Verification of Communication in Web Applications

Nathalie Oostvogels
Vrije Universiteit Brussel
noostvog@vub.ac.be

Joeri De Koster
Vrije Universiteit Brussel
jdekoste@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel
wdmeuter@vub.ac.be

ABSTRACT

In the current world of web applications, none lives on an island. Current web applications often rely on a myriad of external web APIs and communicate with these external APIs through various HTTP requests spread throughout the application. The specification of those APIs is often described textually and programmers have to resort to manual verification of their code in order to verify if their API calls follow the specification. Next to a correct entry point, the data provided with the API call must also obey a set of constraints. Depending on the complexity of the API and its constraints this manual verification can become a difficult task. In this paper we show the need for more expressive machine-readable web APIs that allow the specification of more complex constraints. We also investigate where current verification tools for web APIs fall short and identify the challenges for the development of static verification tools that enable automatic verification of API calls.

Keywords

type system, web API, web application

1. INTRODUCTION

Today it is hard to imagine a web site without cross-website functionality such as a like button from Facebook, a video from YouTube or a Twitter feed. This is in high contrast with the early days of the web, where web pages were static, stand-alone and confined to serving information and media from within a single domain for clients to consume. JavaScript [7] was first introduced in 1995 with the goal to make web pages more dynamic, enabling local validation of forms and simple animations through dynamic manipulation of HTML. Throughout the years, web sites evolved to real web *applications*. Partial page updates were popularized by Asynchronous JavaScript And XML (AJAX), which rapidly became the dominant way of interacting with other web services from JavaScript. Today, external web services expose their functionality through an Application Programming Interface (API) and websites can use those APIs to integrate that functionality with their own web pages. Web applications include that functionality by sending information to and retrieving information from a web service using HTTP requests. The use of other web services and their APIs is adopted in almost every web application, which is echoed by others, such as the OpenAPI Initiative: "APIs form the connecting glue between modern applications. Nearly every application uses APIs to connect with corporate data sources, third party data services or other applications".[1]

Information about the entry points of an API is usually informally described in a textual API specification, which is publicly accessible. For every entry point, it lists inputs (which parameters are accepted and what their values should be) and its outputs (which are returned to the client). Additionally, the specification often lists constraints on the input parameters of each entry point. Examples of such constraints are the type of a field, whether a field is required or optional, which values the field may contain, the maximum length of the field, etc. Over the years, the constraints found in web APIs have become increasingly more complex to the point where some constraints can even span multiple fields.

Because these APIs rely on external resources it is impossible for development environments and tools to automatically verify or aid the programmer in verifying their API calls without some form of machine-readable API specification. Moreover, incorrect usage of web services often results in unexpected behaviour or vague error messages which severely limits the possibilities for runtime verification.

There already exist a number of machine-readable web APIs (see Section 4). However, constraints over multiple fields have largely been ignored by these existing API specifications. We argue that this severely limits their usability for expressing more complex scenarios that are commonly found in today's web APIs. Additionally, we also argue that there currently exist no tools that fully verify the use of machine-readable web APIs in web applications. In order to solve these issues we introduce extensions for multi-field requirements for the OpenAPI specification[2] and identify the challenges of static verification of API usage in web applications.

The contributions of this paper are threefold; we will:

1. Identify a number of multi-field constraints that are commonly found in existing web APIs (Section 3);

ACM ISBN .
DOI:

[1]https://openapis.org/
[2]https://github.com/OAI/OpenAPI-Specification

2. Show where current machine-readable API specifications fall short and introduce a machine-readable API specification based on the OpenAPI specification that meets the constraints of web APIs in the wild (Section 4);

3. Give an overview of current research and technologies for verification of API usage and explain our vision on static verification of web applications (Section 5);

2. WEB API CONSTRAINTS IN THE WILD

Hypertext Transfer Protocol (HTTP) is the universally used protocol for for web communication between different web services. The challenges of typing web communication based on the HTTP protocol are threefold.

Firstly, since the HTTP protocol only supports text based representations, the data that is included with each request needs to be converted to a string based representation (often JSON) before being included with the request. This makes giving any guarantees about the type of data that is sent along with the request more difficult as the string based representation of that data can be pieced together from different origins in the code.

Secondly, the most basic way of sending an HTTP request in JavaScript is making use of an `XMLHTTPRequest` object. Unfortunately, working with `XMLHTTPRequest` objects is cumbersome: the developer needs to set up the request and its associated data and send it, use a callback for monitoring the different status codes, and then deserialize the response (typically JSON) into a JavaScript object. Fortunately, modern web applications rarely use `XMLHTTPRequest` directly: instead they make use of JavaScript frameworks such as jQuery, Prototype and Dojo, or Node.js packages such as `request`. These frameworks define helpers which simplify the use of HTTP requests. For this reason we aim to provide type information about web communication on the level of these helper functions.

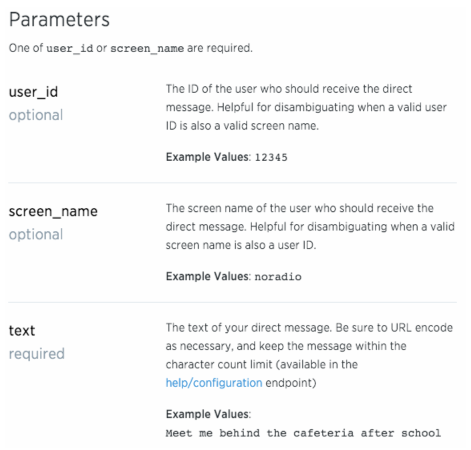
For example, listing 1 shows how to send a private message on Twitter using the `request.post` helper function.[3]

Listing 1: Request to Twitter

```
request.post(
 {url: ‘api.twitter.com/1.1/direct_messages/new.json’,
  form: {user_id: 123, text: "Hi Twitter"}},
 function(error,r,result){ console.log(result) });
```

In this example we use the `post` method of the `request` package to send a POST request to the twitter API. The first argument of the method call contains the form data, in our case a `url` which indicates the entry point of the Twitter API that will be used, and the data that accompanies the request. In this example, we send the ID of the user we want to send the message to, along with the message itself. The second argument of the method call to `request.post` is a callback function that takes as arguments an error code (if any), the response header and the deserialized result. Providing type information on this level makes sense because the types of the different input parameters and the result value are exposed by the request package. The serialization of the form data and the deserialization of the result value are hidden within the package.

[3]For simplicity, we emit user authentication and error checking in this and later examples.



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`

Figure 1: Excerpt from the Twitter API specification

The third and perhaps most important challenge in typing web communication originates from the informal and often incomplete way web APIs are currently specified. Most web services today offer a specification of their API that describes the functionality of the API (the entry points). For every entry point, a list of parameters is provided, together with a description of the return data. For example, an excerpt of the description of the Twitter entry point used in Listing 1 can be found in Figure 1: it lists all possible parameters for a request to the entry point `direct_messages/new`. In this case, there are three parameters that can accompany the request: `user_id`, `screen_name` and `text`. Most API descriptions have several kinds of constraints listed for every parameter (type of the parameter, possible values etc), but Twitter only indicates whether a field is required or optional. The entry point we used above only requires the `text` parameter, and either the user ID or their screen name to indicate the recipient of the private message.

API specifications are often *only* presented informally, with examples of valid requests, parameter listings and descriptions. They cannot be verified automatically, forcing users of these APIs to develop their applications by trial and error. This also severely limits the amount of tool support that can be provided for aiding the developer in verifying the correctness of requests to external APIs. A machine-readable API specification could enable a development environment in aiding the developer by automatically providing extra information while writing the code for issuing a request. Web applications can contain many requests, to many different APIs. These requests can be scattered throughout the entire application code. Manually verifying every request in a web application is a time-consuming and error-prone task, but satisfying the constraints set by the API provider is essential for a request to succeed.

Every request in the application code contains a number of implicit assumptions the developer needs to keep in mind. For example, for the request in Listing 1, the following assumptions are made:

1. There is a web service available at the URL `api.twitter.com/1.1/direct_messages/new.json`;

2. The web service expects a POST request for this entry point;

3. The given parameter object satisfies the constraints for this entry point set by the web service:

    (a) the object contains all required fields, in this case: the `text` field;

    (b) although both `user_id` and `screen_name` are indicated as optional, exactly one of them is required, which means the object must either contain a `user_id` or a `screen_name` field;

Another problem with automatically verifying requests originates from the fact that there is currently no uniform way in which APIs respond to requests that do not satisfy the necessary constraints. In the above example, if assumption 1 is not met, Twitter returns a custom error message as a result. Other web APIs return for example `404 Not found` errors (e.g. Facebook Graph API) or `400 Bad request` errors (e.g. YouTube). Unsatisfied single-field constraints (assumption 2a) often result in clear error messages from the server. However, this is a runtime error and if the request is not covered by a unit test such errors might only pop up after the application is deployed. Sending requests that do not fulfil multi-field constraints (assumption 2b) will often result in bugs that are difficult to find. In our example, when a request has both the `user_id` and the `screen_name` as arguments, Twitter arbitrarily uses the screen name and ignores the user ID. Even when the user ID and the screen name belong to different users, Twitter will silently pick one.

There is a clear need for machine-readable API specifications in order to enable compile time verification of API requests. This in turn would enable tool support for automatically generating and verifying requests during development. There are a number of existing machine-readable API specification languages (See Section 5.1). However, they lack some expressiveness when it comes to specifying constraints over multiple fields. In this paper we define these constraints as *multi-field constraints*. In the next section we introduce three types of multi-field constraints and show that they are commonly found in today's web APIs.

3. MULTI-FIELD CONSTRAINTS

Traditionally, a web API specifies a number of constraints on each field included with a request. For example, these *single-field constraints* can specify the types of values that are accepted or whether the field is required or optional. Single-field constraints are mostly well documented and have proper error handling. Constraints over different fields are a bit less common but still prevalent throughout the different web APIs. However, these multi-field constraints are often not properly documented and there is no uniform way of handling errors when they are not satisfied.

In this section we identify the following three kinds of multi-field constraints found in web APIs today:

1. Exclusive constraints;
2. Dependent constraints; and
3. Group constraints.

We look at how they are currently specified in existing web APIs and how errors are handled when these kinds of constraints are not satisfied.

3.1 Exclusive constraints

A multi-field constraint that often occurs in web APIs is where **exactly one of a number of fields is required**. Figure 1 shows an example of an exclusive constraint. Twitter expects that you either send the `user_id` or `screen_name` along with the request to indicate the recipient of the private message. The involved fields are tagged as optional in the API specification. This is not completely true as one of both must be supplied for the request to succeed.

This exclusive constraint is reflected in the API specification with the following line: *"One of user_id or screen_name are required"*.

Exclusive constraints exist in many API specifications, such as in Facebook[4] where *"either link, place or message must be supplied"* for publishing a status update, or when creating a new charge with Stripe[5], where *"either source or customer is required"*) or when retrieving a playlist from YouTube[6], where you may only provide one filter (*"specify exactly one of the following parameters"*).

We investigated how different API providers respond to requests that do not satisfy exclusive constraints. The following three categories are the most common responses:

1. The API provider returns an **error message**. This is how API providers often deal with unsatisfied single-field constraints as well. However, when the fields do not satisfy the multi-field constraints, more often than not this results in vague error messages. Let us take as an example the Youtube API: when supplying more than one filter for a playlist, the following vague error message is returned: *"Incompatible parameters specified in the request."*.

2. The API provider makes a **silent choice**. For example, Twitter does not complain when both the screen name and user ID are passed along when sending a direct message. However, when the screen name and the user ID belong to different users, Twitter chooses the screen name over the user ID, instead of raising an error. These kinds of errors are very difficult to debug.

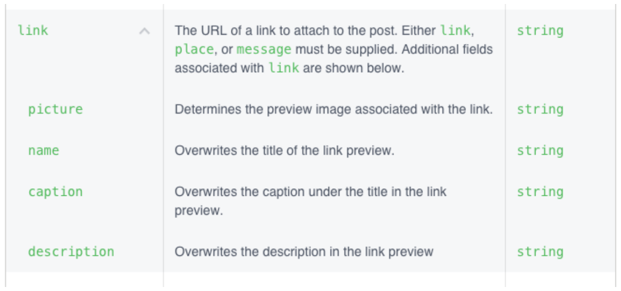
3. **The API specification is incorrect**: in the case of Facebook, where their API specification mentions *"either link, place or message must be supplied"* for publishing a status update, supplying all fields results in sensible status update, where all provided values are combined.

3.2 Dependent constraints

We call a multi-field constraint a *dependent* constraint when **the exclusion of a field (which we call the *base field*) precludes the inclusion of another field or a set of fields**. In other words, a set of fields is dependent to a base field if they should only be included in the request if the base field is also included in the request.

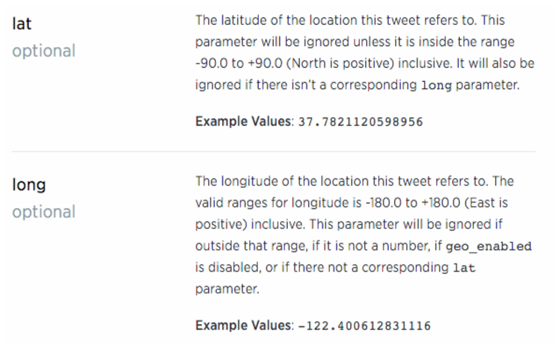
An example of a dependent constraint can be found in Figure 2, which shows an excerpt of the Facebook API specification for publishing a new status update. When posting a `link` on someone's wall, you can also add a `picture`, `name`,

[4]https://developers.facebook.com/docs/graph-api/reference/v2.6/user/feed#publish
[5]https://stripe.com/docs/api/node#create_charge
[6]https://developers.google.com/youtube/v3/docs/videos/list



link	The URL of a link to attach to the post. Either link, place, or message must be supplied. Additional fields associated with link are shown below.	string
picture	Determines the preview image associated with the link.	string
name	Overwrites the title of the link preview.	string
caption	Overwrites the caption under the title in the link preview.	string
description	Overwrites the description in the link preview	string

Figure 2: Dependent fields in the Facebook API



lat optional	The latitude of the location this tweet refers to. This parameter will be ignored unless it is inside the range -90.0 to +90.0 (North is positive) inclusive. It will also be ignored if there isn't a corresponding long parameter. Example Values: 37.7821120598956
long optional	The longitude of the location this tweet refers to. The valid ranges for longitude is -180.0 to +180.0 (East is positive) inclusive. This parameter will be ignored if outside that range, if it is not a number, if geo_enabled is disabled, or if there not a corresponding lat parameter. Example Values: -122.400612831116

Figure 3: A group constraint in the Twitter API

`caption` and `description` for that link. These four fields only make sense when the link itself is also passed along with that request. Thus, we say that the `link` field is the *base* field for these four other fields in a dependent constraint.

Dependent fields can be found in other web APIs as well. For example, in the "add a member to a list" entry point in the Twitter API[7], if you provide the `list_slug` field, an `owner_screen_name` or `owner_id` is also required.

Just like with the exclusive constraint, every API provider has a different response when a dependent constraint is not satisfied and these responses can be similarly categorised. Twitter returns an **error message**: *"You must specify either a list ID or a slug and owner."*, while Facebook just **silently ignores** all the dependent fields when the *base* field is not provided.

3.3 Group constraints

Finally, a group constraint occurs when **a set of fields should either be all excluded from a request or all included**. A simple example of a group constraint is shown in Figure 3, which shows an excerpt of the API specification of Twitter. When creating a new tweet via the Twitter API, the user's current location can be provided via the `lat` and `long` fields. However, it is an error to pass along only `lat` *or* `long`: both fields must be included in the request in order for the resulting tweet to have a location.

Group constraints can be found in many APIs including Flickr[8], where all coordinates of a person in a picture (`x`, `y`, `width` and `height`) must be provided. Another example can be found in the Youtube API[9] for creating a playlist, where the `onBehalfOfContentOwnerChannel` is required when a request specifies a value for the `onBehalfOfContentOwner` field, and it can only be used in conjunction with that field.

When the fields of a request do not satisfy the group constraint, both Flickr and Youtube return an **error message**. Contrary to how they deal with invalid dependent constraints, Twitter requires all group fields to be present and **silently ignores** incomplete groups.

3.4 Multi-field constraints in the wild

We analysed a number of popular web APIs in order to investigate how frequent the different multi-field constraints occur in web APIs in the wild. For this, we analysed the five most popular APIs on the web services directory of ProgrammableWeb[10] and, for every web API, we searched for occurrences of the three multi-field constraints. Table 1 summarises our results.

In every API specification, we have looked for keywords that indicate a multi-field constraint. Exclusive constraints are often indicated with *either* or *one of*, dependent constraints can be specified with keywords such as *additional* and *providing*, and keywords for group constraints include *corresponding* and *providing*.

Table 1: Multi-field constraints in web APIs

	Exclusive	Dependent	Group
Google Maps	10	0	2
Twitter	32	14	6
Youtube	11	0	0
Flickr	12	0	1
Facebook	11	5	1

As can be seen from Table 1, every multi-field constraint occurs in more than one API. Exclusive constraints are the most common multi-field constraint in web APIs with a total of 76 occurrences. Out of 98 entry points, Twitter has 32 entry points with an occurrence of an exclusive constraint, which means that on average, one out of three entry points of Twitter require some of the fields to be exclusive. All other APIs also have at least 10 instances of the exclusive constraint in their API specification. Although less frequent, group constraints occur in four of the five APIs we have investigated. Finally, both Facebook and Twitter have dependent constraints in their API specification.

From this section we can conclude that multi-field constraints are omnipresent in modern web APIs and that the way an API responds to a request that does not satisfy these constraints is not always well defined. The API will either respond with an, often vague, error message or silently ignore part of the request. This signifies the need for a machine-readable API specification that enables the specification and automatic verification of these multi-field constraints.

4. MACHINE-READABLE SPECIFICATIONS FOR WEB APIS

It is not an easy task to verify that every call to an API in a web application is correct. Most API providers only have

[7]https://dev.twitter.com/rest/reference/post/lists/members/create
[8]https://www.flickr.com/services/api/flickr.photos.people.add.html
[9]https://developers.google.com/youtube/v3/docs/playlists/insert
[10]http://www.programmableweb.com/apis/directory

a textual version of their specification, which forces developers to manually verify every call to an API in their application. For a tool to verify the correctness of API calls in a web application, it needs a machine-readable version of the API specification. Throughout the years, many machine-readable specifications for APIs have been proposed. In this section, we look at existing machine-readable specifications and how they fit for web APIs, including the multi-field constraints we identified in the previous section.

Machine-readable specifications for web APIs have been around for quite some time (such as WSDL[4]), but recently, many new machine-readable web API specification languages have emerged. These new languages often come together with tools that improve the development and testing of the documentation and offer code generation. Table 2 lists the four most popular specifications on Stackoverflow, given the list of machine-readable API specifications on Wikipedia: OpenAPI specification (formerly known as the Swagger specificaton), MSON (Markdown Syntax for Object Notation), RAML (RESTful API Modeling Language) and WADL[11] (Web Application Description Language). We also include JSON Schema in our discussion, because it will prove to be an interesting specification for web APIs as well. Note that JSON Schema is used for describing only one object at the time, while the others are specifications for an entire web API.

Table 2 lists for every specification which constraints it can express for the parameters of a request. We have limited the constraints for a single field to those in the first eight columns of the table, but there are specifications which support many more single-field constraints such as an exclusive minimum and maximum, the minimum and maximum length of a string, the minimum and maximum size of an array, whether an array should have unique items, etc. We will discuss the last three columns of the table in detail, as these concern multi-field constraints.

Only OpenAPI and JSON Schema support multi-field constraints. The *allOf* constraint is present in both OpenAPI and JSON Schema: it expresses that every constraint in a set of constraints must be valid. This can range from simple constraints such as "the parameter must be a string and its length must be 5" to combining multiple constraints on the same object. For example, Listing 2 shows a snippet of an JSON Schema where we are defining a schema for an object that must have a `name` field and an `age` field, where `name` must be a string and `age` must be a number. However, the `allOf` is not suited for any of the multi-field constraints we have identified in the previous section.

Listing 2: `allOf` constraint

```
{"allOf" :
  [{ "type": "object",
     "properties": {"name":{"type":"string"}}},
   { "type": "object",
     "properties": {"age":{"type":"number"}}}]};
```

The second multi-field constraint found in JSON Schema is *dependencies*, which can be used for expressing both the dependent constraints and the group constraints from the previous section. Dependencies require that certain other properties must be present if a given property is present. They are well suited for dependent constraints, as can be seen in Listing 3. It shows the JSON Schema for an object with three fields: a picture along with accompanying name and date. The intention behind the JSON Schema is that when the picture itself is not provided, the fields `pic_name` and `pic_date` should not be present either.

For group constraints we can have a mutual dependency as shown in Listing 4, where the `long` and `lat` fields should only occur together.

Listing 3: Dependencies for dependent constraints

```
{ type: 'object',
    properties: {
       pic : { type : 'image'},
       pic_name : { type : 'string'},
       pic_date : { type : 'date'}},
    dependencies : {
       'pic_name' : ['pic'],
       'pic_date' : ['pic']}};
```

Listing 4: Dependencies for group constraints

```
  { type: 'object',
     properties: {
       long : { type : 'number'},
       lat  : { type : 'number'}},
     dependencies : {
       'long' : ['lat'],
       'lat'  : ['long']}};
```

The final multi-field constraint in Table 2 is *oneOf*, which is used to express that exactly one of the provided constraints must be valid for a given object. An example can be found in Listing 5, where an object is considered valid against this schema if it is valid against exactly one of the properties. `oneOf` seems like a good fit for the exclusive constraint we have introduced in Section 3, but there are some design choices for JSON Schema that do not correspond with an exclusive constraint. Consider the example of Listing 5: we would like to have either the field `a` (with type `string`) or `b` (with type `number`). However, an object `{a:42, b:42}` would be accepted as well because `a` is not a string, and therefore the first schema is not considered valid. For the exclusive constraints found in web APIs, this is not a good fit: even though the type is incorrect, we want to ensure that exactly one of those fields is present.

Listing 5: `oneOf` for exclusive constraints

```
{"oneOf":[{"type":"object",
           "properties":{"a":{"type":"string",
                              required:true}}},
          {"type":"object",
           "properties":{"b":{"type":"number",
                              required:true}}}]}
```

At first sight, JSON Schema seems like a good fit for describing web APIs in a machine-readable way. However, on top of the previously mentioned differences between JSON Schema and web API specifications JSON Schema always accepts fields that were not described in the schema. This is not desirable behaviour for validating web APIs: the list of parameters should be an exhaustive list and any other parameter should be rejected. For example, a verification tool using JSON Schema would not be able to detect a parameter with an incorrect parameter name, as the incorrect name would not be described in the JSON Schema and thus be accepted.

In Sections 5.2 and 5.3, we introduce our view on the static verification of APIs in web applications and show a proof of

Table 2: Constraints in web API specifications

	Type	Min	Max	Pattern	Required	Enum	AllOf	Dependencies	OneOf
OpenAPI Spec	✓	✓	✓	✓	✓	✓	✓		
MSON	✓				✓	✓			
RAML	✓	✓	✓	✓	✓	✓			
WADL	✓				✓	✓			
JSON Schema	✓	✓	✓	✓	✓	✓	✓	✓	✓

concept by means of a runtime verification tool. To do so, we first define our extension of the OpenAPI specification, such that it supports multi-field constraints as well. The OpenAPI initiative is aiming to be a vendor-neutral API specification for web services, and is supported by many companies such as Google and Microsoft and will likely be used by many API providers.

5. VERIFICATION OF WEB APIS

Manually checking if every request in a web application is correct, is a very tedious task. In this section we will discuss current tools that try to alleviate developers from this task by automatically verifying API requests, discuss how they do not suffice for real-world web APIs and introduce our view on the static verification of web APIs. Our current prototype tool is not yet able to statically verify API constrains but is able to verify requests with multi-field constraints at runtime.

5.1 Related Work

Statically Typed JavaScript.

In recent years, many statically typed versions of JavaScript have emerged, such as TypeScript, Flow, and Tejas [14], as well as specialised type systems for JavaScript such as [1, 12, 16]. These approaches all add a form of static typing to JavaScript, with the aim of helping developers find type errors faster. However, none of these support thorough type checking for web API requests. At most, they check whether web API requests are conform to the interface for the request library being used. For the code in Listing 1, TypeScript only checks that a URI or URL parameter is provided with as type `string`, and that the callback function has three arguments. None of the constraints imposed by the API providers are taken into account, which means that mistakes in API requests are not caught by the type checker.

JavaScript Alternatives.

There is a field of research that aims to replace JavaScript by more easily typechecked languages, which are very different from JavaScript but can compile down to JavaScript code. Examples of such languages are ClojureScript and Scala. Existing JavaScript web applications will still need to be maintained, however, and there is no clear migration strategy proposed. Furthermore, their respective type systems do not take into account any of the multi-field constraints we have identified in this paper.

In a similar vein, other researchers propose to remove the boundaries between clients and server. Languages such as Links [6], Hop [15], Opa[11], Ocsigen [3] and Ur/Web [5] allow developers to write web applications using a single language. Special syntax and primitive functions define where code is executed (either on the client or on the server) and communication between the two is abstracted away. Web applications written in this style are very easy to verify completely, since the entire flow of requests and responses is available to the type checker. For example, in Ocsigen you represent interactions with the remote server using static types (and subsequently sharing types between client and server). This way they can ensure all HTTP requests send data of the correct type and that all responses are likewise type safe. We level a similar remark against these “melded” web applications, namely that they do not help in maintaining and developing the web applications we have now. Additionally, they also do not take into account constraints for multiple fields.

[11]http://opalang.org

Specifications.

In Section 4, we discussed how specification languages for web APIs deal with multi-field constraints. We have looked at the five most used specification languages and concluded that here is very minimal support for multi-field constraints. There exist many more specifications for web APIs such as WSDL [4], WifL [8], Web IDL [2] and hRests [13]. However, to the best of our knowledge, none of these deal with multi-field constraints.

Security.

In [10], Guha et al. analyze how dynamic web applications interact with web services. Their approach builds up a control-flow graph (using a context-sensitive control-flow algorithm) that matches a normal usage profile for a given web application. Every request chain that does not match this control-flow graph is considered suspected behavior, which is detected and blocked by a proxy they insert between client and server. The focus of this research is on ensuring requests are made in the correct order, rather than checking if the constraints of a request are satisfied.

SAFE$_{\text{WAPI}}$ [2] is a tool that statically analyses web API requests in web applications. Given an API specification in the Web Interface Description Language, SAFE$_{\text{WAPI}}$ statically checks that all arguments required are provided and that the parameters have the correct type. However, SAFE$_{\text{WAPI}}$ only checks a limited set of constraints: types of parameters, and whether they are required or optional.

There exist many tools for the verification of requests in web applications at runtime, both on the client- and server side. The OpenAPI specification (based on the Swagger specification), provides many tools that aid the developer in writing machine-readable API documentation. An example of such a tool is `swagger-js` which, given an OpenAPI specification, constructs a proxy object which the programmer can use to communicate with the provided external API.

When a request is made, swagger-js first verifies whether all required fields are included.

[8] introduces WIfL tools, which generates an API validator from the API specifications. Contrary to our tool, this tool is targeted for testing and developing APIs and their documentation instead of validating requests in a web application.

None of these tools use specifications that can express multi-field requirements, which means they are not taken into account by SAFE$_{\text{WAPI}}$, swagger-js and the Wifl tools.

JSON Schema express is middleware for the Node.js Express framework which verifies whether the parameters of incoming HTTP requests match a JSON Schema specification. However, it does not automatically match entry points to JSON schemas: the programmer needs to indicate which schema is expected for every request in the web application. Moreover, we have already discussed how JSON Schema is not a good fit for web APIs in Section 4.

5.2 Static Verification of Web APIs

We envision a static type system as a means for a web developer to statically ensure every request to a web API happens correctly. Such type systems have traditionally been used to enforce type safety, but nowadays they cover other invariants as well. Verifying that all requests are correct at compile has many advantages: requests are scattered across the entire web application and called at arbitrary times, which means malformed request code may remain undetected for a long time.

In order to fulfill this vision, two requirements must be met. First of all, the type checker must have access to machine-readable specifications of all the APIs which are accessed by the web application. These specifications need to describe the following for every entry point of the API: all the parameters expected for every entry point, and any constraints that are imposed on those parameters. These constraints can describe both single fields (ie. type, required or optional) and multiple fields. As we have indicated in Section 3, constraints over multiple fields occur in many web APIs and thus must be included in the specification as well. We envision that API specifications are stored in a public repository similar to the “DefinitelyTyped” repository for TypeScript, which provides type definitions for common JavaScript libraries. Such a public repository has two key advantages for developers: first of all, they can save work by reusing already existing specifications. Secondly, developers can improve existing specifications in order to refine them or to cope with API changes.

The second requirement for our vision is a static type system and corresponding type checker. This type checker determines the validity of every request in the web application. To do so, it needs to check the following:

1. Whether the request is sent to an existing entry point;
2. Whether the correct HTTP method is used;
3. Whether all constraints imposed on the parameters are met.

We would like to extend the type checker of an existing statically typed version of JavaScript, for example TypeScript. TypeScript is already a popular type-checked alternative for JavaScript, which has as advantage that definitions have been written for many existing libraries, while existing JavaScript code can be gradually transformed to TypeScript.

To check all three bullet points, we need information about the entry point used, which is determined by the request method and path, both represented as parameters in the program text. This is not easy for regular type systems, as they often abstract these parameters to their types, even though their values might be statically known. Dependent types can be used as an inspiration, although these require significant effort from the developer.

The third constraint can be partly solved by state of the art type systems and -checkers, most notably verifying parameter types and enumeration values. Next to these simple constraints, however, there are other kinds of constraints which need to be checked. Required and optional field constraints are related to the current research in nullable and optional types [9, 17], which our type system can incorporate. Finally, the multi-field constraints we listed in Section 3 (the exclusive constraint, the dependent constraint and the group constraint) need to be verified as well. For this, we envision the type checker to encompass a validation on the structure of the request parameters.

Both requirements for verifying web APIs can be tackled independently.

5.3 Prototype of a Runtime Verification Tool

In this section we describe a prototype *runtime* verification tool, as a first step towards the static verification of web APIs. We use this tool for exploring the design space of web API verification in web applications, as it allows to faster iterate over API specifications. However, we will show how the runtime verification tool is also already useful by itself.

We have developed a prototype tool which verifies the usage of APIs at *runtime* by intercepting method calls on the well-known request library of Node.js. The code is written in such a way that it can easily be adapted for other request-making libraries as well, such as jQuery. Before any request can be verified, the developer needs to provide a specification for the APIs he intends to use:

```
request.addDefinition(twitterDefinition);
```

These specifications are JSON objects which embody the OpenAPI specification, extended with multi-field constraints. Listing 6 contains an abbreviated specification for the /direct_messages/new entry point, extended with a multi-field constraint. Lines 13 and 14 demand that exactly one of the user_id or the screen_name parameters should be provided.

Listing 6: OpenAPI specification of a part of the Twitter API

```
1  "/direct_messages/new": {
2    "post": {
3      "parameters": [
4        {   "name": "user_id",
5            "type": "integer",
6            "required": false     },
7        {   "name": "screen_name",
8            "type": "string",
9            "required": false     },
10       {   "name": "text",
11           "type": "string",
12           "required": true      } ],
13     "constraints": {
14       "xor": [["user_id", "screen_name"]] },
15     "responses": { ... }}},
```

When the program now sends a GET or POST request, our wrapper verifies whether all the constraints are satisfied. If this is the case, the call is forwarded to the original request module. If not, our tool raises an error, alerting the programmer. For example, when a developer provides both `user_id` and `screen_name` in a request to `/direct_messages/new`, the request will succeed and Twitter will silently choose A over B without the runtime tool. However, *with* the runtime tool, the developer will get notified of his mistake with a detailed warning such as: `Error: in a request to https://api.twitter.com/1.1/direct_messages/new.json: exclusive constraint not satisfied (user_id and screen_name cannot be included together within the same request).`

6. CONCLUSION AND FUTURE WORK

In recent years, JavaScript has been the driver for responsive web applications, which often communicate with external servers using HTTP requests. Every request needs to match certain constraints of the API that is used. These constraints are usually defined in a, often textual, public API specification. In this paper we have shown the limitations of textual API specifications in automatically verifying requests. Moreover, we have shown that these constraints are not always unambiguously defined and that some constraints can be hidden within the description. In addition, there is no designated way in which services respond to incorrect use of an API. Either the API returns an error or silently ignores part of the request or returns a custom response. All these limitations severely limit debugging incorrect use of an API. Furthermore, every time the API changes, developers have to manually verify and transition their code to fit the new API description. We have shown that there is a need for a machine-readable API specification in order to enable advanced tool support for aiding developers in correctly using web APIs.

Currently existing API specification languages already enable the specification of single-field constraints within an API. However, in this paper we looked into several web APIs in the wild and identified three kinds of multi-field constraints that are commonly used. We also show the limitations of existing API specification languages in expressing these multi-field constraints.

In this paper we presented our vision for a novel API specification language based on the OpenAPI specification. Our API specification language is a first step towards expressing multi-field constraints. We also envision accompanying tool support for automatic verification of API calls. We have identified challenges for the development of a static verification tool and presented our prototype for the automatic verification of the use of APIs in web applications *at runtime*. Going further, we intend to integrate this analysis into TypeScript, with the aim of checking API requests at compile time (given an extended API specification). Its type system is already capable of checking single-field constraints such as parameter types and optional or required parameters, but we want to improve TypeScript such that it can also statically type check the multi-field constraints of web APIs.

7. REFERENCES

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP 2005-Object-Oriented Programming*, pages 428–452. Springer, 2005.

[2] S. Bae, H. Cho, I. Lim, and S. Ryu. Safewapi: Web api misuse detector for web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 507–517, New York, NY, USA, 2014. ACM.

[3] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a web programming framework. *ACM Sigplan Notices*, 44(9):311–316, 2009.

[4] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*, 26:19, 2007.

[5] A. Chlipala. Ur/web: A simple model for programming the web. *ACM SIGPLAN Notices*, 50(1):153–165, 2015.

[6] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[7] D. Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[8] P. J. Danielsen and A. Jeffrey. Validation and interactivity of web api documentation. In *2013 IEEE 20th International Conference on Web Services (ICWS)*, pages 523–530. IEEE, 2013.

[9] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM SIGPLAN Notices*, volume 38, pages 302–312. ACM, 2003.

[10] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web*, pages 561–570. ACM, 2009.

[11] M. J. Hadley. Web application description language (wadl). 2006.

[12] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP 2010–Object-Oriented Programming*, pages 200–224. Springer, 2010.

[13] J. Kopecky, K. Gomadam, and T. Vitvar. hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, volume 1, pages 619–625. IEEE, 2008.

[14] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. Tejas: retrofitting type systems for javascript. In *ACM SIGPLAN Notices*, volume 49, pages 1–16. ACM, 2013.

[15] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, pages 975–985, 2006.

[16] P. Thiemann. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems*, pages 408–422. Springer, 2005.

[17] S. Tobin-Hochstadt. Practical optional types for clojure. In *Programming Languages and Systems: 25th European Symposium on Programming*, volume 9632, page 68. Springer, 2016.