

```
var msg3: PrivateMessage = {text: "Hello",
                            user_id: 42,
                            screen_name: "Alice"}
```

Accessing properties

➔ only allowed when property is known to be present or absent

```
function getInfo(msg: PrivateMessage) {

    msg.text;                          //:: string

    msg.user_id;                       ✗

    msg.screen_name;                   ✗

}
```

Accessing properties

if-tests provide more information about presence of properties

```
function getInfo(msg: PrivateMessage) {

    msg.text;                          //:: string

    if (msg.user_id !== undefined) {
      msg.user_id;                     //:: number

    } else {
      msg.user_id;                     //:: undefined

    }

}
```

Accessing properties

if-tests provide more information about presence of properties

```
function getInfo(msg: PrivateMessage) {

    msg.text;                          //:: string

    if (msg.user_id !== undefined) {
      msg.user_id;                     //:: number
      msg.screen_name;                 //:: undefined
    } else {
      msg.user_id;                     //:: undefined
      msg.screen_name;                 //:: string
    }

}
```

Updating properties

```
function setInfo(msg: PrivateMessage) {
    msg.text = "Hello again";

    msg.user_id = 43;
```

```
 if (msg.user_id !== undefined) {
     msg.user_id = 43;

     msg.screen_name = undefined;
```

```
  }

}
```

TypeScript strict null checking mode

Updating multiple properties

```
var msg: PrivateMessage = {text: “Hello", user_id: 42};

if (msg.user_id !== undefined) {
  msg.user_id = undefined;
  msg.screen_name = "Alice";
}
```

Updating multiple properties

```
var msg1: PrivateMessage = {text: “Hello", user_id: 42};

var msg2 = assign(msg1, {user_id: undefined,
                         screen_name: "Alice"});
```

```
var msg3 = assign(msg1, {user_id: undefined});
```

all related properties need to be updated

Updating multiple properties

constraints form clusters

```
interface PrivateMessage’ {
  text: string;
  r_user_id: number;
  r_screen_name: string;
  s_user_id: number;
  s_screen_name: string;
} constraining {
  present(text);
  present(r_user_id) XOR
    present(r_screen_name);
  present(s_user_id) XOR
    present(s_screen_name);
}
```

```
var msg: PrivateMessage’ =
  {text: "Hello",
   r_user_id: 42, s_user_id: 43};

var msg2 = assign(msg,
     {r_user_id: 44,
      r_screen_name: undefined});
```

Updating multiple properties

constraints form clusters

```
interface PrivateMessage’ {
  text: string;
  r_user_id: number;
  r_screen_name: string;
  s_user_id: number;
  s_screen_name: string;
} constraining {
  present(text);
  present(r_user_id) XOR
    present(r_screen_name);
  present(s_user_id) XOR
    present(s_screen_name);
}
```

```
var msg: PrivateMessage’ =
  {text: "Hello",
   r_user_id: 42, s_user_id: 43};

var msg2 = assign(msg,
     {r_user_id: 44,
      r_screen_name: undefined});
```

assign is **functional**

Verifying inter-property constraints with propositional logic

Object literals have to satisfy constraints

```
var msg: PrivateMessage = {text: "Hello", user_id: 42};
```

valuation **v**: assigns **truth values** to **proposition letters**

presence or absence

property names

❗ right-hand side needs to be an object literal

Property Access

$$\{\ \texttt{interface constraints}\ \} \quad \models \quad \texttt{present(property)}$$

Constraints dictate property presence

verify presence/absence with logical entailment

```
function getInfo(msg: PrivateMessage) {
    msg.text;                          //:: string
    msg.user_id;
}
```

$$\left\{ \begin{array}{l} \texttt{present(text)} \\ \texttt{present(screen\_name) XOR present(user\_id)} \end{array} \right\} \models \texttt{present(text)}$$

Constraints dictate property presence

verify presence/absence with logical entailment

```
function getInfo(msg: PrivateMessage) {
   msg.text;                          //:: string
   msg.user_id;
}
```

$$\left\{ \begin{array}{l} \texttt{present(text)} \\ \texttt{present(screen\_name) XOR present(user\_id)} \end{array} \right\} \not\models \texttt{present(user\_id)}$$

Constraints dictate property presence

verify presence/absence with logical entailment

```
function getInfo(msg: PrivateMessage) {
   msg.text;                          //:: string
   msg.user_id;
}
```

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(screen\_name) XOR present(user\_id)}\end{array}\right\} \not\models \begin{array}{l}\texttt{present(user\_id)}\\ \texttt{¬present(user\_id)}\end{array}$$

Constraints dictate property presence

verify presence/absence with logical entailment

```
function getInfo(msg: PrivateMessage) {
    msg.text;                          //:: string
    if (msg.user_id !== undefined) {
        msg.user_id;
    }
}
```

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(screen\_name) XOR present(user\_id)}\end{array}\right\} \models \texttt{present(user\_id)}$$

Constraints dictate property presence

verify presence/absence with logical entailment

```
function getInfo(msg: PrivateMessage) {
    msg.text;                          //:: string
    if (msg.user_id !== undefined) {
        msg.user_id;                   //:: number

    }
}
```

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(screen\_name) XOR present(user\_id)}\\ \textbf{present(user\_id)}\end{array}\right\} \models \texttt{present(user\_id)}$$

Constraints dictate property presence

verify presence/absence with logical entailment

```
function getInfo(msg: PrivateMessage) {
    msg.text;                          //:: string
    if (msg.user_id !== undefined) {
        msg.user_id;                   //:: number
        msg.screen_name;               //:: undefined
    }
}
```

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(screen\_name) XOR present(user\_id)}\\ \textbf{present(user\_id)}\end{array}\right\} \models \neg\texttt{present(screen\_name)}$$

Interface Compatibility

var target = source

$$\{\ \text{source constraints}\ \} \models \{\ \text{target constraints}\ \}$$
$$\{\ +\ \text{structural delta}\ \} \quad\quad \{\ +\ \text{structural delta}\ \}$$

Assignment

```
interface PrivateMessage {
  text: string;
  user_id: number;
  screen_name: string;
} constraining {
  present(text);
  present(user_id) XOR
      present(screen_name);
}
```

```
interface PrivateMessageId {
  text: string;
  user_id: number;

} constraining {
  present(text);
  present(user_id);

}
```

```
var msg: PrivateMessage = ...;
var msgId: PrivateMessageId = ...;
msgId = msg;
msg = msgId;
```

Assignment

```
interface PrivateMessage {
  text: string;
  user_id: number;
  screen_name: string;
} constraining {
  present(text);
  present(user_id) XOR
      present(screen_name);
}
```

```
interface PrivateMessageId {
  text: string;
  user_id: number;

} constraining {
  present(text);
  present(user_id);

}
```

```
var source: PrivateMessageId = {text: "Hello", user_id: 42};
var target: PrivateMessage   = source;
```

$$\left\{ \begin{array}{l} \texttt{present(text)} \\ \texttt{present(user\_id)} \end{array} \right\} \not\models \begin{array}{l} \texttt{present(text)} \\ \texttt{present(user\_id) XOR} \\ \qquad \texttt{present(screen\_name)} \end{array}$$

source: PrivateMessageId *target: PrivateMessage*

Assignment

```
interface PrivateMessage {
  text: string;
  user_id: number;
  screen_name: string;
} constraining {
  present(text);
  present(user_id) XOR
      present(screen_name);
}
```

```
interface PrivateMessageId {
  text: string;
  user_id: number;

} constraining {
  present(text);
  present(user_id);

}
```

```
var source: PrivateMessageId = {text: "Hello", user_id: 42};
var target: PrivateMessage   = source;
```

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id)}\\ \textbf{¬present(screen\_name)}\end{array}\right\} \models \begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) XOR}\\ \qquad\texttt{present(screen\_name)}\end{array}$$

source: PrivateMessageId

target: PrivateMessage

Updating multiple properties

```
interface PrivateMessage’ {
  text: string;
  r_user_id: number;
  r_screen_name: string;
  s_user_id: number;
  s_screen_name: string;
} constraining {
  present(text);
  present(r_user_id) XOR
    present(r_screen_name);
  present(s_user_id) XOR
    present(s_screen_name);
}
```

```
var msg: PrivateMessage’ =
  {text: "Hello",
   r_user_id: 42, s_user_id: 43};

var msg2 = assign(msg,
      {r_user_id: 44,
       r_screen_name: undefined});
```

Updating multiple properties

```
interface PrivateMessage’ {
  text: string;
  r_user_id: number;
  r_screen_name: string;
  s_user_id: number;
  s_screen_name: string;
} constraining {
  present(text);
  present(r_user_id) XOR
    present(r_screen_name);
  present(s_user_id) XOR
    present(s_screen_name);
}
```

```
var msg: PrivateMessage’ =
  {text: "Hello",
   r_user_id: 42, s_user_id: 43};

var msg2 = assign(msg,
      {r_user_id: 44,
       r_screen_name: undefined});
```

Future Work

value-dependent constraints

the property **infoWindow** is ignored,
when **suppressInfoWindows** is *true*

if the **steppedLine** value is set to anything other than *false*,
lineTension will be ignored

Implementation

```
interface PrivateMessage {
  text?: string;
  userid?: number;
  screenname?: string;
} constrains {
  present(text);
  xor(present(userid), present(screenname));
}

let msg: PrivateMessage = {text: "Hello", userid: 42, screenname: "Alice"};
```

```
Type '{ text: string; userid: number; screenname: string; }' is not assignable to type
'PrivateMessage'.
Predicate xor(present(userid),present(screenname)) was not satisfied in type { text: string; userid:
number; screenname: string; }
```

https://github.com/noostvog/TypeScriptIPC

Revisiting the mock-up server

```
export module Twitter{
    export interface PrivateMessage {
        text: string;
        user_id: number;
        screen_name: string;
    } constraining {
        present(text);
        present(user_id) XOR present(screen_name);
    }

     export function sendPrivateMessage(body: PrivateMessage){
        // ...
     }

}
```

interfaces with inter-property constraints

propositional logic to define and verify constraints

Spotter guide to inter-property constraints:

Exclusive constraint:
either, *exactly*, *subsumed* or *one of*

Dependent constraint:
additional or *providing*

Group constraint:
corresponding or *providing*