meta-paradigm

with JavaScript

JavaScript Turns 25

Vue.js 3.0 “One Piece” release

Decorators: a new proposal iteration

Rome first beta release

ECMAScript 2020

Playwright announced

WebStorm turns 10

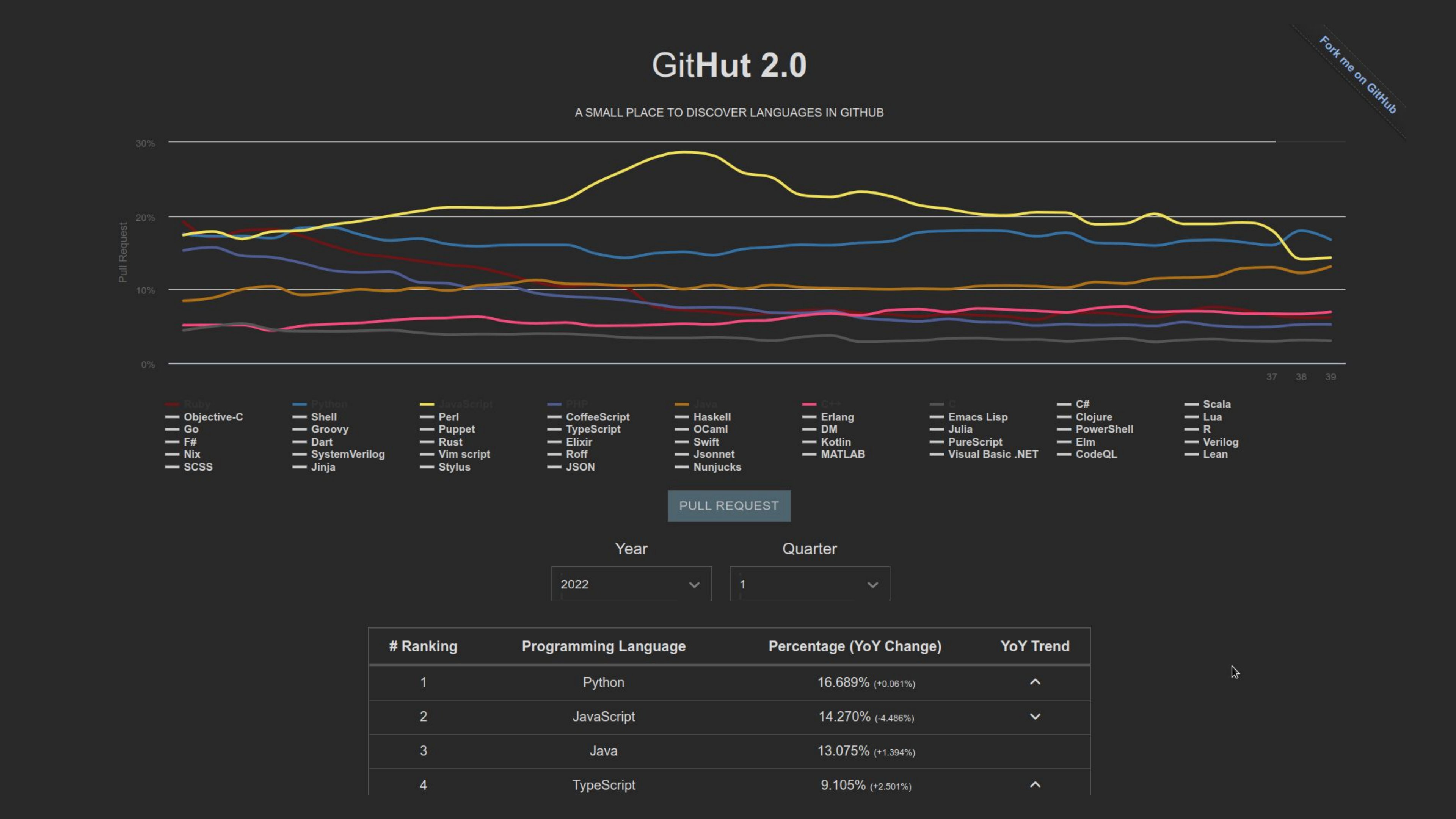
Angular 9.0.0 with Ivy

Deno 1.0 released

Deno finally sees the light of day with its first public release. It’s still not clear whether it’s going to become the next big thing but the hopes are high.

JavaScript makes it into space

The SpaceX Dragon launch brings JavaScript to space! The Dragon 2 flight interface was built using Chromium and JavaScript along with C++ for flight



GitHut 2.0

A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB

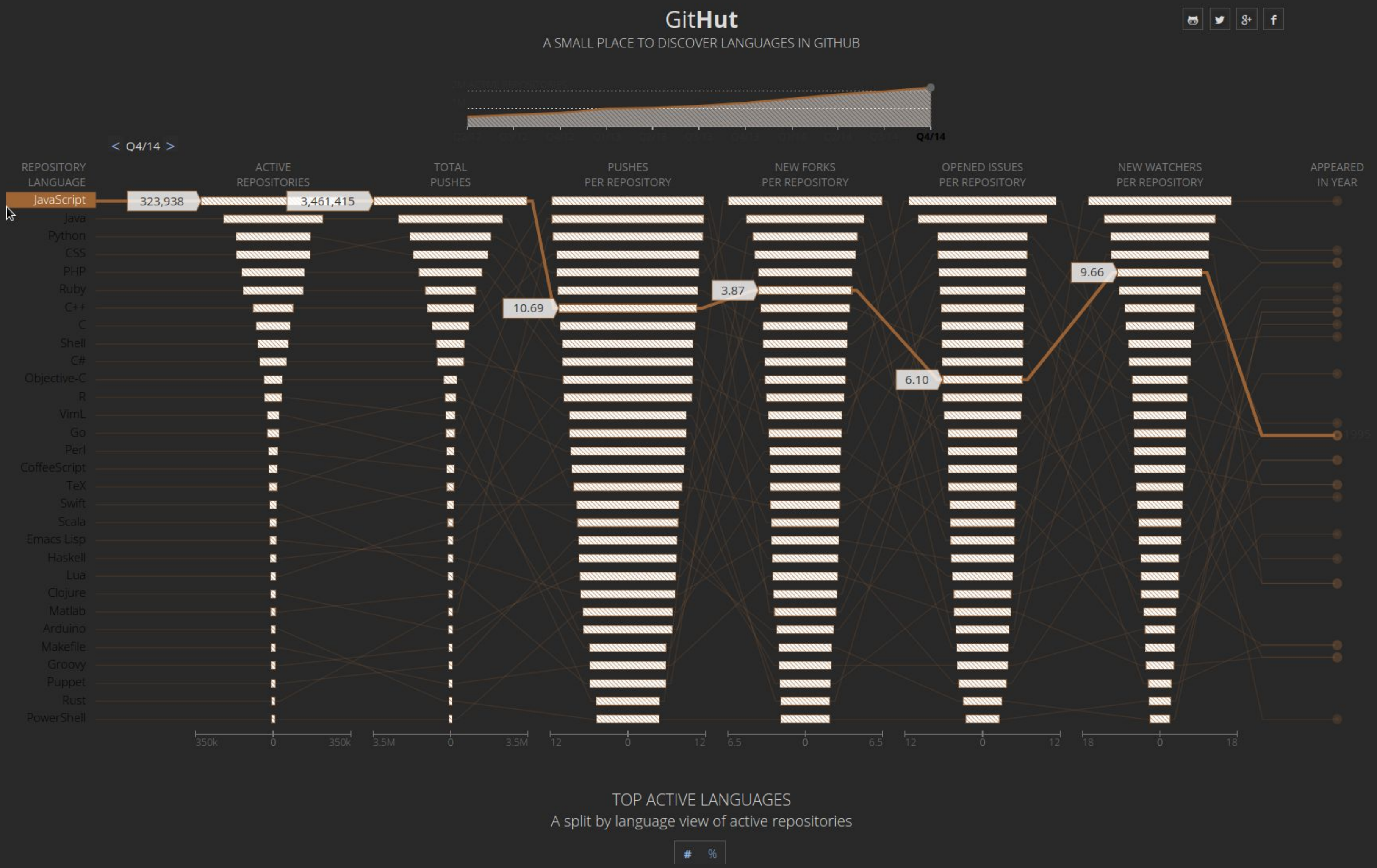
Year

2022

Quarter

1

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	Python	16.689% (+0.061%)	⌃
2	JavaScript	14.270% (-4.486%)	⌄
3	Java	13.075% (+1.394%)	
4	TypeScript	9.105% (+2.501%)	⌃

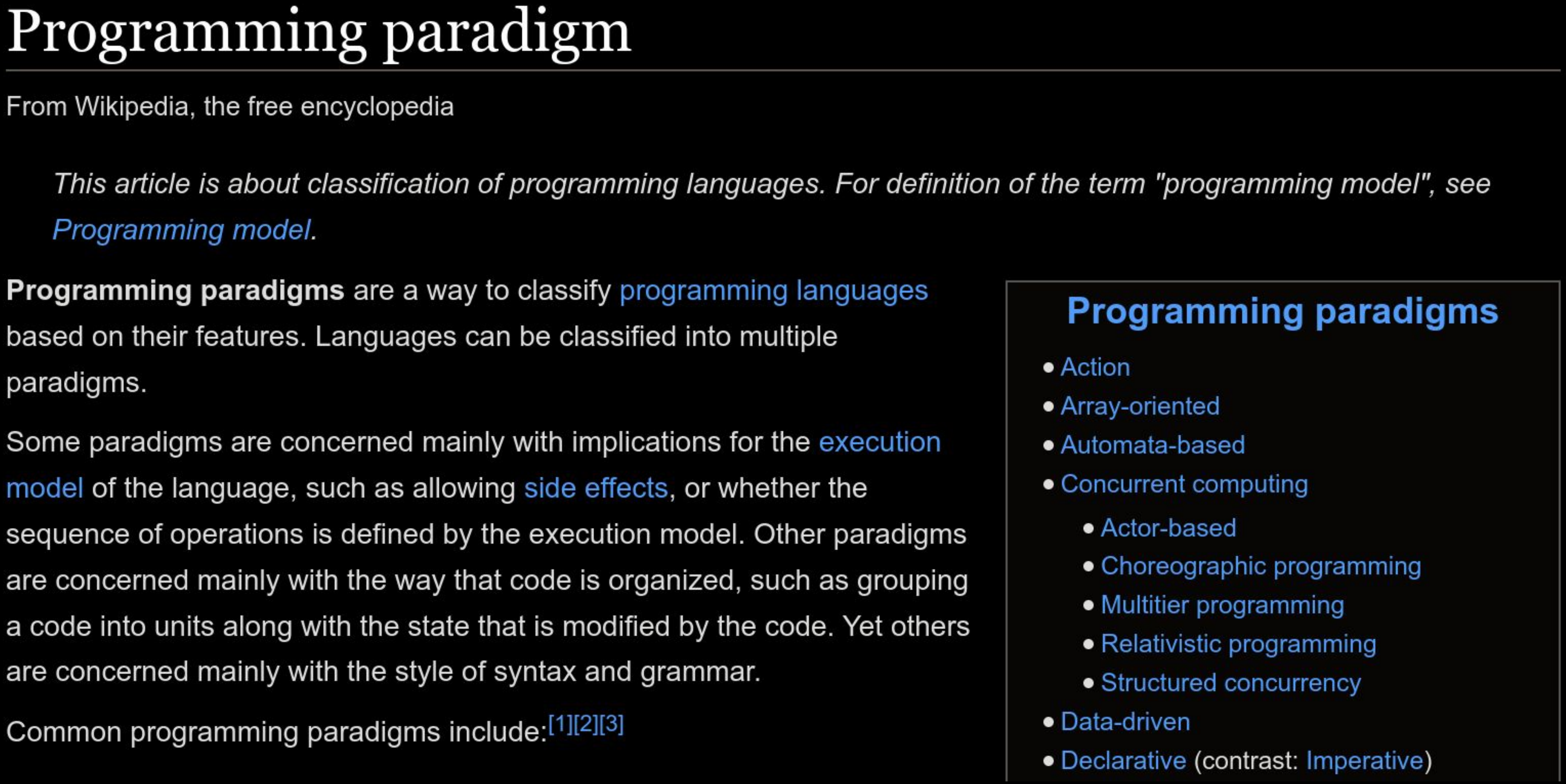


GitHut

A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB

TOP ACTIVE LANGUAGES

A split by language view of active repositories



Programming paradigm

From Wikipedia, the free encyclopedia

This article is about classification of programming languages. For definition of the term "programming model", see Programming model.

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

Common programming paradigms include:[1][2][3]

Programming paradigms

- Action
- Array-oriented
- Automata-based
- Concurrent computing
  - Actor-based
  - Choreographic programming
  - Multitier programming
  - Relativistic programming
  - Structured concurrency
- Data-driven
- Declarative (contrast: Imperative)

demo

demo

```
class FirstClass extends BaseClass {
  constructor () {
     super ();
     this . myField = mySpecialField;
  }
}
```

demo

```
class SecondClass extends BaseClass {
  constructor () {
    super ();
    this . myField = mySpecialField;
  }
}
```

demo

```
'use strict';

const { BaseClass } = require ('typeomatica');

const { mySpecialField } = require ('./code/fields');
```

demo

```
class FirstClass extends BaseClass {
  constructor () {
      super ();
      this . myField = mySpecialField;
  }
}
```

demo

```
class SecondClass extends BaseClass {
  constructor () {
    super ();
    this . myField = mySpecialField;
  }
}
```

demo

```
const firstInstance = new FirstClass ();
const secondInstance = new SecondClass ();
```

demo

```
const firstInstance = new FirstClass ();
const secondInstance = new SecondClass ();
// will print → 'initial value'
console . log( firstInstance . myField );
```

demo

```
const firstInstance = new FirstClass ();
const secondInstance = new SecondClass ();
// will print → 'initial value'
console . log( firstInstance . myField );
// will print → 'initial value'
console . log( secondInstance . myField );
```

demo

```
firstInstance . myField = 're-assigned value';
```

demo

```
firstInstance . myField = 're-assigned value';
// will print → 're-assigned value'
console . log( firstInstance . myField );
```

demo

```
firstInstance . myField = 're-assigned value';
// will print → 're-assigned value'
console . log( firstInstance . myField );
// expectations → 'initial value'
console . log( secondInstance . myField );
```

demo

```
 firstInstance . myField = 're-assigned value';
 // will print → 're-assigned value'
 console . log( firstInstance . myField );
// expectations → 'initial value'
console . log( secondInstance . myField );
// but will also print → 're-assigned value' !
```

demo

```
firstInstance . myField = 're-assigned value';
// will print → 're-assigned value'
console . log( firstInstance . myField );
// expectations → 'initial value'
console . log( secondInstance . myField );
// but will also print → 're-assigned value' !
```

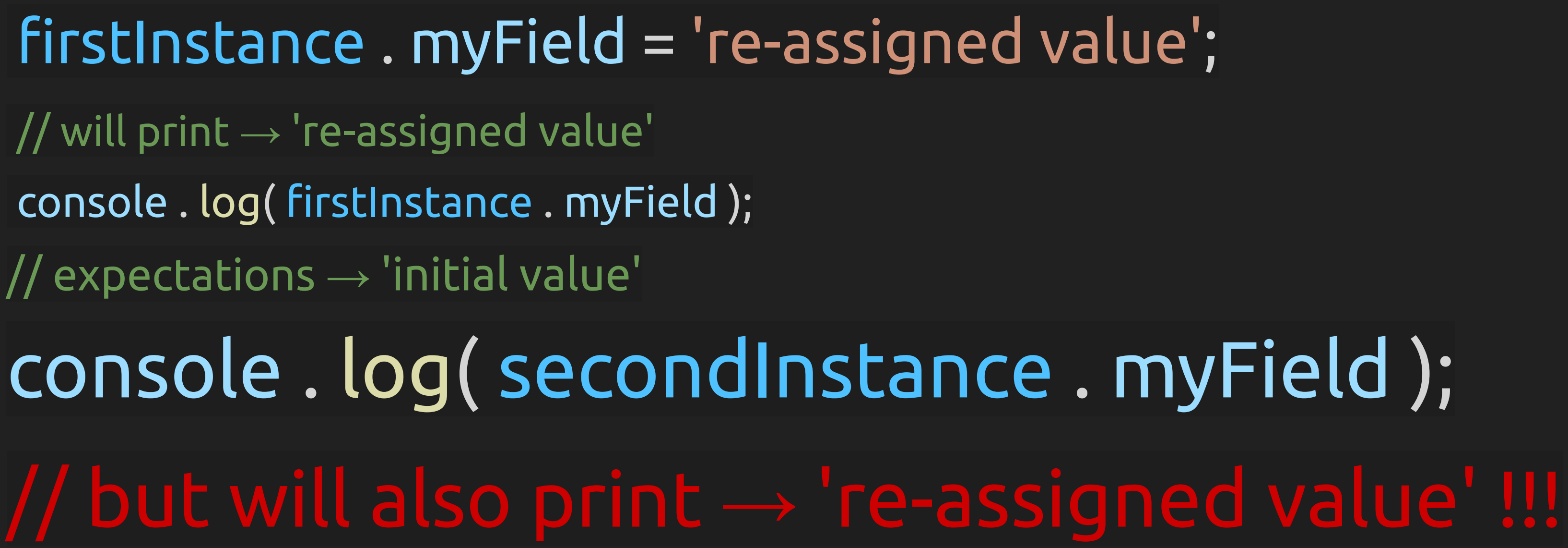
demo

```
class FirstClass extends BaseClass {
  constructor () {
     super ();
     this . myField = mySpecialField;
  }
}
```

demo

```
class SecondClass extends BaseClass {
  constructor () {
    super ();
    this . myField = mySpecialField;
  }
}
```

demo



```
firstInstance . myField = 're-assigned value';
// will print → 're-assigned value'
console . log( firstInstance . myField );
// expectations → 'initial value'
console . log( secondInstance . myField );
// but will also print → 're-assigned value' !!!
```

demø

intro

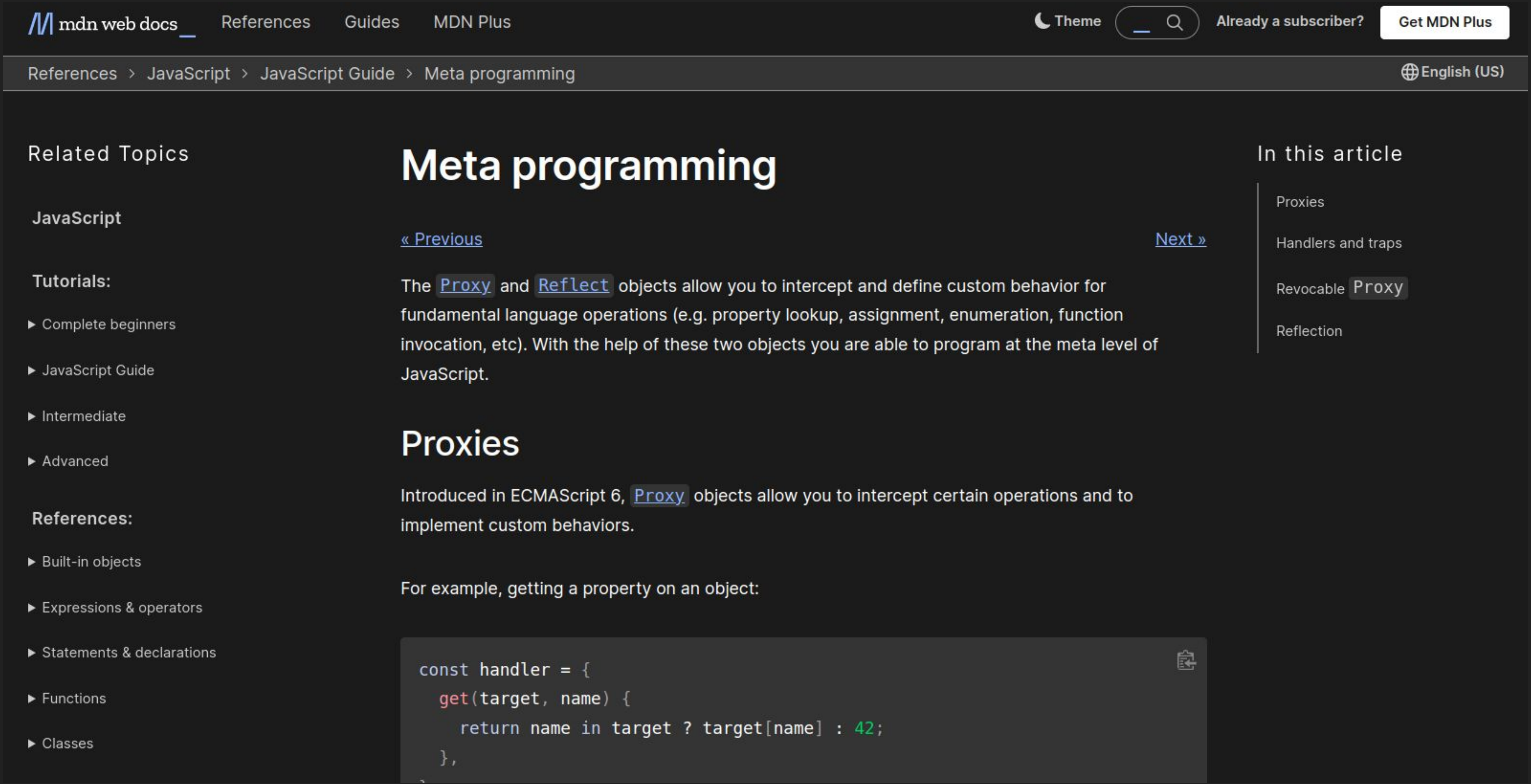
The Magic of
Prototype Inheritance

Strict Types in JavaScript

hitchhiker's guide

Multiplie Inheritance
in JavaScript

intrø



Meta programming

« Previous

Next »

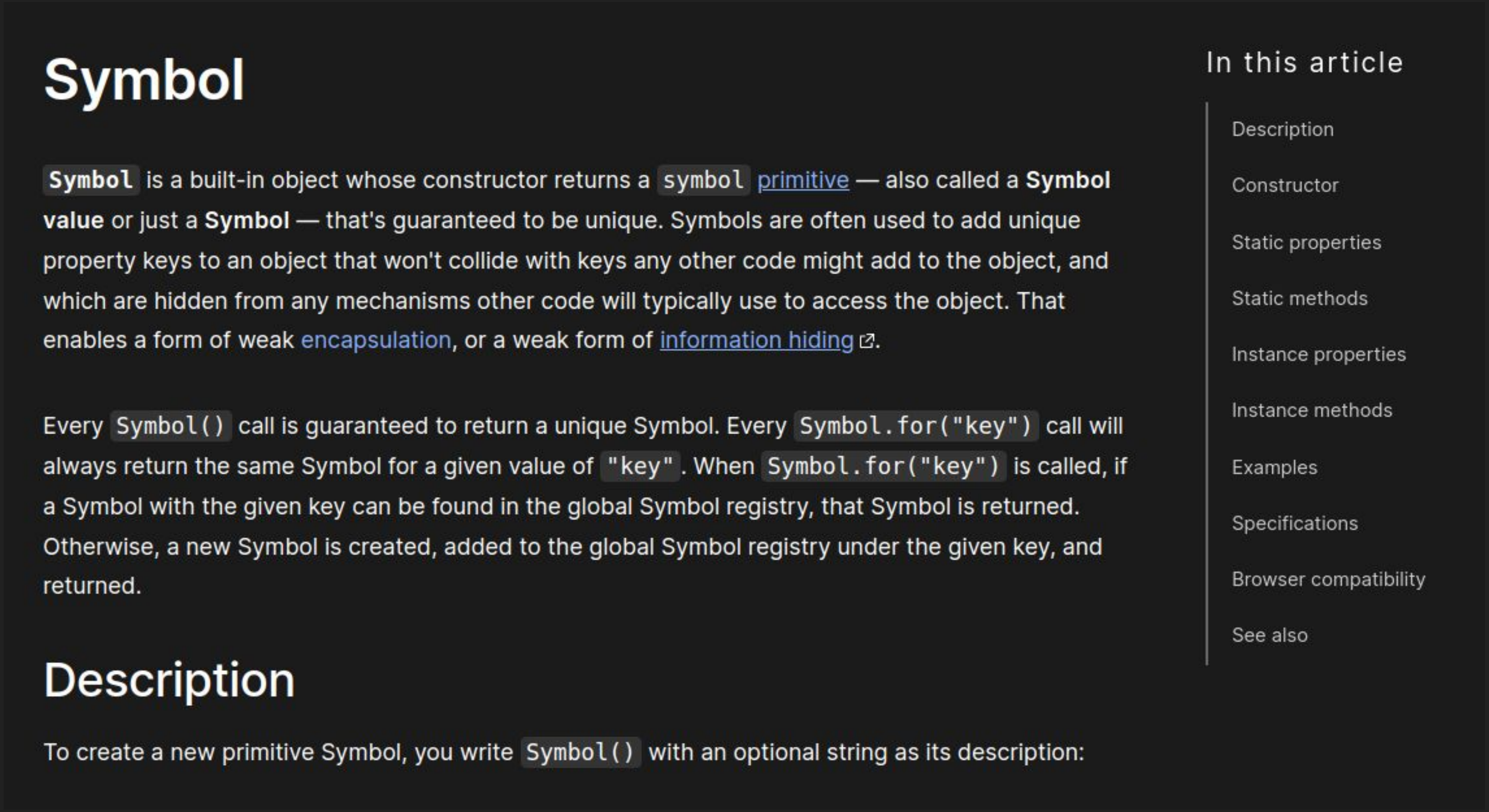
The `Proxy` and `Reflect` objects allow you to intercept and define custom behavior for fundamental language operations (e.g. property lookup, assignment, enumeration, function invocation, etc). With the help of these two objects you are able to program at the meta level of JavaScript.

Proxies

Introduced in ECMAScript 6, `Proxy` objects allow you to intercept certain operations and to implement custom behaviors.

For example, getting a property on an object:

```
const handler = {
  get(target, name) {
    return name in target ? target[name] : 42;
  },
```



Symbol

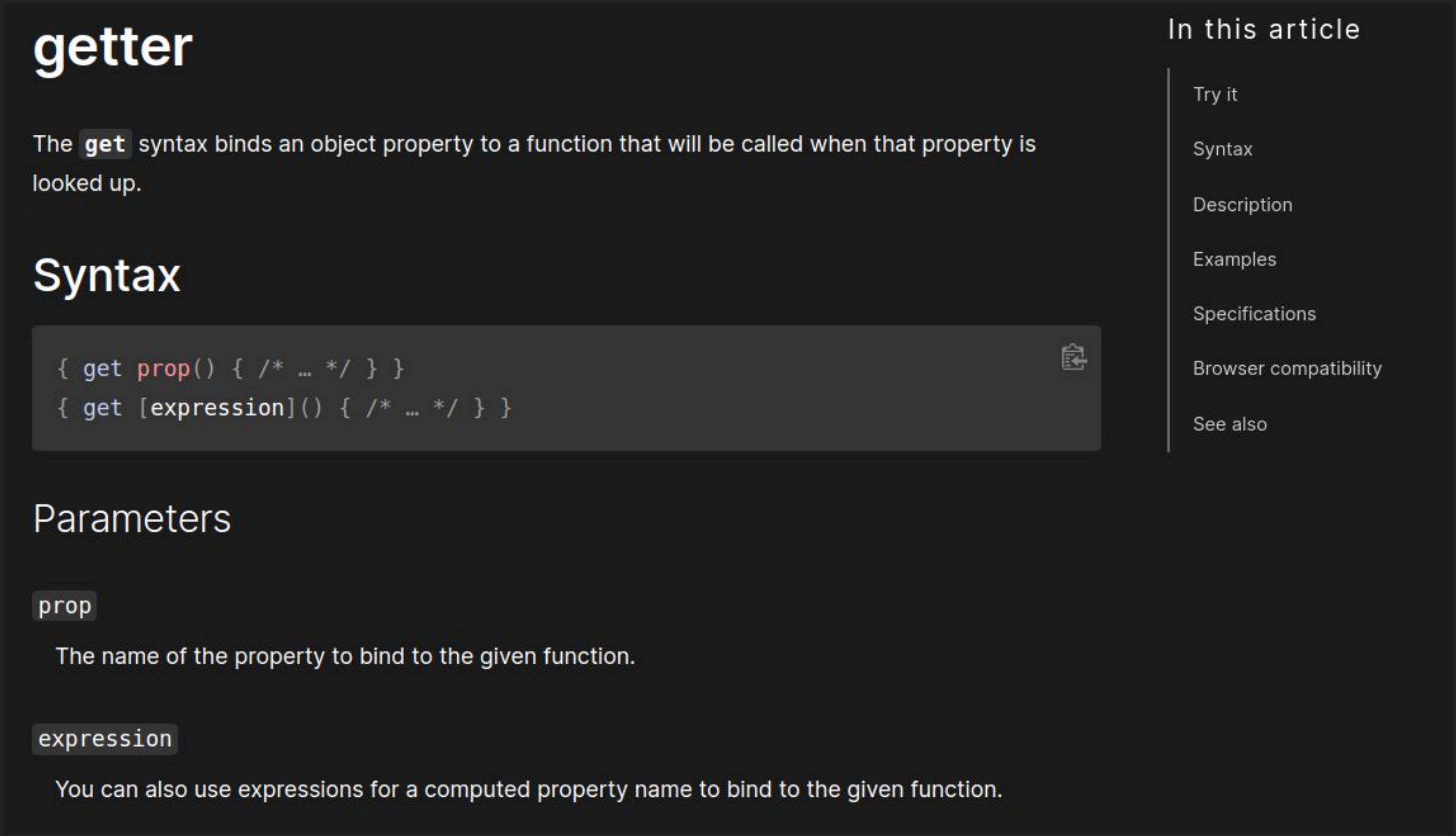
`Symbol` is a built-in object whose constructor returns a `symbol` primitive — also called a **Symbol value** or just a **Symbol** — that's guaranteed to be unique. Symbols are often used to add unique property keys to an object that won't collide with keys any other code might add to the object, and which are hidden from any mechanisms other code will typically use to access the object. That enables a form of weak encapsulation, or a weak form of information hiding.

Every `Symbol()` call is guaranteed to return a unique Symbol. Every `Symbol.for("key")` call will always return the same Symbol for a given value of `"key"`. When `Symbol.for("key")` is called, if a Symbol with the given key can be found in the global Symbol registry, that Symbol is returned. Otherwise, a new Symbol is created, added to the global Symbol registry under the given key, and returned.

Description

To create a new primitive Symbol, you write `Symbol()` with an optional string as its description:

In this article



getter

The `get` syntax binds an object property to a function that will be called when that property is looked up.

Syntax

```
{ get prop() { /* … */ } }
{ get [expression]() { /* … */ } }
```

Parameters

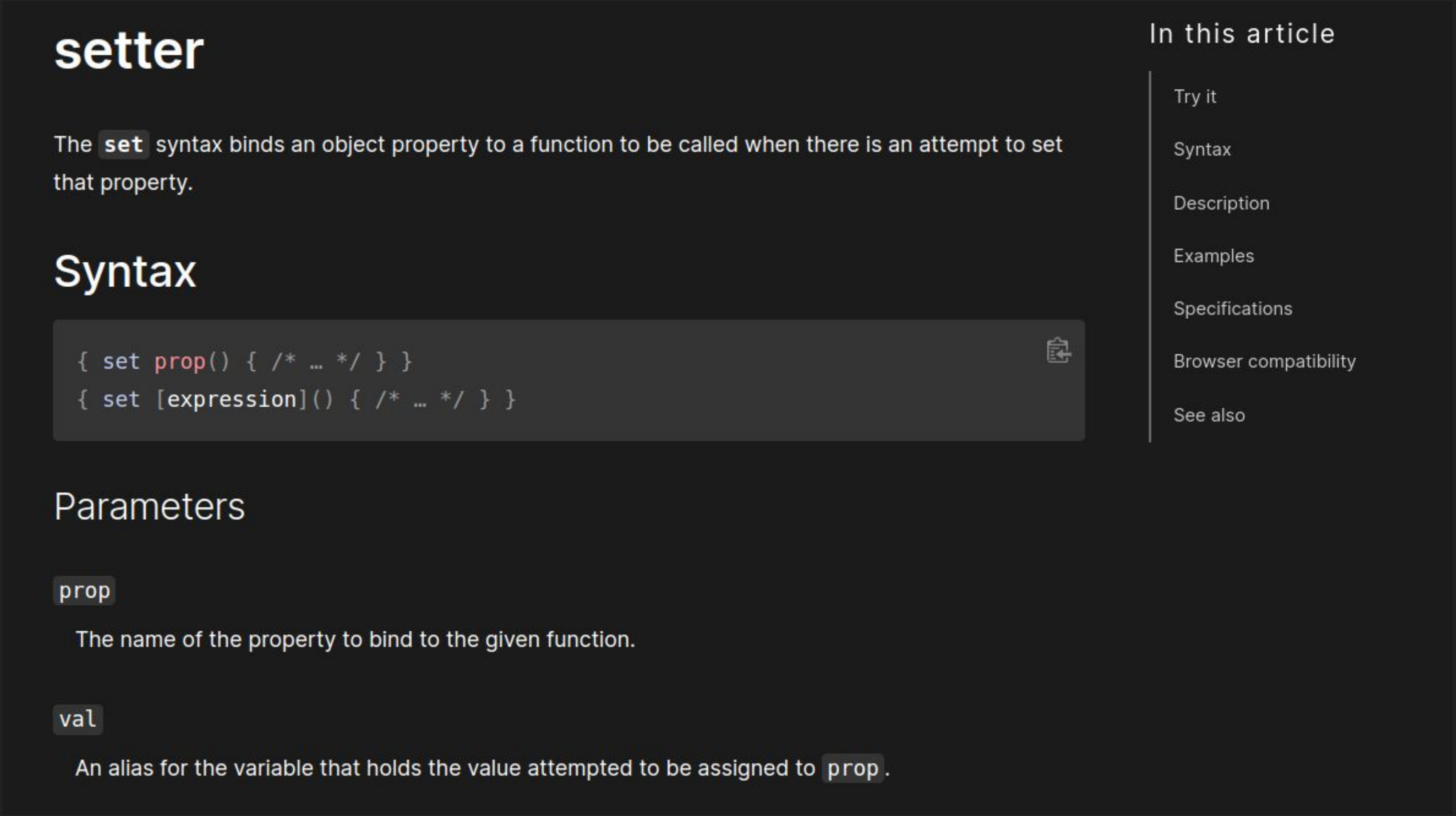
`prop`

The name of the property to bind to the given function.

`expression`

You can also use expressions for a computed property name to bind to the given function.

In this article



setter

The `set` syntax binds an object property to a function to be called when there is an attempt to set that property.

Syntax

```
{ set prop() { /* … */ } }
{ set [expression]() { /* … */ } }
```

Parameters

`prop`

The name of the property to bind to the given function.

`val`

An alias for the variable that holds the value attempted to be assigned to `prop`.

In this article

demo

demo

```
const firstInstance = new FirstClass ();
const secondInstance = new SecondClass ();

firstInstance . myField = 're-assigned value';
// will print → 're-assigned value'
console . log( secondInstance . myField );
```

```
class ExtendedClass extends BasePrototype ( secondInstance ) {
  constructor () {
    super ();
    this . myField = myExtendedField;   // words order reverted
  }
}
const thirdInstance = new ExtendedClass ();
// will print → 'value re-assigned'
console . log( 'thirdInstance  : ',  thirdInstance . myField );
```

```
class ExtendedClass extends BasePrototype ( secondInstance ) {
  constructor () {
     super ();
     this . myField = myExtendedField;   // words order reverted
  }
}
const thirdInstance = new ExtendedClass ();
// will print → 'value re-assigned'
console . log( 'thirdInstance  : ',  thirdInstance . myField );
```

```
class ExtendedClass extends BasePrototype ( secondInstance ) {
  constructor () {
     super ();
     this . myField = myExtendedField;   // words order reverted
  }
}
const thirdInstance = new ExtendedClass ();
// will print → 'value re-assigned'
console . log( 'thirdInstance  : ',  thirdInstance . myField );
```

```
class ExtendedClass extends BasePrototype ( secondInstance ) {
  constructor () {
     super ();
     this . myField = myExtendedField;   // words order reverted
  }
}
const thirdInstance = new ExtendedClass ();
// will print → 'value re-assigned'
console . log( 'thirdInstance  : ',  thirdInstance . myField );
```

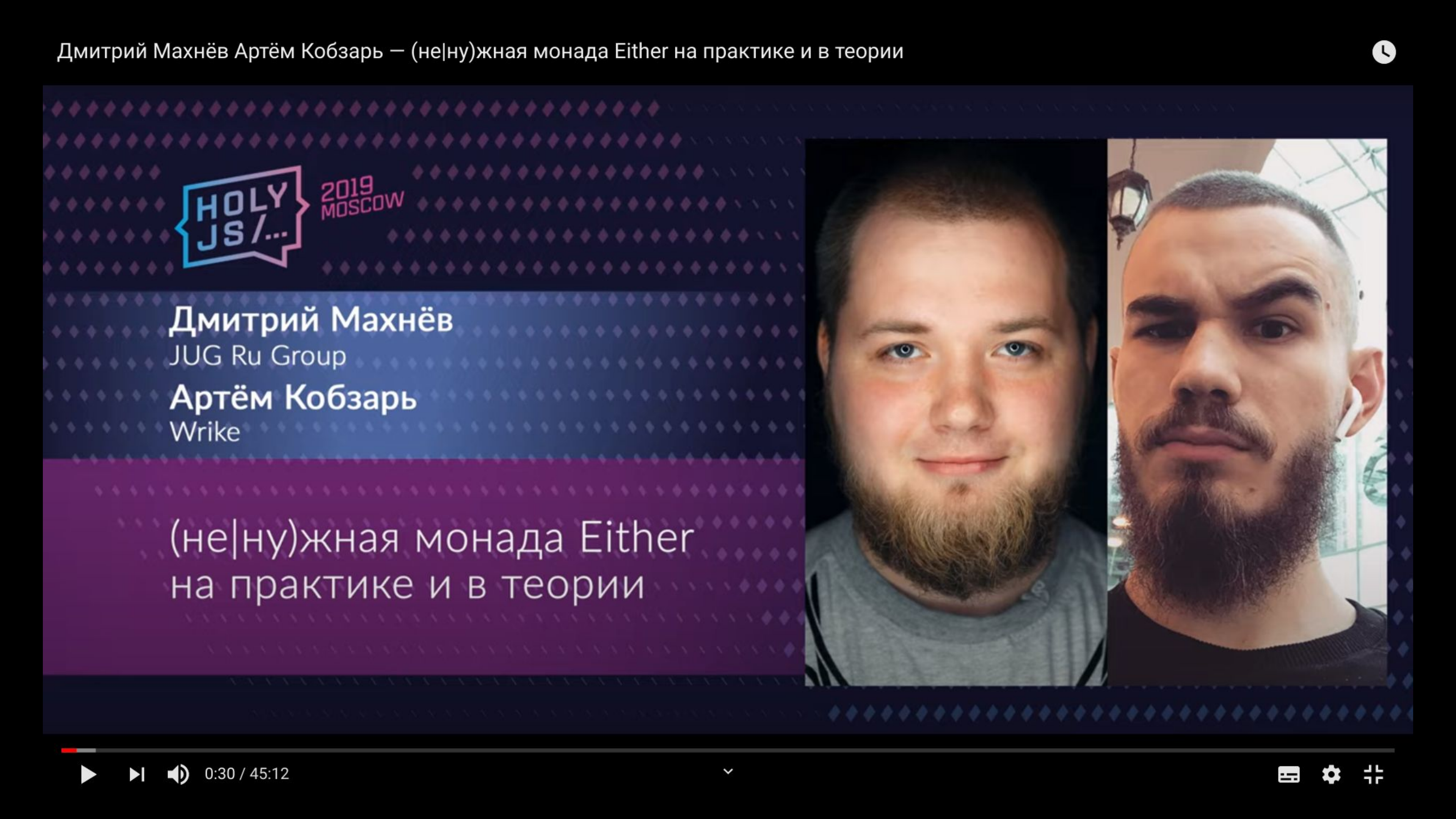
demo and …

demo

```
try {
    thirdInstance . myField = 123;
} catch ( error ) {
  console . error ( error );
}
```

demø

fin



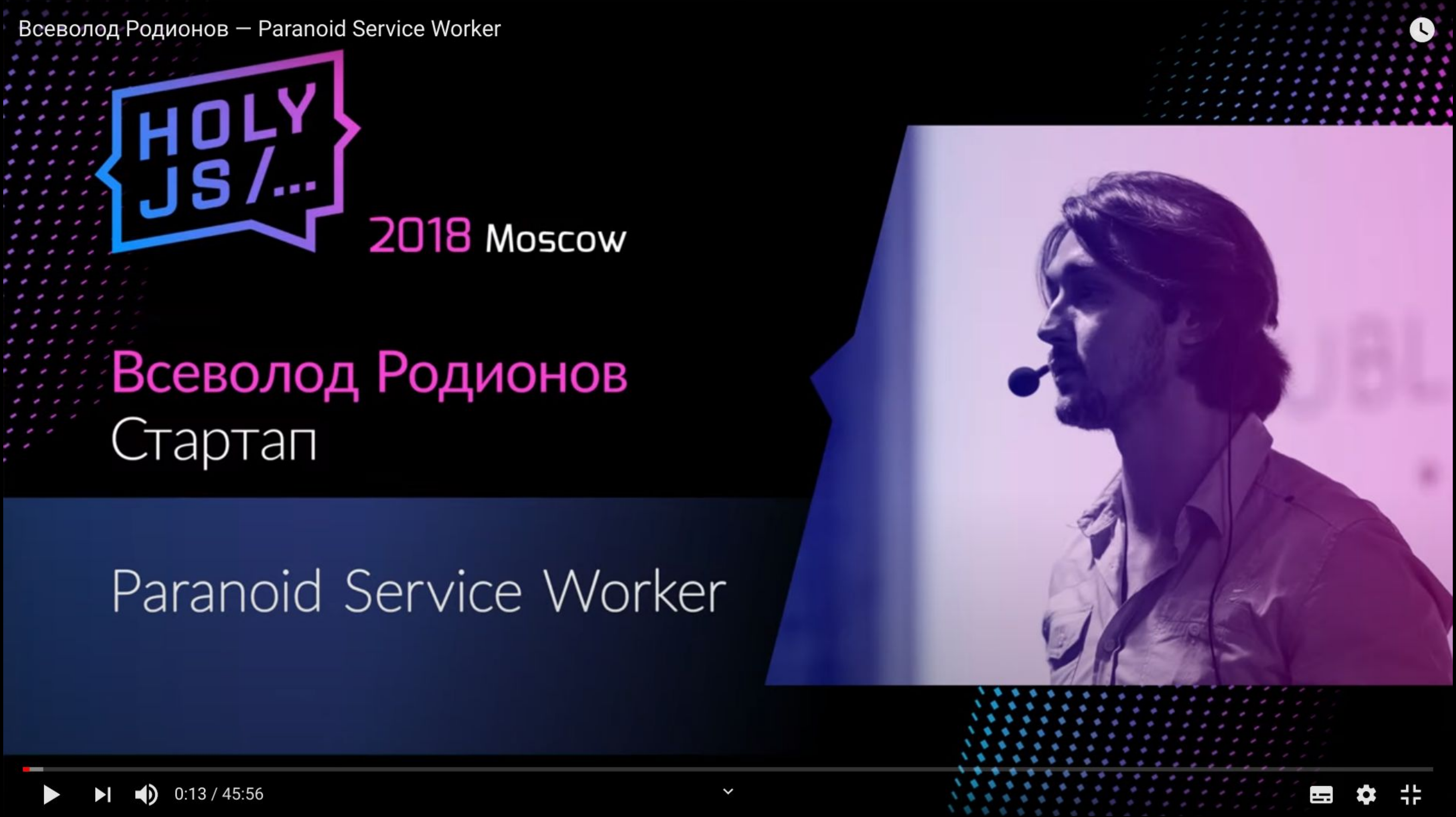
Дмитрий Махнёв Артём Кобзарь — (не|ну)жная монада Either на практике и в теории

HOLY JS /... 2019 MOSCOW

Дмитрий Махнёв
JUG Ru Group

Артём Кобзарь
Wrike

(не|ну)жная монада Either
на практике и в теории



Всеволод Родионов — Paranoid Service Worker

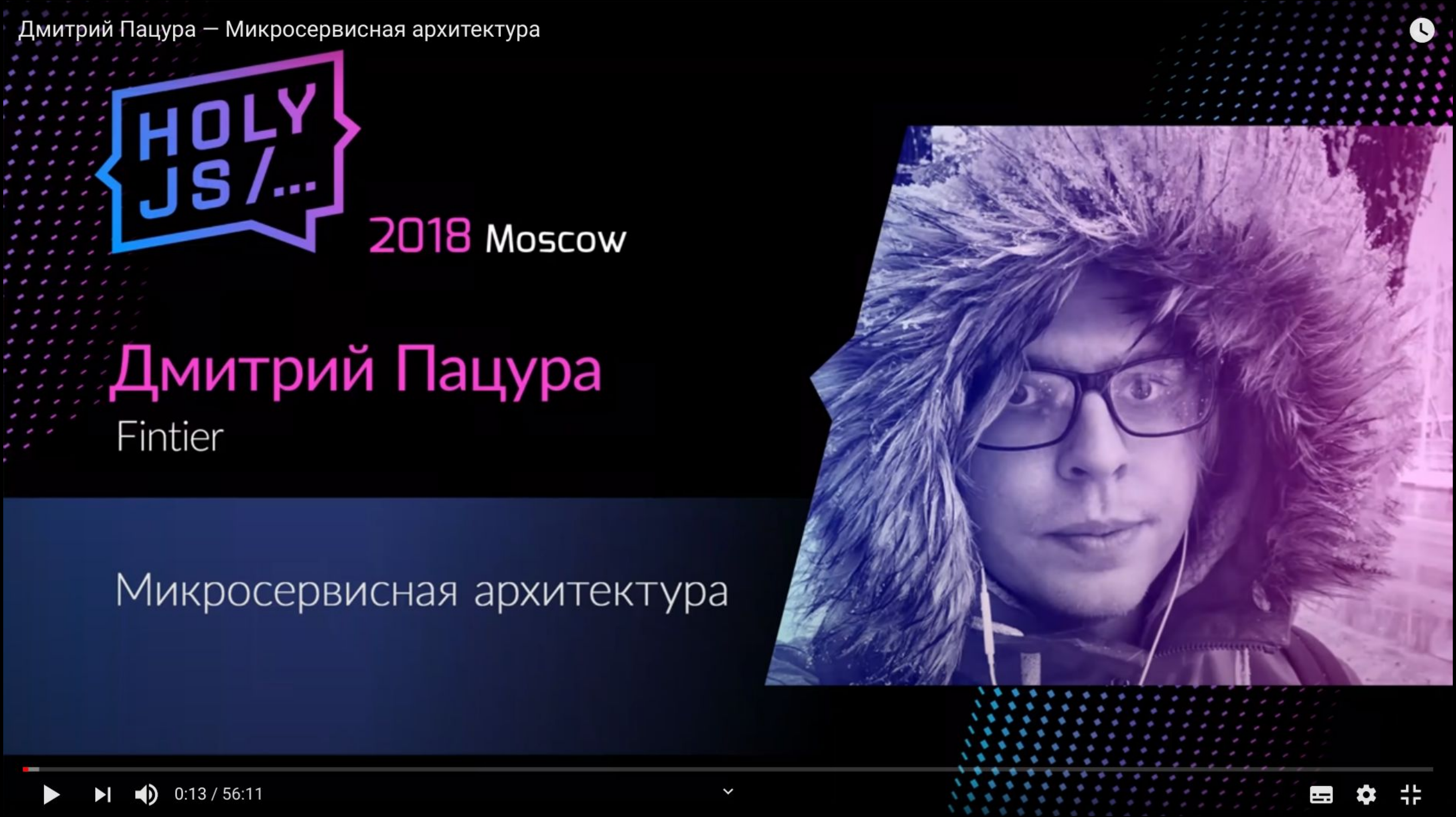
HOLY JS /...

2018 Moscow

Всеволод Родионов

Стартап

Paranoid Service Worker



Дмитрий Пацура — Микросервисная архитектура

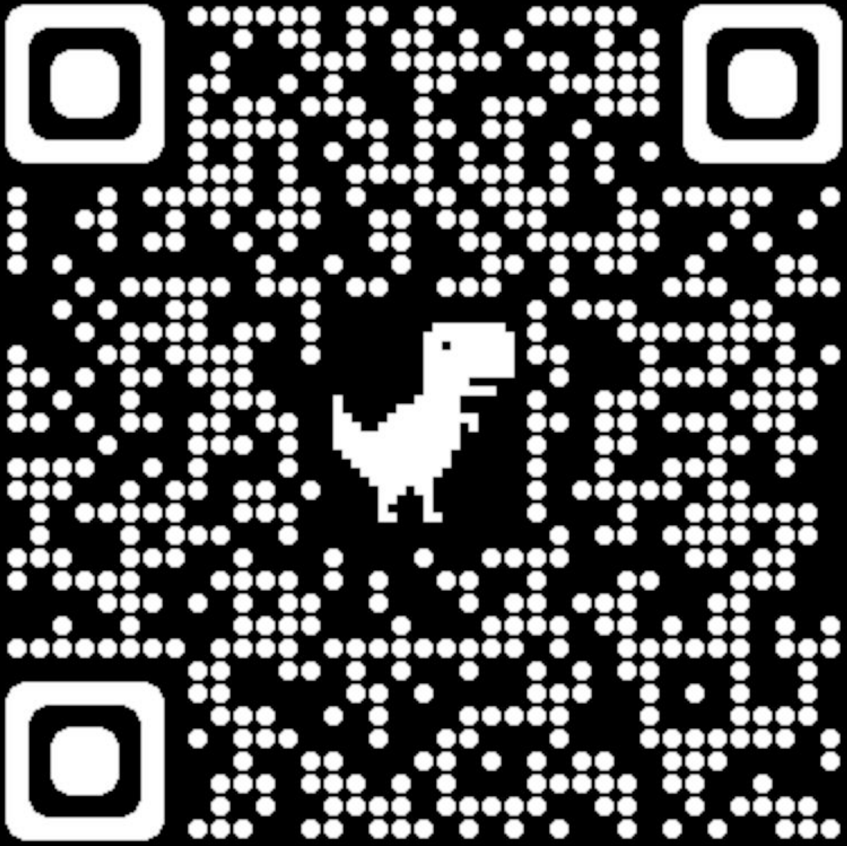
HOLY JS/...

2018 Moscow

Дмитрий Пацура

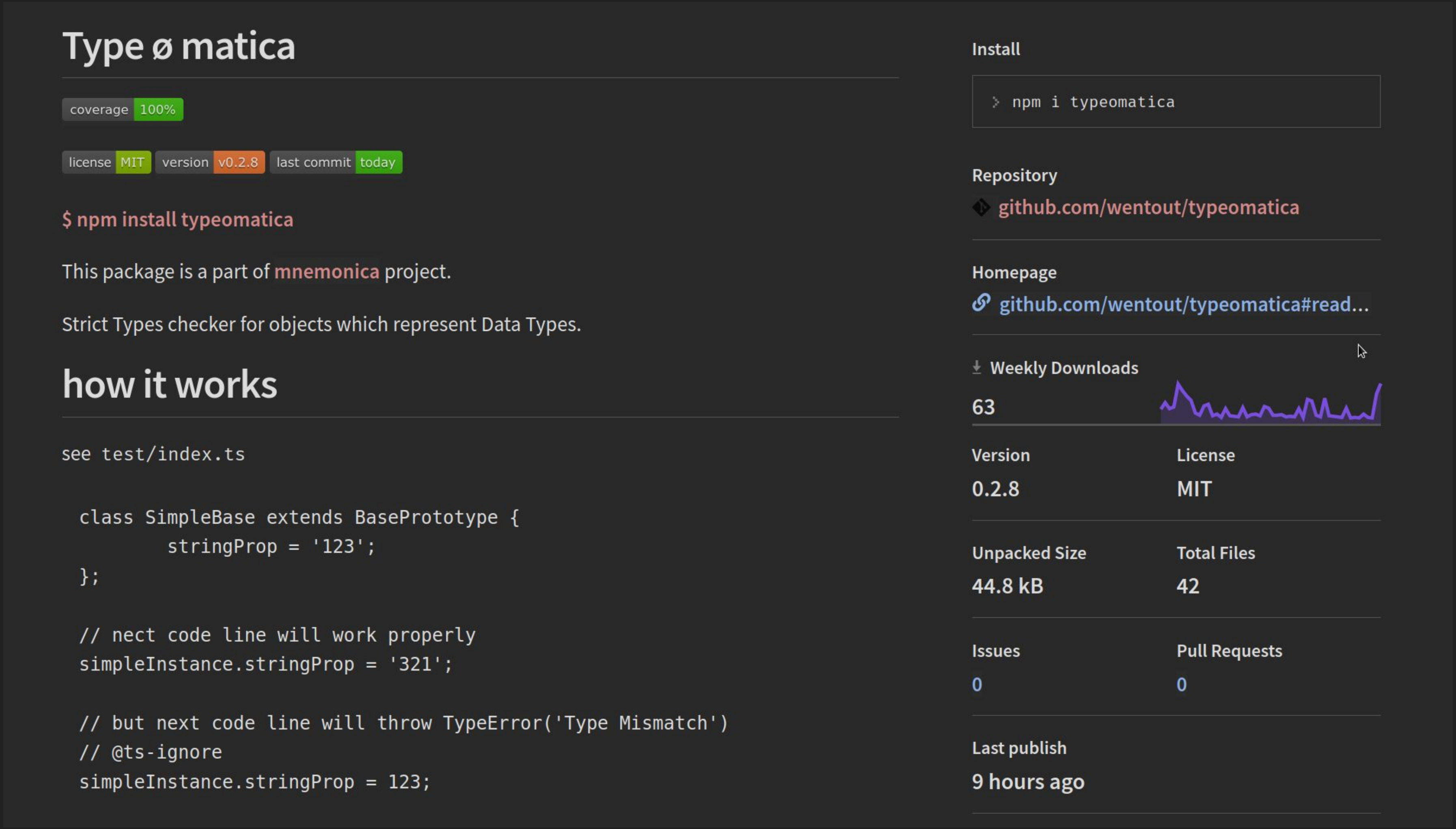
Fintier

Микросервисная архитектура



¿ Q & Æ ?

...



Type ø matica

coverage 100%

license MIT version v0.2.8 last commit today

$ npm install typeomatica

This package is a part of mnemonica project.

Strict Types checker for objects which represent Data Types.

how it works

```
see test/index.ts

  class SimpleBase extends BasePrototype {
          stringProp = '123';
  };

  // nect code line will work properly
  simpleInstance.stringProp = '321';

  // but next code line will throw TypeError('Type Mismatch')
  // @ts-ignore
  simpleInstance.stringProp = 123;
```

Install

```
> npm i typeomatica
```

Repository

github.com/wentout/typeomatica

Homepage

github.com/wentout/typeomatica#read...

Weekly Downloads

63

Version	License
0.2.8	MIT

Unpacked Size	Total Files
44.8 kB	42

Issues	Pull Requests
0	0

Last publish

9 hours ago