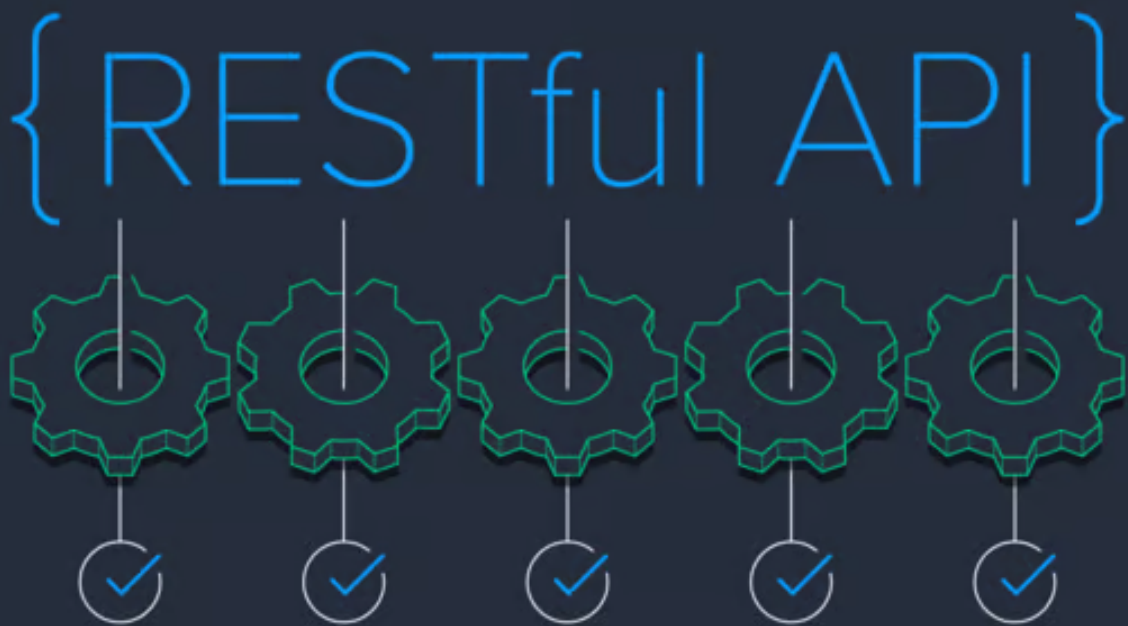
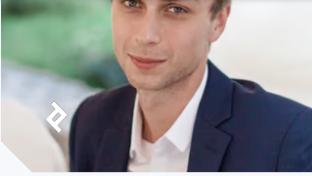


5 Things You Have Never Done With a REST Specification



The existence of RESTful APIs is a popular myth in web development—but it's a myth we can work with. The right tools can help keep documentation consistent and streamline automated testing. In this article, Toptal Freelance JavaScript Developer Alexander Zinchuk explores several time-saving approaches to developing REST APIs, with examples in Node.js and Ruby on Rails.

Toptal authors are vetted experts in their fields and write on topics in which they have demonstrated experience. All of our content is peer reviewed and validated by Toptal experts in the same field.



Alex's decade-plus of JS coding has taught him the language's internals. He's lead a dev team for Yandex and built fault-tolerant systems.

EXPERTISE

API

YEARS OF EXPERIENCE

18

SHARE THIS ARTICLE

in

f

Most front-end and back-end developers have dealt with REST specifications and RESTful APIs before. But not all RESTful APIs are created equal. In fact, they're rarely RESTful at all...

What /s a RESTful API?

It's a myth.

If you think that your project has a RESTful [API](#), you are most likely mistaken. The idea behind a RESTful API is to develop in a way that follows all the architectural rules and limitations that are described in the REST specification. Realistically, however, this is largely impossible in practice.

On the one hand, REST contains too many blurry and ambiguous definitions. For example, in practice, some terms from the HTTP method and status code dictionaries



On the other hand, REST development creates too many limitations. For example, atomic resource use is suboptimal for real-world APIs that are used in mobile applications. Full denial of data storage between requests essentially bans the “user session” mechanism seen just about everywhere.

But wait, it's not that bad!

What Do You Need A REST API Specification for?

Despite these drawbacks, with a sensible approach, REST is still an amazing concept for [creating really great APIs](#). These APIs can be consistent and have a clear structure, good documentation, and high unit test coverage. You can achieve all of this with a high-quality *API specification*.

Usually a REST API specification is associated with its *documentation*. Unlike a specification—a formal description of your API—documentation is meant to be human-readable: for example, read by the developers of the mobile or web application that uses your API.

A correct API description isn't just about writing API documentation well. In this article I want to share examples of how you can:

- Make your unit tests simpler and more reliable;
- Set up user input preprocessing and validation;
- Automate serialization and ensure response consistency; and even
- Enjoy the benefits of static typing.

But first, let's start with an introduction to the API specification world.

OpenAPI



sections:

1. A header with the API name, description, and version, as well as any additional information.
2. Descriptions of all resources, including identifiers, HTTP methods, all input parameters, response codes, and body data types, with links to definitions.
3. All definitions that can be used for input or output, in JSON Schema format (which, yes, can also be represented in YAML.)

OpenAPI's structure has two significant drawbacks: It's too complex and sometimes redundant. A small project can have a JSON specification of thousands of lines. Maintaining this file manually becomes impossible. This is a significant threat to the idea of keeping the specification up-to-date while the API is being developed.

There are multiple editors that allow you to describe an API and produce OpenAPI output. Additional services and cloud solutions based on them include Swagger, Apiary, Stoplight, Restlet, and many others.

However, these services were inconvenient for me due to the complexity of quick specification editing and aligning it with code changes. Additionally, the list of features was dependant on a specific service. For example, creating full-fledged unit tests based on the tools of a cloud service is next to impossible. Code generation and mocking endpoints, while seeming to be practical, turn out to be mostly useless in practice. This is mostly because endpoint behavior usually depends on various things such as user permissions and input parameters, which may be obvious to an API architect but are not easy to automatically generate from an OpenAPI spec.

Tinyspec

In this article, I will use examples based on my own REST API definition format, [tinyspec](#). Definitions consist of small files with an intuitive syntax. They describe



is automatically compiled into a full-fledged OpenAPI format that can be immediately used in your project.

I will also use Node.js (Koa, Express) and Ruby on Rails examples, but the practices I will demonstrate are applicable to most technologies, including Python, PHP, and Java.

Where API Specification Rocks

Now that we have some background, we can explore how to get the most out of a properly specified API.

1. Endpoint Unit Tests

Behavior-driven development (BDD) is ideal for developing REST APIs. It is best to write unit tests not for separate classes, models, or controllers, but for particular endpoints. In each test you emulate a real HTTP request and verify the server's response. For Node.js there are the [supertest](#) and [chai-http](#) packages for emulating requests, and for Ruby on Rails there is [airborne](#).

Let's say we have a `User` schema and a `GET /users` endpoint that returns all users. Here is some tinyspec syntax that describes this:

```
# user.models.tinyspec
User {name, isAdmin: b, age?: i}

# users.endpoints.tinyspec
GET /users
=> {users: User[]}
```

And here is how we would write the corresponding test:

®

```
describe('/users', () => {
  it('List all users', async () => {
    const { status, body: { users } } = request.get('/users');

    expect(status).to.equal(200);
    expect(users[0].name).to.be('string');
    expect(users[0].isAdmin).to.be('boolean');
    expect(users[0].age).to.be.oneOf(['boolean', null]);
  });
});
```

RUBY ON RAILS

```
describe 'GET /users' do
  it 'List all users' do
    get '/users'

    expect_status(200)
    expect_json_types('users.*', {
      name: :string,
      isAdmin: :boolean,
      age: :integer_or_null,
    })
  end
end
```

When we already have the specification that describes server responses, we can simplify the test and just check if the response follows the specification. We can use tinyspec models, each of which can be transformed into an OpenAPI specification that follows the JSON Schema format.

Any literal object in JS (or `Hash` in Ruby, `dict` in Python, *associative array* in PHP, and even `Map` in Java) can be validated for JSON Schema compliance. There are even appropriate plugins for testing frameworks, for example [jest-ajv](#) (npm), [chai-ajv-json-schema](#) (npm), and [json_matchers](#) for RSpec (rubygem).

®

before each test run):

```
tinyspec -j -o openapi.json
```

NODE.JS

Now you can use the generated JSON in the project and get the `definitions` key from it. This key contains all JSON schemas. Schemas may contain cross-references (`$ref`), so if you have any embedded schemas (for example, `Blog {posts: Post[]}`), you need to unwrap them for use in validation. For this, we will use [json-schema-deref-sync](#) (npm).

```
import deref from 'json-schema-deref-sync';
const spec = require('./openapi.json');
const schemas = deref(spec).definitions;

describe('/users', () => {
  it('List all users', async () => {
    const { status, body: { users } } = request.get('/users');

    expect(status).toEqual(200);
    // Chai
    expect(users[0]).toBe.validWithSchema(schemas.User);
    // Jest
    expect(users[0]).toMatchSchema(schemas.User);
  });
});
```

RUBY ON RAILS

The `json_matchers` module knows how to handle `$ref` references, but requires separate schema files in the specified location, so you will need to [split the swagger.json file into multiple smaller files first](#):

®

```
require 'json_matchers/spec'

JsonMatchers.schema_root = 'spec/schemas'

# Fix for json_matchers single-file restriction
file = File.read 'spec/schemas/openapi.json'
swagger = JSON.parse(file, symbolize_names: true)
swagger[:definitions].keys.each do |key|
  File.open("spec/schemas/#{key}.json", 'w') do |f|
    f.write(JSON.pretty_generate({
      '$ref': "swagger.json#/definitions/#{key}"
    }))
  end
end
```

Here is how the test will look like:

```
describe 'GET /users' do
  it 'List all users' do
    get '/users'

    expect_status(200)
    expect(result[:users][0]).to match_json_schema('User')
  end
end
```

Writing tests this way is incredibly convenient. Especially so if your IDE supports running tests and debugging (for example, WebStorm, RubyMine, and Visual Studio). This way you can avoid [using other software](#), and the entire API development cycle is limited to three steps:

1. Designing the specification in tinyspec files.
2. Writing a full set of tests for added/edited endpoints.
3. Implementing the code that satisfies the tests.

2. Validating Input Data

®

updates.

Let's say that we have the following specification, which describes the patching of a user record and all available fields that are allowed to be updated:

```
# user.models.tinyspec
UserUpdate !{name?, age?: i}

# users.endpoints.tinyspec
PATCH /users/:id {user: UserUpdate}
=> {success: b}
```

Previously, we explored the plugins for in-test validation, but for more general cases, there are the [ajv](#) (npm) and [json-schema](#) (rubygem) validation modules. Let's use them to write a controller with validation:

NODE.JS (KOA)

This is an example for Koa, the successor to Express—but the equivalent Express code would look similar.

```
import Router from 'koa-router';
import Ajv from 'ajv';
import { schemas } from './schemas';

const router = new Router();

// Standard resource update action in Koa.
router.patch('/:id', async (ctx) => {
  const updateData = ctx.body.user;

  // Validation using JSON schema from API specification.
  await validate(schemas.UserUpdate, updateData);

  const user = await User.findById(ctx.params.id);
  await user.update(updateData);
```

®

```
async function validate(schema, data) {  
  const ajv = new Ajv();  
  
  if (!ajv.validate(schema, data)) {  
    const err = new Error();  
    err.errors = ajv.errors;  
    throw err;  
  }  
}
```

In this example, the server returns a `500 Internal Server Error` response if the input does not match the specification. To avoid this, we can catch the validator error and form our own answer that will contain more detailed information about specific fields that failed validation, and follow the specification.

Let's add the definition for the `FieldsValidationError`:

```
# error.models.tinyspec  
Error {error: b, message}  
  
InvalidField {name, message}  
  
FieldsValidationError < Error {fields: InvalidField[]}
```

And now let's list it as one of the possible endpoint responses:

```
# users.endpoints.tinyspec  
PATCH /users/:id {user: UserUpdate}  
  => 200 {success: b}  
  => 422 FieldsValidationError
```

This approach allows you to write unit tests that test the correctness of error scenarios when invalid data comes from the client.

3. Model Serialization

®

represented by models and their instances and collections.

The process of forming the JSON representations for these entities to be sent in the response is called *serialization*.

There are a number of plugins for doing serialization: For example, [sequelize-to-json](#) (npm), [acts_as_api](#) (rubygem), and [jsonapi-rails](#) (rubygem). Basically, these plugins allow you to provide the list of fields for a specific model that must be included in the JSON object, as well as additional rules. For example, you can rename fields and calculate their values dynamically.

It gets harder when you need several different JSON representations for one model, or when the object contains nested entities—associations. Then you start needing features like inheritance, reuse, and serializer linking.

Different modules provide different solutions, but let's consider this: Can the specification help out again? Basically all the information about the requirements for JSON representations, all possible field combinations, including embedded entities, are already in it. And this means that we can write a single automated serializer.

Let me present the small [sequelize-serialize](#) (npm) module, which supports doing this for Sequelize models. It accepts a model instance or an array, and the required schema, and then iterates through it to build the serialized object. It also accounts for all the required fields and uses nested schemas for their associated entities.

So, let's say we need to return all users with posts in the blog, including the comments to these posts, from the API. Let's describe it with the following specification:

```
# models.tinyspec
Comment {authorId: i, message}
Post {topic, message, comments?: Comment[]}
User {name, isAdmin: b, age?: i}
```

®

```
GET /blog/users  
=> {users: UserWithPosts[]}
```

Now we can build the request with Sequelize and return the serialized object that corresponds to the specification described above exactly:

```
import Router from 'koa-router';  
import serialize from 'sequelize-serialize';  
import { schemas } from './schemas';  
  
const router = new Router();  
  
router.get('/blog/users', async (ctx) => {  
  const users = await User.findAll({  
    include: [{  
      association: User.posts,  
      required: true,  
      include: [Post.comments]  
    }]  
  });  
  
  ctx.body = serialize(users, schemas.UserWithPosts);  
});
```

This is almost magical, isn't it?

4. Static Typing

If you are cool enough to use TypeScript or Flow, you might have already asked, “What of my precious static types?!” With the [sw2dts](#) or [swagger-to-flowtype](#) modules you can generate all necessary static types based on JSON schemas and use them in tests, controllers, and serializers.

```
tinyspec -j  
  
sw2dts ./swagger.json -o Api.d.ts --namespace Api
```

®

```
router.patch('/users/:id', async (ctx) => {
  // Specify type for request data object
  const userData: Api.UserUpdate = ctx.request.body.user;

  // Run spec validation
  await validate(schemas.UserUpdate, userData);

  // Query the database
  const user = await User.findById(ctx.params.id);
  await user.update(userData);

  // Return serialized result
  const serialized: Api.User = serialize(user, schemas.User);
  ctx.body = { user: serialized };
});
```

And tests:

```
it('Update user', async () => {
  // Static check for test input data.
  const updateData: Api.UserUpdate = { name: MODIFIED };

  const res = await request.patch('/users/1', { user: updateData });

  // Type helper for request response:
  const user: Api.User = res.body.user;

  expect(user).to.be.validWithSchema(schemas.User);
  expect(user).to.containSubset(updateData);
});
```

Note that the generated type definitions can be used not only in the API project, but also in client application projects to describe types in functions that work with the API. (Angular developers will be especially happy about this.)

5. Casting Query String Types

®

this:

```
param1=value&param2=777&param3=false
```

The same goes for query parameters (for example, in `GET` requests). In this case, the web server will fail to automatically recognize types: All data [will be in string format](#), so after parsing you will get this object:

```
{ param1: 'value', param2: '777', param3: 'false' }
```

In this case, the request will fail schema validation, so you need to verify the correct parameters' formats manually and cast them to the correct types.

As you can guess, you can do it with our good old schemas from the specification. Let's say we have this endpoint and the following schema:

```
# posts.endpoints.tinyspec
GET /posts?PostsQuery

# post.models.tinyspec
PostsQuery {
  search,
  limit: i,
  offset: i,
  filter: {
    isRead: b
  }
}
```

Here is how the request to this endpoint looks:

```
GET /posts?search=needle&offset=10&limit=1&filter[isRead]=true
```

®

```
function castQuery(query, schema) {
  _._mapValues(query, (value, key) => {
    const { type } = schema.properties[key] || {};

    if (!value || !type) {
      return value;
    }

    switch (type) {
      case 'integer':
        return parseInt(value, 10);
      case 'number':
        return parseFloat(value);
      case 'boolean':
        return value !== 'false';
      default:
        return value;
    }
  });
}
```

A fuller implementation with support for nested schemas, arrays, and `null` types is available in the [cast-with-schema](#) (npm) module. Now let's use it in our code:

```
router.get('/posts', async (ctx) => {
  // Cast parameters to expected types
  const query = castQuery(ctx.query, schemas.PostsQuery);

  // Run spec validation
  await validate(schemas.PostsQuery, query);

  // Query the database
  const posts = await Post.search(query);

  // Return serialized result
  ctx.body = { posts: serialize(posts, schemas.Post) };
});
```

Note that three of the four lines of code use specification schemas.



There are a number of best practices we can follow here.

Use Separate Create and Edit Schemas

Usually the schemas that describe server responses are different from those that describe inputs and are used to create and edit models. For example, the list of fields available in `POST` and `PATCH` requests must be strictly limited, and `PATCH` usually has all fields marked optional. The schemas that describe the response can be more freeform.

When you [generate CRUDL endpoints automatically](#), tinyspec uses `New` and `Update` postfixes. `User*` schemas can be defined in the following way:

```
User {id, email, name, isAdmin: b}  
UserNew !{email, name}  
UserUpdate !{email?, name?}
```

Try to not use the same schemas for different action types to avoid accidental security issues due to the reuse or inheritance of older schemas.

Follow Schema Naming Conventions

The content of the same models may vary for different endpoints. Use `With*` and `For*` postfixes in schema names to show the difference and purpose. In tinyspec, models can also inherit from each other. For example:

```
User {name, surname}  
UserWithPhotos < User {photos: Photo[]}  
UserForAdmin < User {id, email, lastLoginAt: d}
```




Separating Endpoints Based on Client Type

Often the same endpoint returns different data based on client type, or the role of the user who sent the request. For example, the `GET /users` and `GET /messages` endpoints can be significantly different for mobile application users and back office managers. The change of the endpoint name can be overhead.

To describe the same endpoint multiple times you can add its type in parentheses after the path. This also makes tag use easy: You split endpoint documentation into groups, each of which is intended for a specific API client group. For example:

```
Mobile app:
  GET /users (mobile)
    => UserForMobile[]

CRM admin panel:
  GET /users (admin)
    => UserForAdmin[]
```

REST API Documentation Tools

After you get the specification in tinyspec or OpenAPI format, you can generate nice-looking documentation in HTML format and publish it. This will make developers who use your API happy, and it sure beats filling in a REST API documentation template by hand.

Apart from the cloud services mentioned earlier, there are CLI tools that convert OpenAPI 2.0 to HTML and PDF, which can be deployed to any static hosting. Here are some examples:

- [bootprint-openapi](#) (npm, used by default in tinyspec)



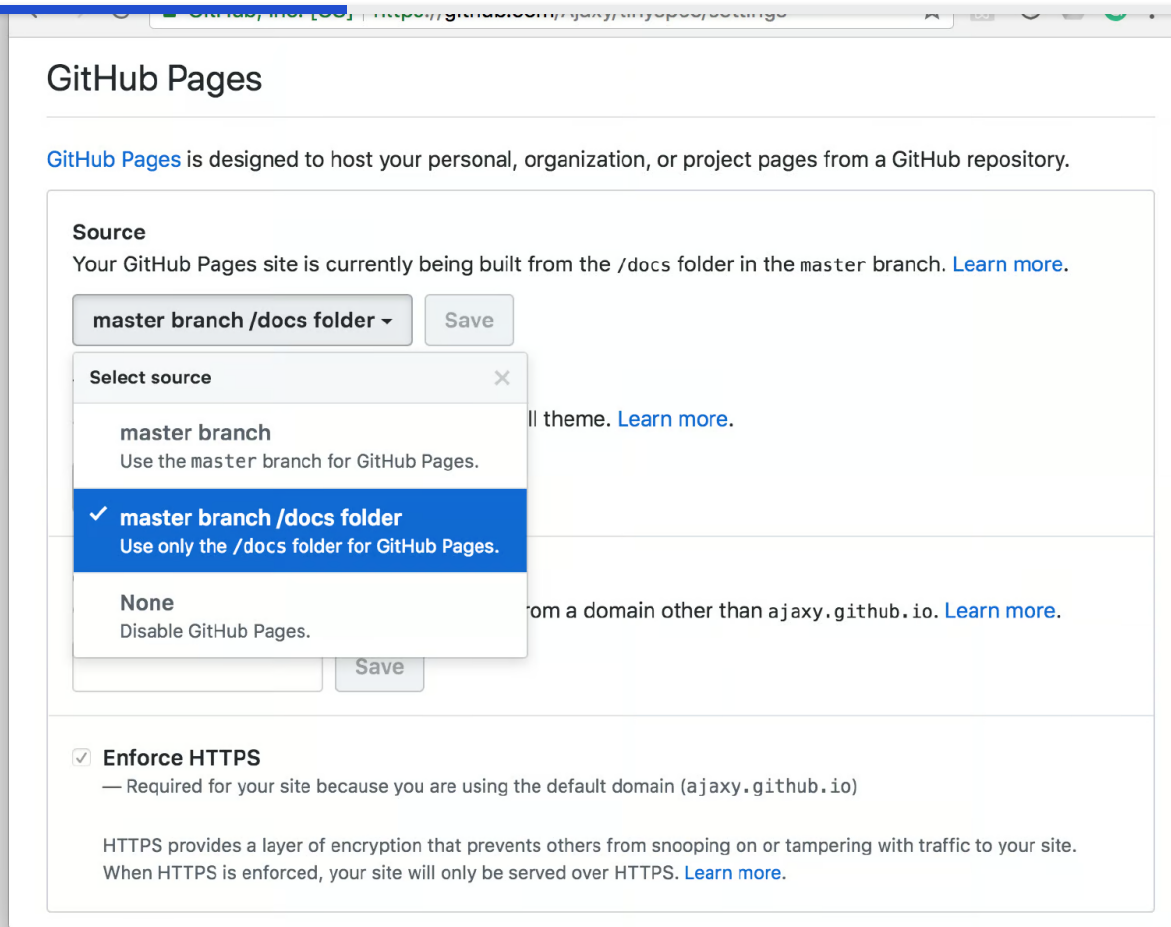
- [redoc-cli](#) (npm)
- [widdershins](#) (npm)

Do you have more examples? Share them in the comments.

Sadly, despite being released a year ago, OpenAPI 3.0 is still poorly supported and I failed to find proper examples of documentation based on it both in cloud solutions and in CLI tools. For the same reason, tinyspec does not support OpenAPI 3.0 yet.

Publishing on GitHub

One of the simplest ways to publish the documentation is [GitHub Pages](#). Just enable support for static pages for your `/docs` folder in the repository settings and store HTML documentation in this folder.



You can add the command to generate documentation through tinyspec or a different CLI tool in your `scripts/package.json` file to update the documentation automatically after each commit:

```
"scripts": {
  "docs": "tinyspec -h -o docs/",
  "precommit": "npm run docs"
}
```

Continuous Integration

You can add documentation generation to your CI cycle and publish it, for example, to Amazon S3 under different addresses depending on the environment or API version (like `/docs/2.0`, `/docs/stable`, and `/docs/staging`.)



If you like the tinyspec syntax, you can become an early adopter for tinyspec.cloud. We plan to build a cloud service based on it and a CLI for automated deployment of documentation with a wide choice of templates and the ability to develop personalized templates.

REST Specification: A Marvelous Myth

REST API development is probably one of the most pleasant processes in modern web and mobile services development. There are no browser, operating system, and screen-size zoos, and everything is fully under your control, at your fingertips.

This process is made even easier by the support for automation and up-to-date specifications. An API using the approaches I've described becomes well-structured, transparent, and reliable.

The bottom line is, if we are making a myth, why not make it a marvelous myth?

Related: [ActiveResource.js ORM: Building a Powerful JavaScript SDK For Your JSON API, Fast](#)

UNDERSTANDING THE BASICS

^ What is REST?

REST is a web service architectural style defining a set of required constraints. It is based around resources with unique identifiers (URIs) and the operations with said resources. Additionally, a REST specification requires a client-server model, a uniform interface, and the absence of server-stored state.

∨ What is the OpenAPI specification?

∨ What is Swagger?

∨ What is JSON Schema?

∨ What exactly is an API?



-
- ✓ What makes a RESTful API?
 - ✓ What is meant by behavior-driven development?
 - ✓ What is tinyspec?

TAGS

RubyOnRails

REST

OpenAPI

Node.js

Turn insights into action.

Hire a Toptal Expert



Alexander Zinchuk

 **Verified Expert** in Engineering

Located in Barcelona, Spain

Member since September 15, 2016

ABOUT THE AUTHOR

Alex's decade-plus of JS coding has taught him the language's internals. He's lead a dev team for Yandex and built fault-tolerant systems.



EXPERTISE

API

YEARS OF EXPERIENCE

18

[Hire Alexander](#)

TRENDING ARTICLES

[ENGINEERING](#) > [DATA SCIENCE AND DATABASES](#)

Advantages of AI: Using GPT and Diffusion Models for Image Generation

[ENGINEERING](#) > [DATA SCIENCE AND DATABASES](#)

Ask an NLP Engineer: From GPT Models to the Ethics of AI

[ENGINEERING](#) > [TECHNOLOGY](#)

How to Use JWT and Node.js for Better App Security

[ENGINEERING](#) > [WEB FRONT-END](#)

Next.js vs. React: A Comparative Tutorial

SEE OUR RELATED TALENT

[API Developers](#)



World-class articles, delivered weekly.

Enter your email

Sign Me Up

Subscription implies consent to our [privacy policy](#).

Toptal Developers

Algorithm Developers

Angular Developers

AWS Developers

Azure Developers

Big Data Architects

Blockchain Developers

Business Intelligence
Developers

C Developers

Computer Vision Developers

Django Developers

Docker Developers

Elixir Developers

Go Engineers

GraphQL Developers

Jenkins Developers

Kotlin Developers

Kubernetes Experts



.NET Developers

SQL Developers

[View More](#)

R Developers

Sys Admins

[Freelance Developers →](#)

React Native Developers

Tableau Developers

Join the Toptal® community.

Hire a Developer

or

Apply as a Developer

ON-DEMAND TALENT

Hire Freelance Developers

Hire Freelance Designers

Hire Freelance Finance Experts

Hire Freelance Project Managers

Hire Freelance Product Managers

MANAGEMENT CONSULTING

Strategy Consulting

People & Organization Consulting

Innovation & Experience Consulting

TECHNOLOGY SERVICES

Application Services

Cloud Services

Information Security Services

Quality Assurance Services

[Top 3%](#)[Clients](#)[Freelance Jobs](#)[Specialized Services](#)[Utilities & Tools](#)[Research & Analysis Center](#)[About Us](#)[Contact Us](#)[Press Center](#)[Careers](#)[FAQ](#)

The World's Top Talent, On Demand ®



Copyright 2010 - 2023 Toptal, LLC

[Privacy Policy](#) [Website Terms](#) [Accessibility](#)