REST API

001

01 REST API

WHAT IS A REST API?

A **REST** is an application programming interface **(API)** that uses a representational state transfer (REST) architectural style.

02 WHAT IS API?

WHAT IS API?

[1] **A**pplication **P**rogramming **I**nterface

[2] Software intermediary that allows communication between two separate applications.

Each time you use an app like Facebook, send an instant message, or check the weather on your phone, you’re using an API.

WHAT IS API?

EXAMPLE

When you use a mobile application, it connects to the internet and transmits data to a server. The server interprets this data, executes required actions, and returns processed information to your phone. The application then presents this data in a readable format, facilitating your interaction. This entire process occurs through an API.

03

REST

ARCHITECTURE STYLE

REST ARCHITECTURE STYLE

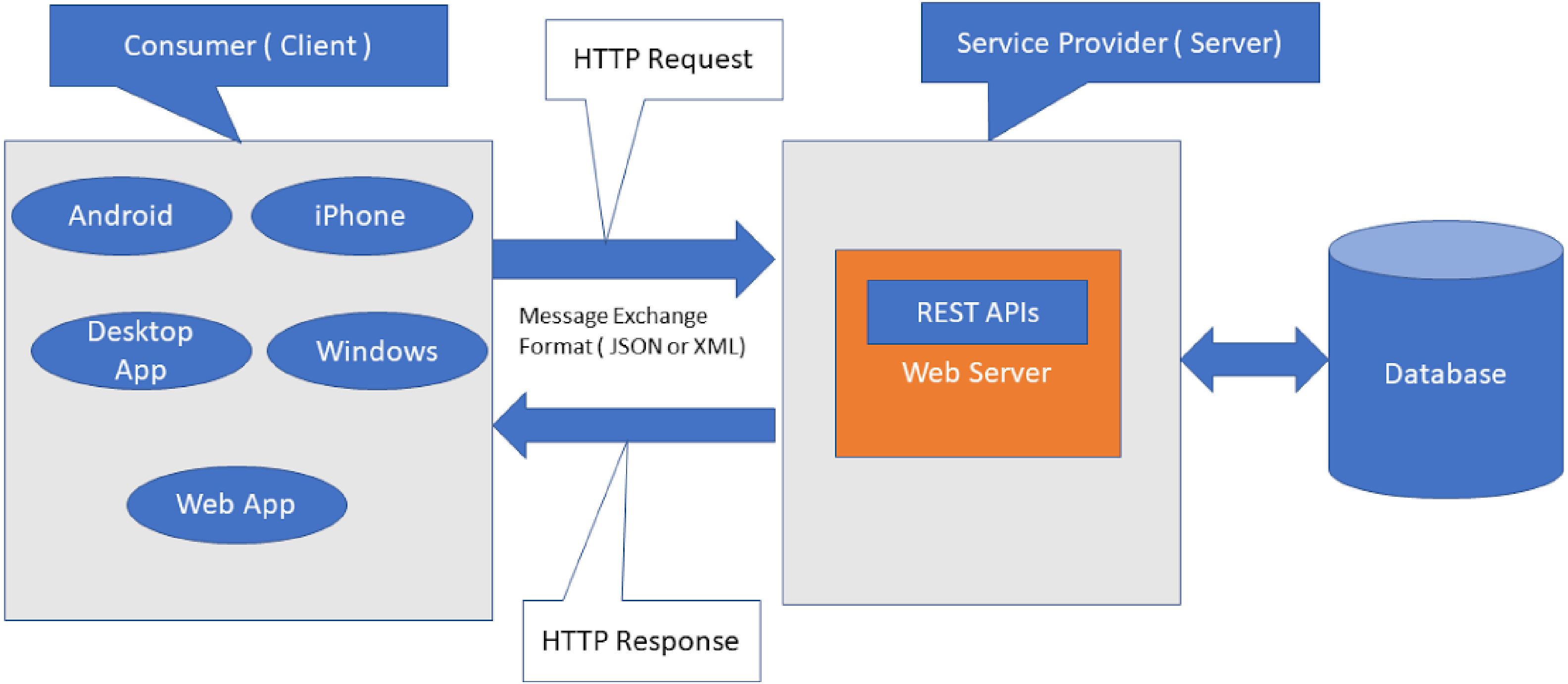
The Representational State Transfer (REST) architectural style is a worldview that elevates information into a first-class element of architectures. REST allows us to achieve the architectural properties of performance, scalability, generality, simplicity, modifiability, and extensibility.

REST ARCHITECTURE STYLE

The **REST architectural style** uses HTTP to request access and use data. This allows for interaction with RESTful web services.

Its principles were formulated in **2000** by computer scientist **ROY FIELDING** and gained popularity as a scalable and flexible alternative to older methods of machine-to-machine communication. It still remains the gold standard for public APIs.

REST – Architecture



04 HTTP METHODS

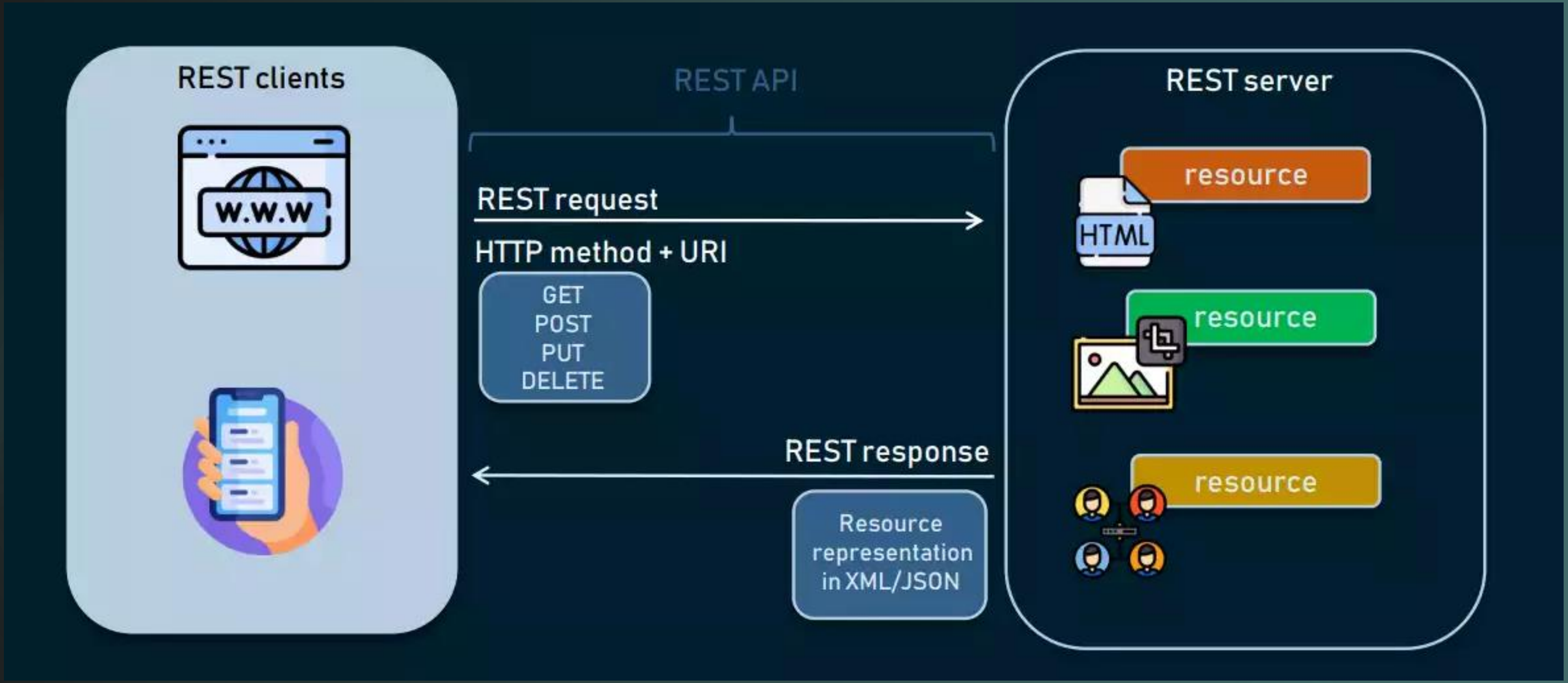
KEY ELEMENTS OF REST API

a **Client** or **Software** that runs on a user’s computer or smartphone and initiates communication.

a **Server** that offers an API as a means of access to its data or features.

a **Resource**, which is any piece of content that the server can provide to the client (for example, a video or a text file).

REST API IN ACTION



To get access to a resource, the client sends an **HTTP request.**

In return, the server generates an **HTTP response** with encoded data on the resource. Both types of REST messages are *self-descriptive,* meaning they contain information on how to interpret and process them.

REST REQUEST STRUCTURE

Any **REST** request includes four essential parts: an **HTTP Method**, an endpoint, headers, and a body.

An **HTTP Method** describes what is to be done with a *resource*. There are four basic methods also named **CRUD** operations:

REST REQUEST STRUCTURE

REST request
HTTP method
GET
POST
PUT
DELETE

POST to Create a resource,

GET to Retrieve a resource,

PUT to Update a resource, and

DELETE to Delete a resource.

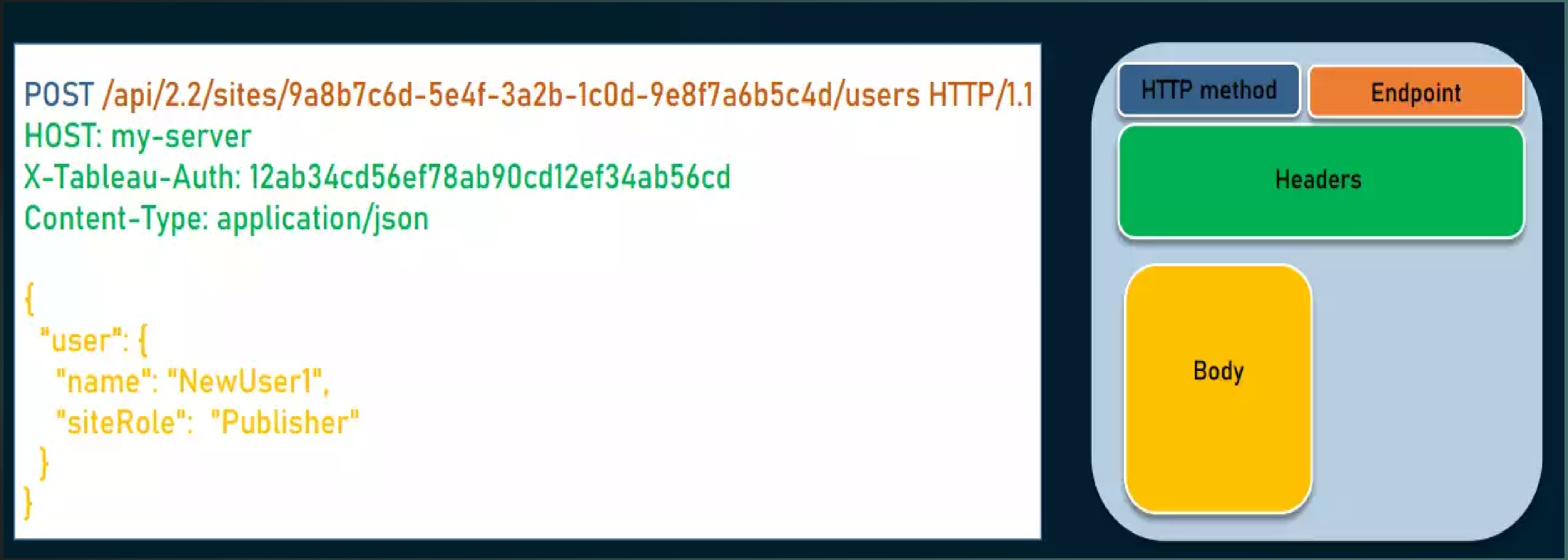
REST REQUEST STRUCTURE

An **Endpoint** contains a *Uniform Resource Identifier (URI)* indicating where and how to find the resource on the Internet. The most common type of URI is a *Unique Resource Location* (URL), serving as a complete web address.

Headers store information relevant to both the client and server. Mainly, headers provide authentication data — such as an API key, the name or IP address of the computer where the server is installed, and the information about the response format.

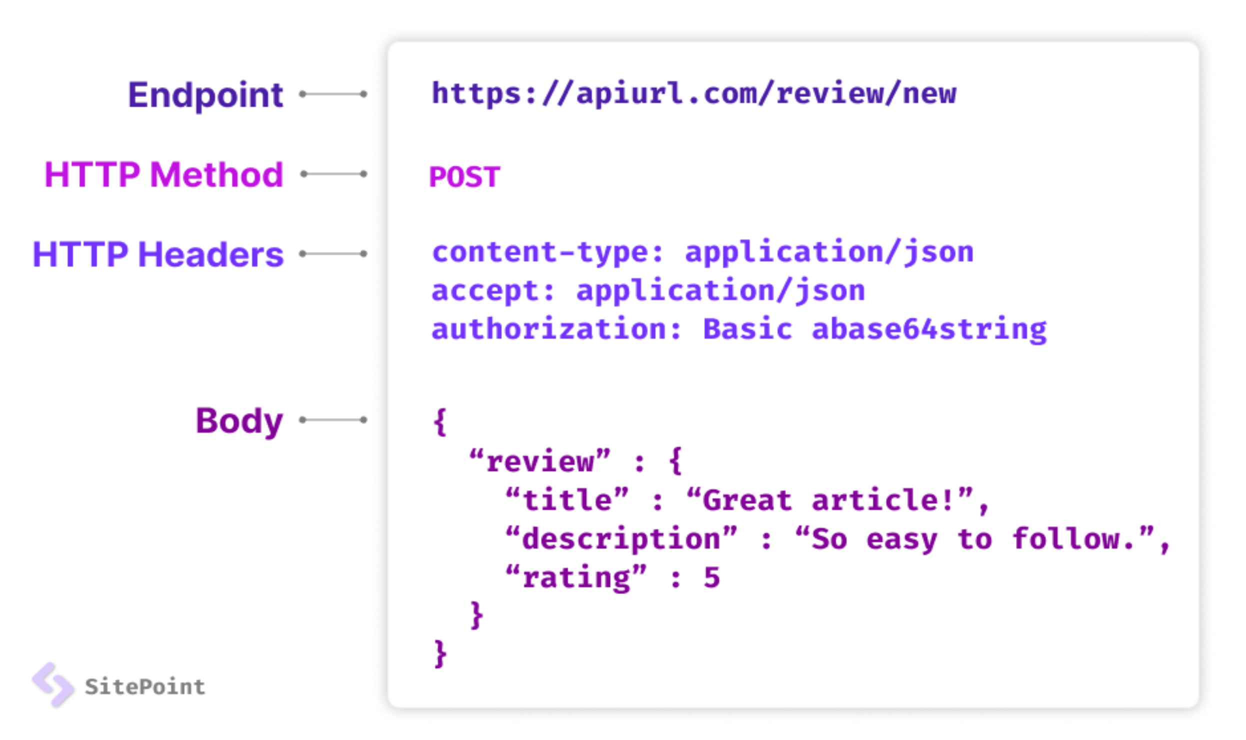
A **body** is used to convey additional information to the server. For instance, it may be a piece of data you want to add or replace.

REST REQUEST STRUCTURE



```
POST /api/2.2/sites/9a8b7c6d-5e4f-3a2b-1c0d-9e8f7a6b5c4d/users HTTP/1.1
HOST: my-server
X-Tableau-Auth: 12ab34cd56ef78ab90cd12ef34ab56cd
Content-Type: application/json

{
 "user": {
  "name": "NewUser1",
  "siteRole":  "Publisher"
 }
}
```

REST REQUEST STRUCTURE



Endpoint

```
https://apiurl.com/review/new
```

HTTP Method

```
POST
```

HTTP Headers

```
content-type: application/json
accept: application/json
authorization: Basic abase64string
```

Body

```
{
  “review” : {
    “title” : “Great article!”,
    “description” : “So easy to follow.”,
    “rating” : 5
  }
}
```

SitePoint

05

WHY WE USE REST API?

WHY WE USE REST API?

REST works on top of the HTTP transport. It takes advantage of HTTP’s native capabilities, such as **GET**, **PUT**, **POST** and **DELETE**. When a request is sent to a RESTful API, the response (the “representation” of the information “resource” being sought) returns in either the JSON, XML or HTML format. A RESTful API is defined by a web address, or Uniform Resource Identifier (URI), which typically follows a naming convention.

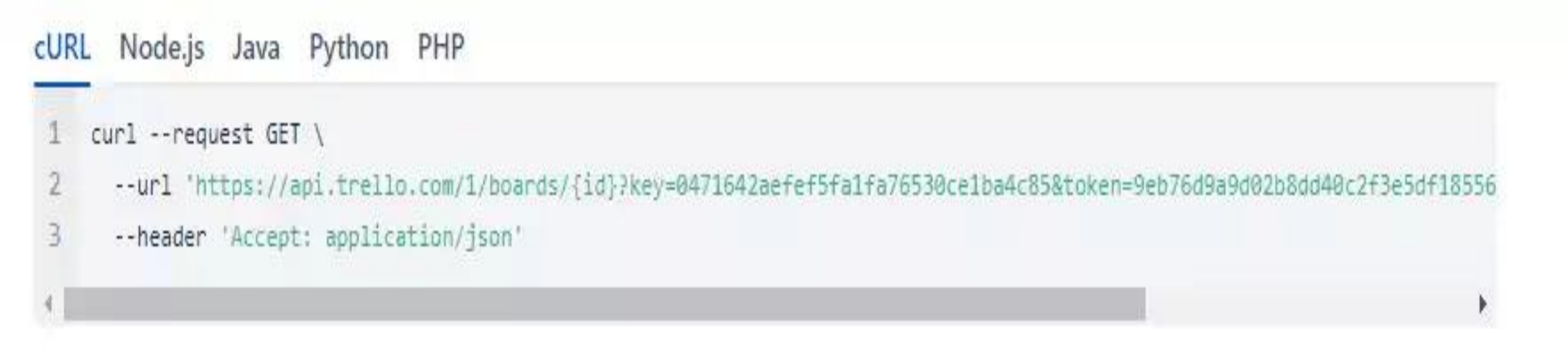
REST is easier to work with and more flexible:

- No expensive tools needed in order for interaction with web services.
- Shorter learning curve.
- More efficient (XML, used in SOAP messages, is longer than REST’s message formats).
- Faster, with less processing required.

05

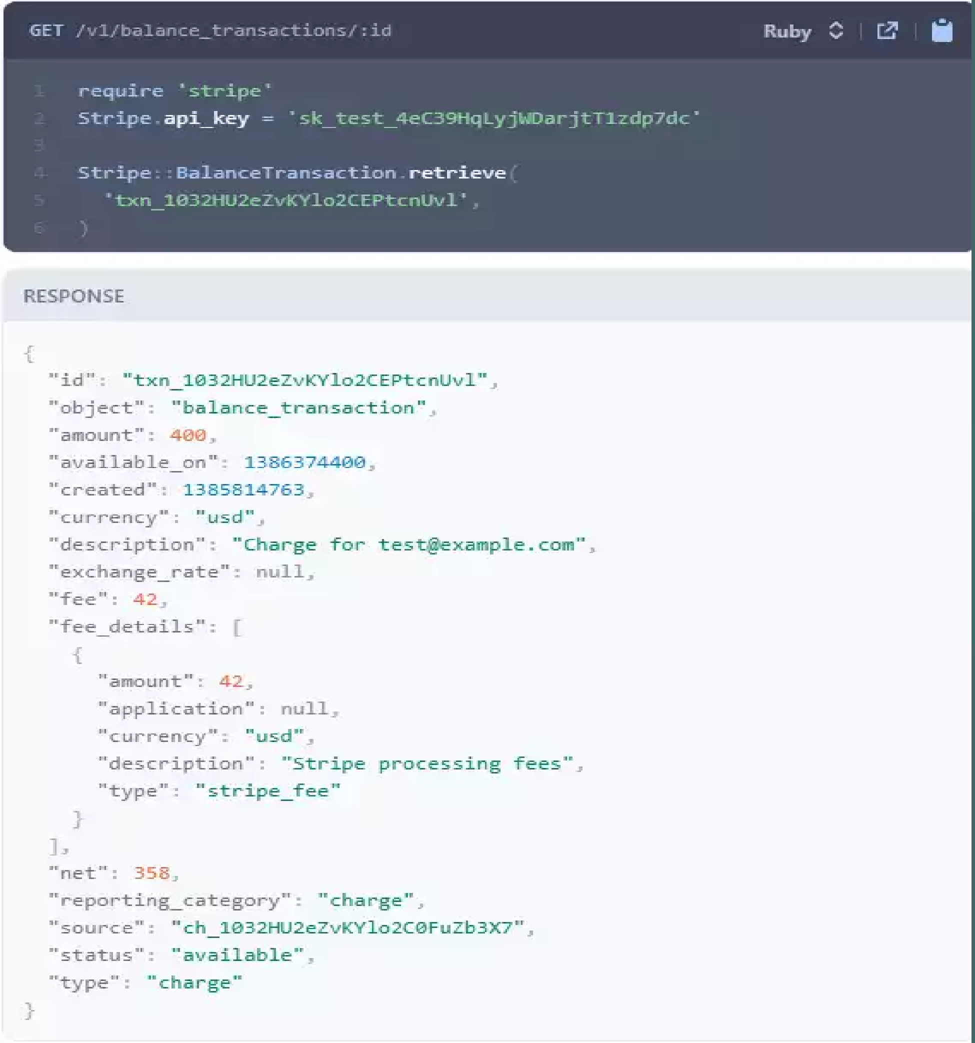
EXAMPLES OF REST API

TRELLO API



```
curl --request GET \
  --url 'https://api.trello.com/1/boards/{id}?key=0471642aefef5fa1fa76530ce1ba4c85&token=9eb76d9a9d02b8dd40c2f3e5df18556
  --header 'Accept: application/json'
```

STRIPE API



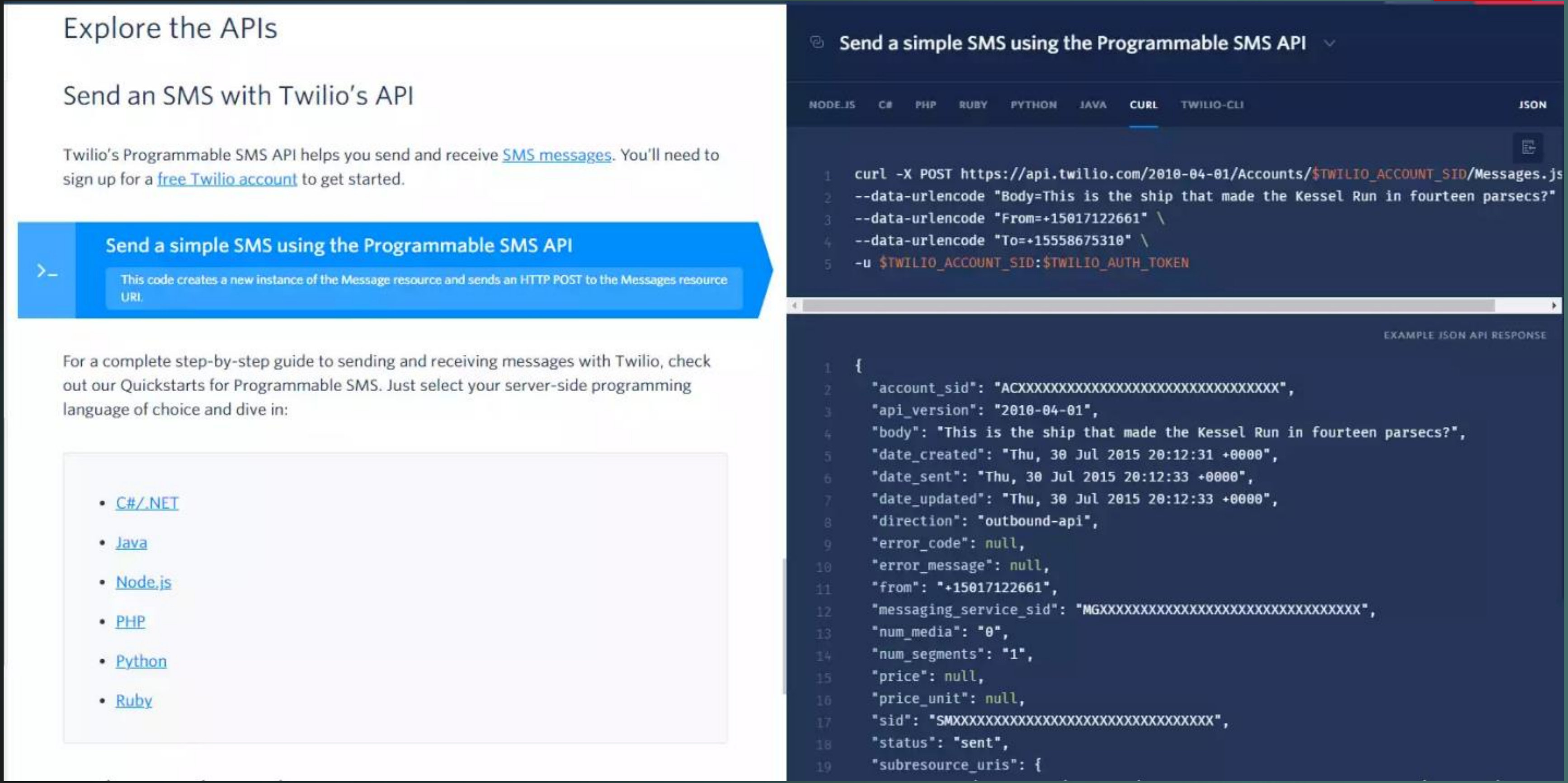
GET /v1/balance_transactions/:id

```ruby
require 'stripe'
Stripe.api_key = 'sk_test_4eC39HqLyjWDarjtT1zdp7dc'

Stripe::BalanceTransaction.retrieve(
  'txn_1032HU2eZvKYlo2CEPtcnUvl',
)
```

RESPONSE

```json
{
  "id": "txn_1032HU2eZvKYlo2CEPtcnUvl",
  "object": "balance_transaction",
  "amount": 400,
  "available_on": 1386374400,
  "created": 1385814763,
  "currency": "usd",
  "description": "Charge for test@example.com",
  "exchange_rate": null,
  "fee": 42,
  "fee_details": [
    {
      "amount": 42,
      "application": null,
      "currency": "usd",
      "description": "Stripe processing fees",
      "type": "stripe_fee"
    }
  ],
  "net": 358,
  "reporting_category": "charge",
  "source": "ch_1032HU2eZvKYlo2C0FuZb3X7",
  "status": "available",
  "type": "charge"
}
```

TWILIO API



Explore the APIs

Send an SMS with Twilio's API

Twilio's Programmable SMS API helps you send and receive SMS messages. You'll need to sign up for a free Twilio account to get started.

>_ **Send a simple SMS using the Programmable SMS API**

This code creates a new instance of the Message resource and sends an HTTP POST to the Messages resource URI.

For a complete step-by-step guide to sending and receiving messages with Twilio, check out our Quickstarts for Programmable SMS. Just select your server-side programming language of choice and dive in:

- C#/.NET
- Java
- Node.js
- PHP
- Python
- Ruby

Send a simple SMS using the Programmable SMS API

```
curl -X POST https://api.twilio.com/2010-04-01/Accounts/$TWILIO_ACCOUNT_SID/Messages.js
--data-urlencode "Body=This is the ship that made the Kessel Run in fourteen parsecs?"
--data-urlencode "From=+15017122661" \
--data-urlencode "To=+15558675310" \
-u $TWILIO_ACCOUNT_SID:$TWILIO_AUTH_TOKEN
```

EXAMPLE JSON API RESPONSE

```
{
  "account_sid": "ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "api_version": "2010-04-01",
  "body": "This is the ship that made the Kessel Run in fourteen parsecs?",
  "date_created": "Thu, 30 Jul 2015 20:12:31 +0000",
  "date_sent": "Thu, 30 Jul 2015 20:12:33 +0000",
  "date_updated": "Thu, 30 Jul 2015 20:12:33 +0000",
  "direction": "outbound-api",
  "error_code": null,
  "error_message": null,
  "from": "+15017122661",
  "messaging_service_sid": "MGXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "num_media": "0",
  "num_segments": "1",
  "price": null,
  "price_unit": null,
  "sid": "SMXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "status": "sent",
  "subresource_uris": {
```

MODEL VS. INTERFACE

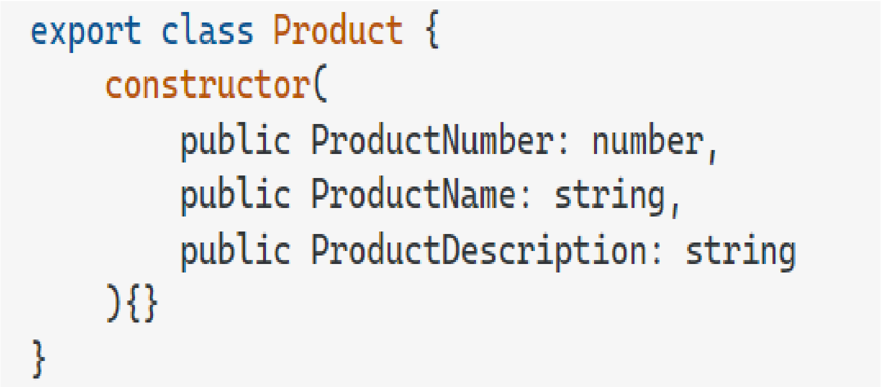
MODEL

- Blueprints for creating new objects
- Properties and methods may be public, private, or static.
- Include a constructor.
- Can be use during run time.

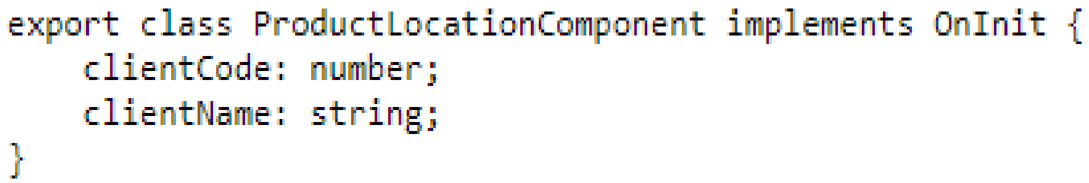
FUNCTION OF MODEL

- Instantiate using a new keyword
- To have a constructor that sets up default variables or initialization.
- When you want to associate behaviors with data more closely.
- You enforce constraints on the creation of your instances.

Example:



```
export class Product {
    constructor(
        public ProductNumber: number,
        public ProductName: string,
        public ProductDescription: string
    ){}
}
```



```
export class ProductLocationComponent implements OnInit {
    clientCode: number;
    clientName: string;
}
```

INTERFACE

- In Typescript, the interface is also known as "duck typing" or "structural subtyping.“

- It specifies the properties and function of an object along with its name and type.

INTERFACE

- used only by the TypeScript compiler for type verification.

- Strongly typed data checking at compile time is accomplished by Typescript using an Interface.

FUNCTION OF MODEL

- when you need to construct a shape of associated variables and methods that characterize an object.

- Using classes, establish some fundamental rules for your properties and methods.

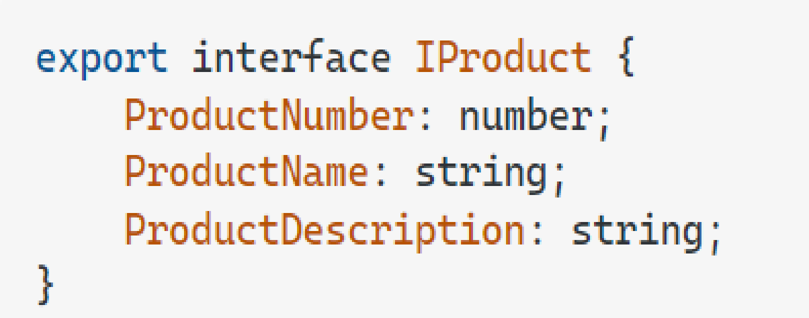
FUNCTION OF MODEL

- Data must be communicated; no actions or logic are necessary (constructor initialization, methods).

- Use the implements keyword to make a class implement an Interface.

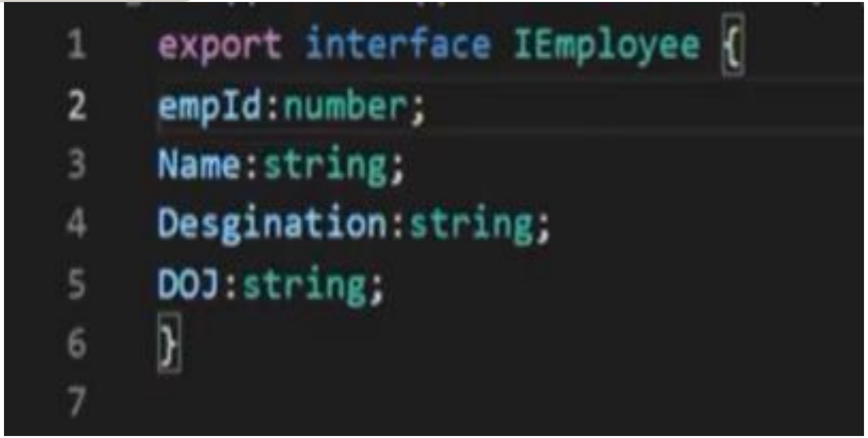
FUNCTION OF MODEL

- When you don't want to add extra overhead to the output but just need the specification for the server data.



```
export interface IProduct {
    ProductNumber: number;
    ProductName: string;
    ProductDescription: string;
}
```

Example:



```
export interface IEmployee {
empId:number;
Name:string;
Desgination:string;
DOJ:string;
}
```

DIFFERENCE BETWEEN MODEL AND INTERFACE

- An interface defines the properties and methods that are required by an object, while the model class defines how those properties will be instantiated and used.

- Model classes are more abstract than interfaces because they allow for the implementation of more complex logic than just being able to describe an object.

DIFFERENCE BETWEEN MODEL AND INTERFACE

- When code is being compiled, a class cannot disappear, but an interface can.

- To generate an object, a class can be instantiated. It is not possible to instantiate an interface.

DIFFERENCE BETWEEN MODEL AND INTERFACE

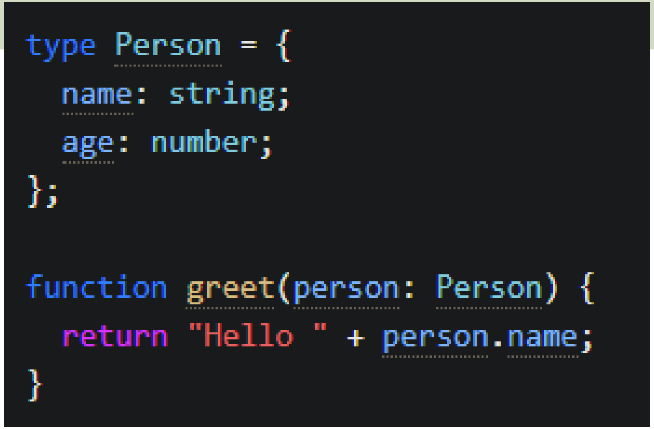
- A class's methods are used to carry out a certain function. Although an interface's methods are simply abstract (the only declaration, not have a body).

- A class may have public, protected, or private members. An interface's members are always visible to the public.

Example:

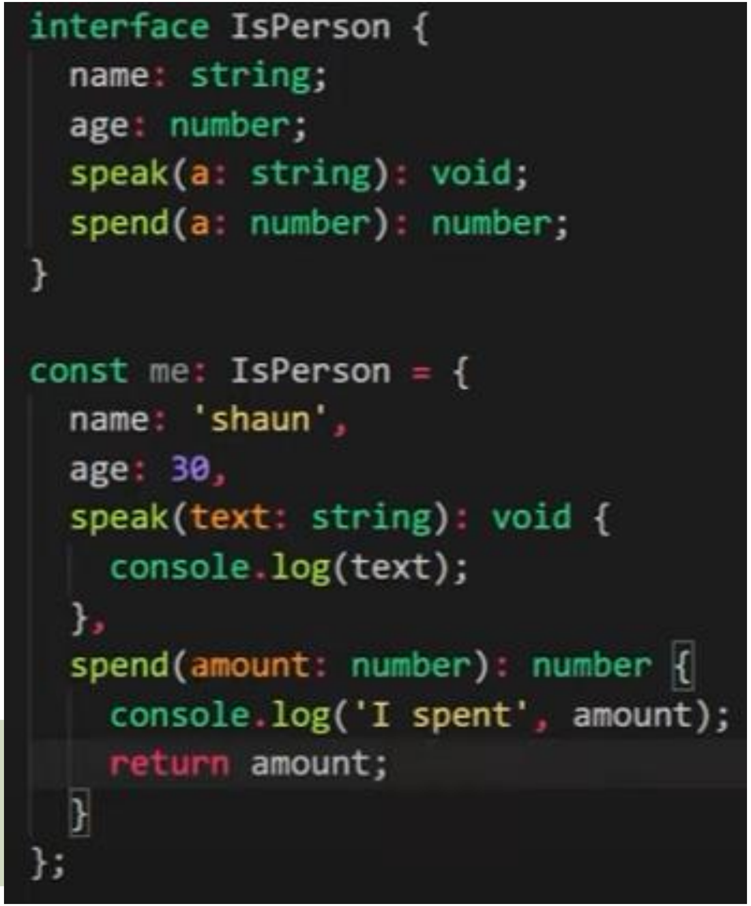
INTERFACE

Interface with properties



```
type Person = {
  name: string;
  age: number;
};

function greet(person: Person) {
  return "Hello " + person.name;
}
```

Interface with properties and methods

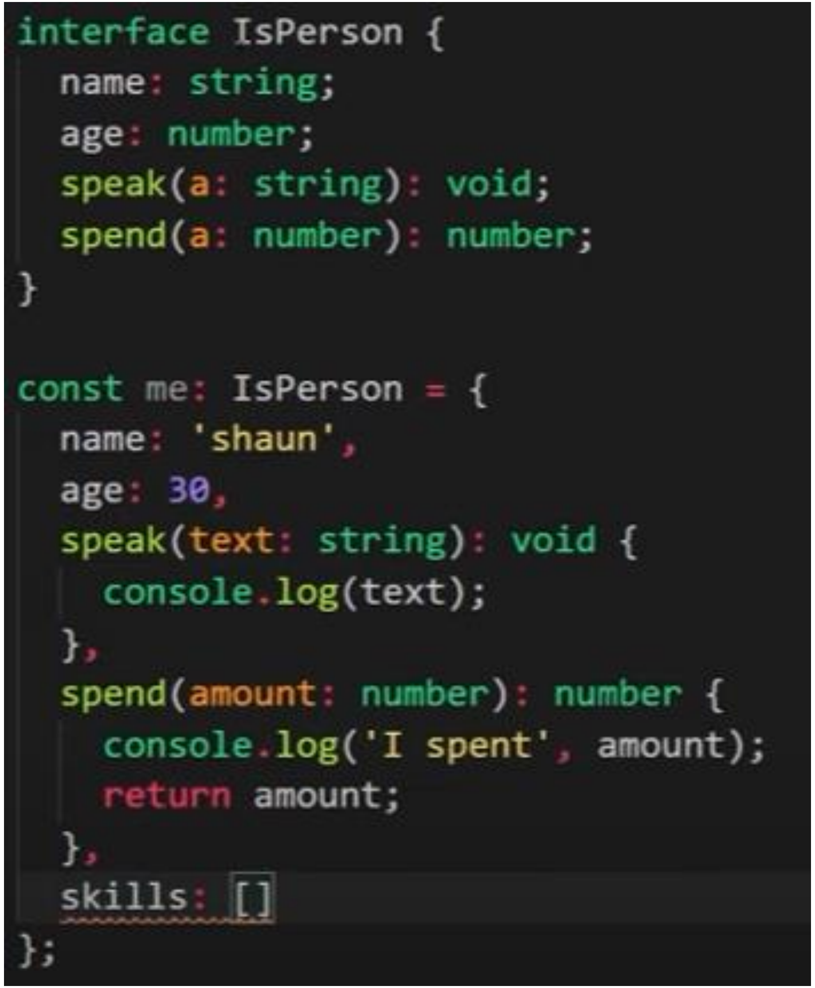


```
interface IsPerson {
  name: string;
  age: number;
  speak(a: string): void;
  spend(a: number): number;
}

const me: IsPerson = {
  name: 'shaun',
  age: 30,
  speak(text: string): void {
    console.log(text);
  },
  spend(amount: number): number {
    console.log('I spent', amount);
    return amount;
  }
};
```

Example:

INTERFACE

An object with mismatched shape

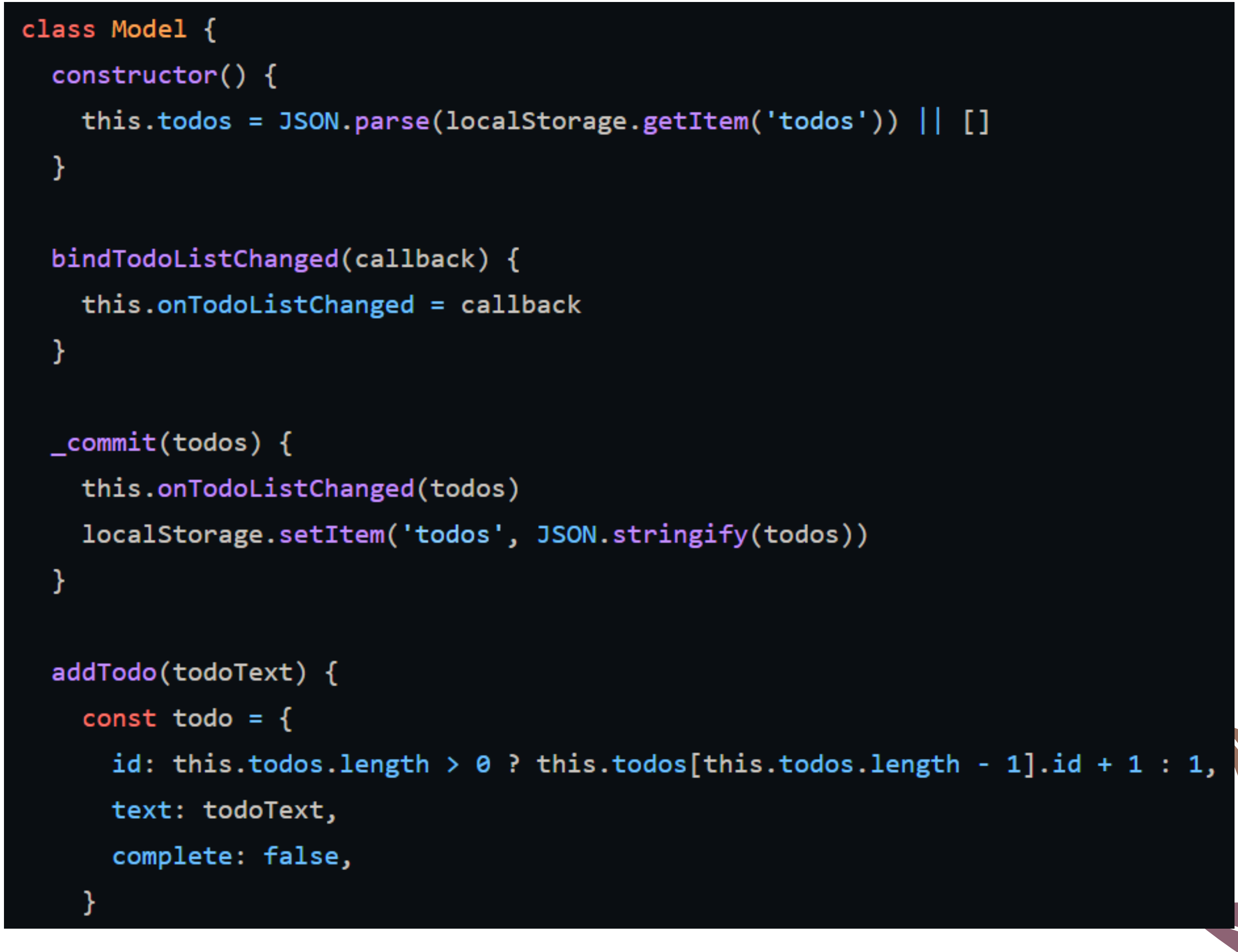


```
interface IsPerson {
  name: string;
  age: number;
  speak(a: string): void;
  spend(a: number): number;
}

const me: IsPerson = {
  name: 'shaun',
  age: 30,
  speak(text: string): void {
    console.log(text);
  },
  spend(amount: number): number {
    console.log('I spent', amount);
    return amount;
  },
  skills: []
};
```

Example:

Model (MVC)

A model class

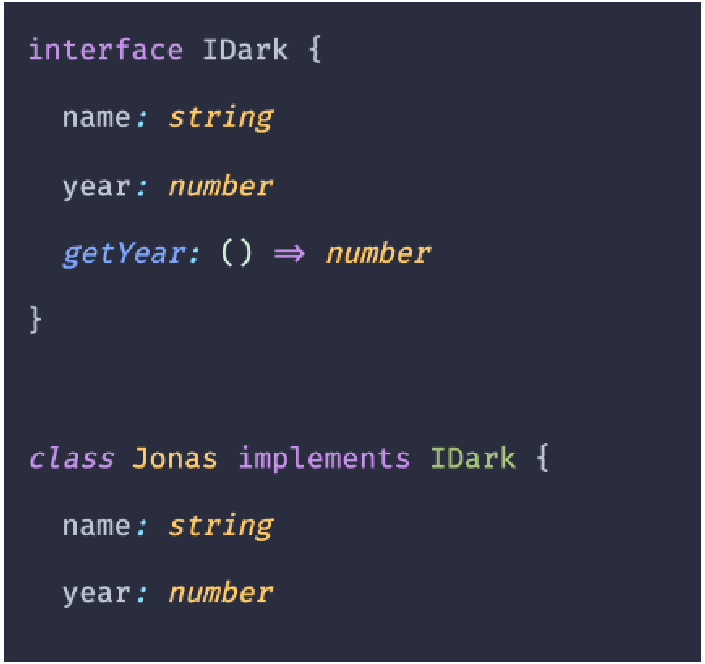


```
class Model {
  constructor() {
    this.todos = JSON.parse(localStorage.getItem('todos')) || []
  }

  bindTodoListChanged(callback) {
    this.onTodoListChanged = callback
  }

  _commit(todos) {
    this.onTodoListChanged(todos)
    localStorage.setItem('todos', JSON.stringify(todos))
  }

  addTodo(todoText) {
    const todo = {
      id: this.todos.length > 0 ? this.todos[this.todos.length - 1].id + 1 : 1,
      text: todoText,
      complete: false,
    }
```

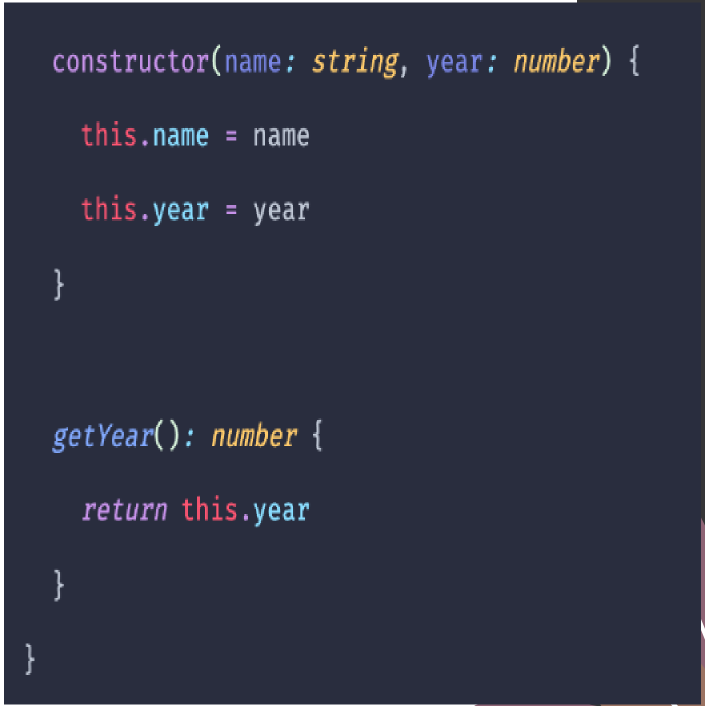
Example:

Model (MVC)

A class implementing an interface



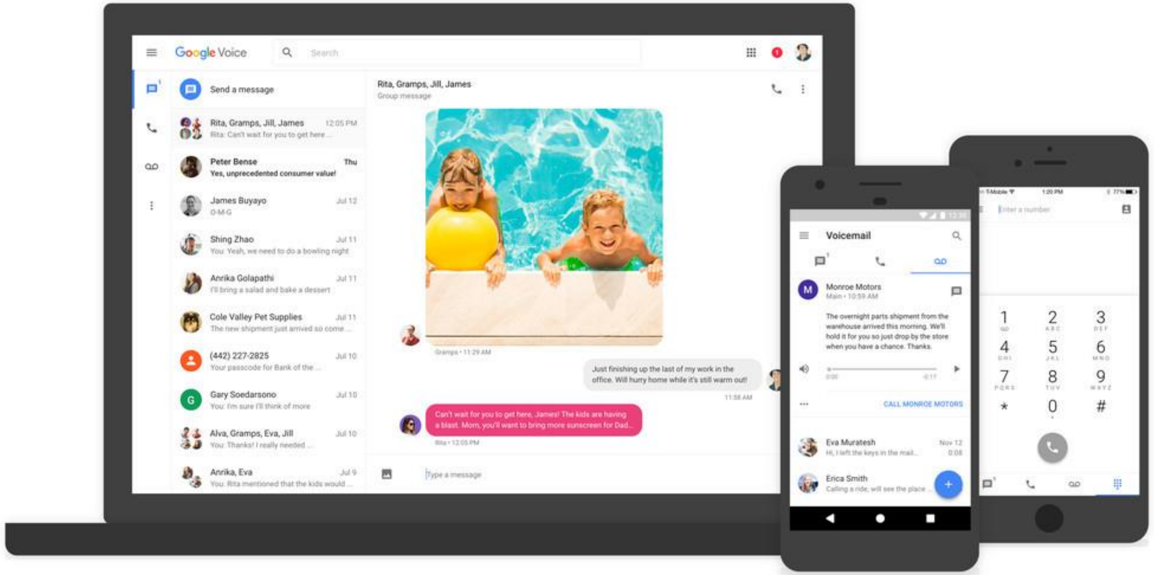
```
interface IDark {
  name: string
  year: number
  getYear: () => number
}

class Jonas implements IDark {
  name: string
  year: number
```



```
  constructor(name: string, year: number) {
    this.name = name
    this.year = year
  }

  getYear(): number {
    return this.year
  }
}
```

Google

Firebase



Rita, Gramps, Jill, James
Group message

Rita, Gramps, Jill, James 12:05 PM
Rita: Can't wait for you to get here ...

Peter Bense Thu
Yes, unprecedented consumer value!

James Buyayo Jul 12
O-M-G

Shing Zhao Jul 11
You: Yeah, we need to do a bowling night

Anrika Golapathi Jul 11
I'll bring a salad and bake a dessert

Cole Valley Pet Supplies Jul 11
The new shipment just arrived so come ...

(442) 227-2825 Jul 10
Your passcode for Bank of the ...

Gary Soedarsono Jul 10
You: I'm sure I'll think of more

Alva, Gramps, Eva, Jill Jul 10
You: Thanks! I really needed ...

Anrika, Eva Jul 9
You: Rita mentioned that the kids would ...

Gramps • 11:29 AM

Just finishing up the last of my work in the office. Will hurry home while it's still warm out!
11:58 AM

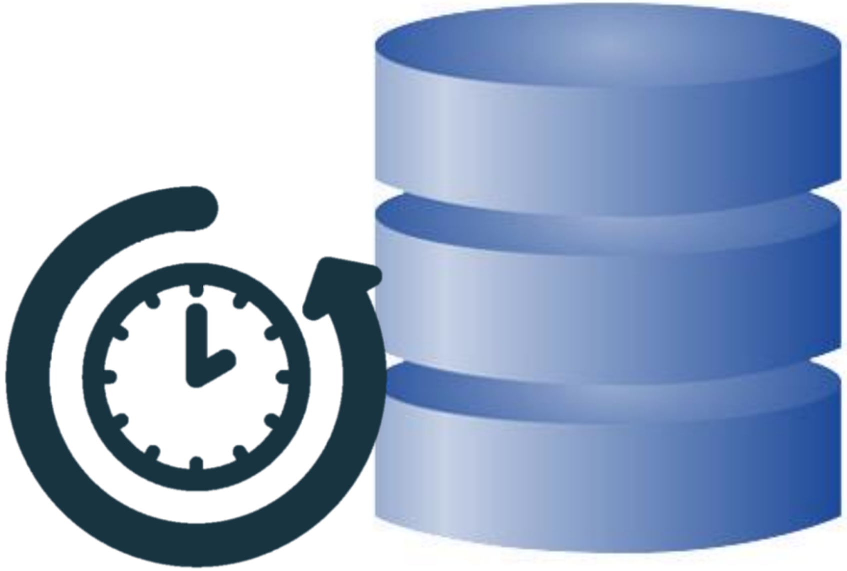
Can't wait for you to get here, James! The kids are having a blast. Mom, you'll want to bring more sunscreen for Dad...
Rita • 12:05 PM

Monroe Motors
Main • 10:59 AM
The overnight parts shipment from the warehouse arrived this morning. We'll hold it for you so just drop by the store when you have a chance. Thanks.

Eva Muratesh Nov 12
Hi, I left the keys in the mail... 0:08

Erica Smith
Calling a ride, will see the place ...

noSQL



Key Differences between SQL and NoSQL

SQL	NoSQL
Relational Database Management System	Distributed Database Management System
Structured Query Language	Document-Oriented Json Tree
Fixed/Predefined Schema	Dynamic Schema
Complex Queries	Hierarchical Data Storage

Merits and Demerits of NoSQL

Why?

Lesser Investment

Rapid Development Cycle

Faster than SQL

Cloud Technology

Why Not?

Still Evolving

Multiple Databases

Data Duplication

Step 1: Application Workflow and Query Patterns

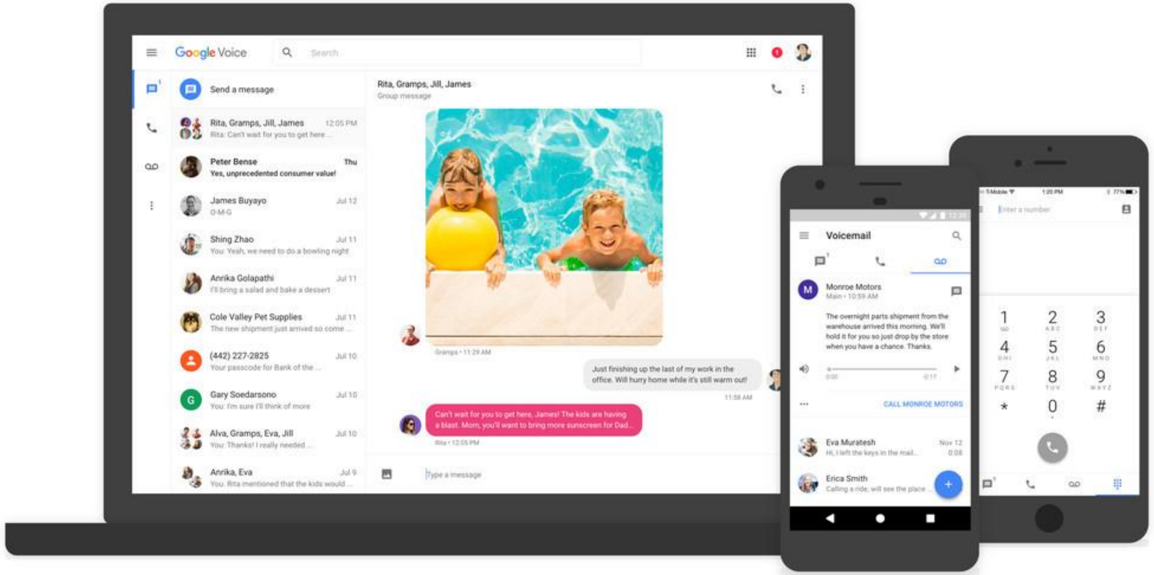
- Business Data Entities
- User Stories
- Query Pattern

How to design schema in NoSQL Database?

QUERY DRIVEN DESIGN

- Design Containers
- Denormalize Data
- Design Primary Key
- Design Indexes

Step 2: Designing a Schema



Rita, Gramps, Jill, James 12:05 PM
Rita: Can't wait for you to get here ...

Peter Bense Thu
Yes, unprecedented consumer value!

James Buyayo Jul 12
O-M-G

Shing Zhao Jul 11
You: Yeah, we need to do a bowling night

Anrika Golapathi Jul 11
I'll bring a salad and bake a dessert

Cole Valley Pet Supplies Jul 11
The new shipment just arrived so come ...

(442) 227-2825 Jul 10
Your passcode for Bank of the ...

Gary Soedarsono Jul 10
You: I'm sure I'll think of more

Alva, Gramps, Eva, Jill Jul 10
You: Thanks! I really needed ...

Anrika, Eva Jul 9
You: Rita mentioned that the kids would ...

Rita, Gramps, Jill, James
Group message

Gramps • 11:29 AM

Just finishing up the last of my work in the office. Will hurry home while it's still warm out!
11:58 AM

Can't wait for you to get here, James! The kids are having a blast. Mom, you'll want to bring more sunscreen for Dad...
Rita • 12:05 PM

Monroe Motors
Main • 10:59 AM
The overnight parts shipment from the warehouse arrived this morning. We'll hold it for you so just drop by the store when you have a chance. Thanks.
0:00 -0:17

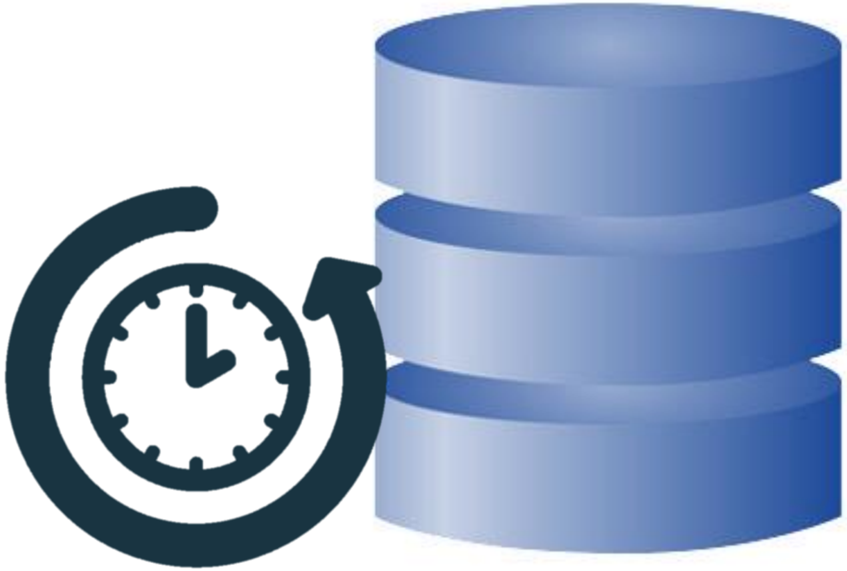
Eva Muratesh Nov 12
Hi, I left the keys in the mail... 0:08

Erica Smith
Calling a ride, will see the place ...

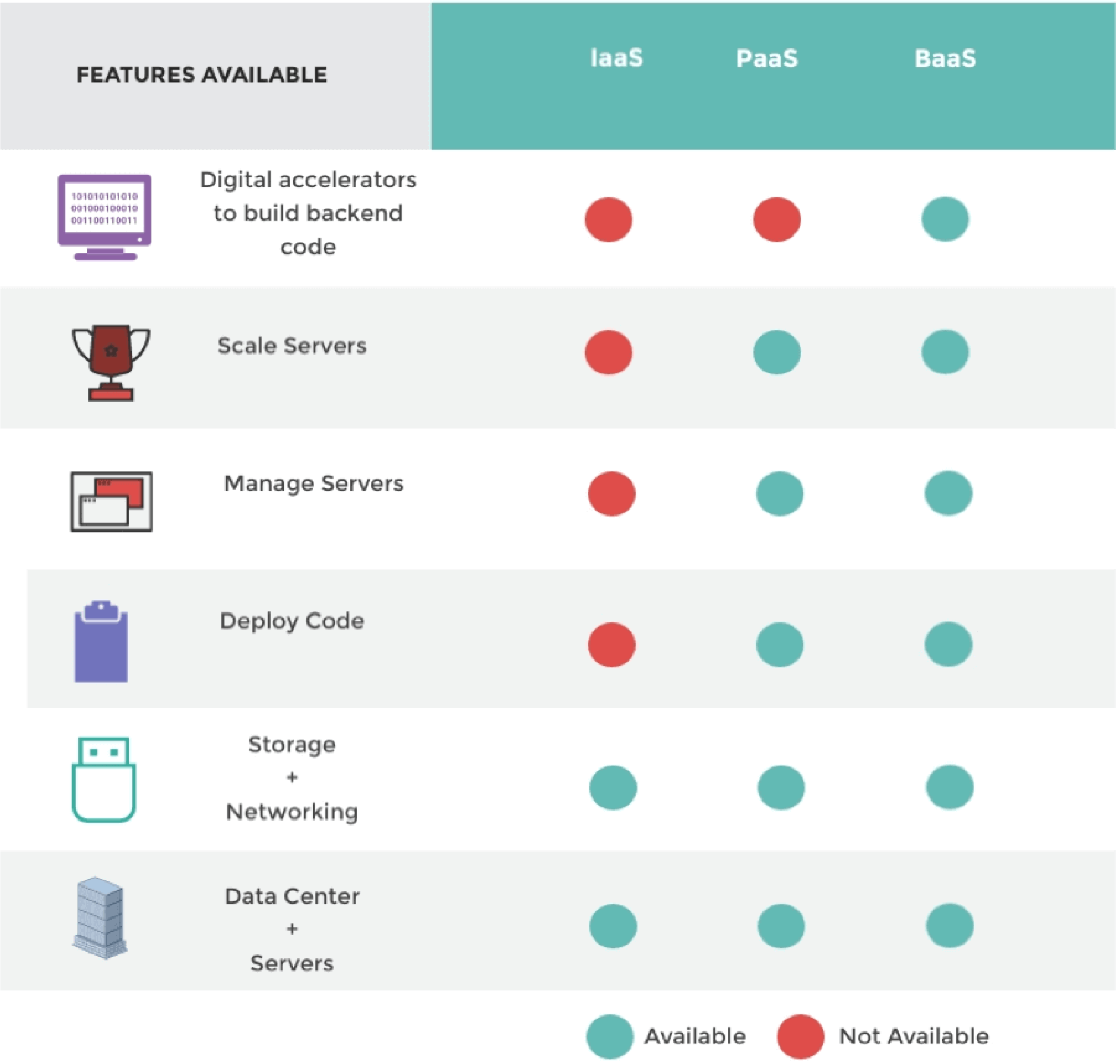
noSQL

BaaS

Backend as a Service

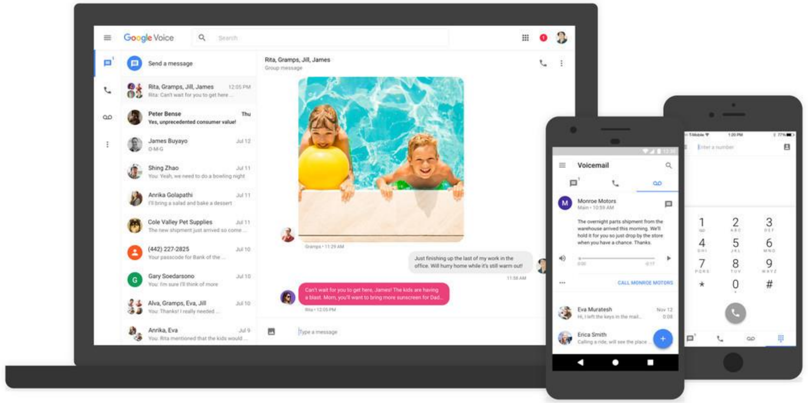


Differences between Cloud Service Providers



FEATURES AVAILABLE	IaaS	PaaS	BaaS
Digital accelerators to build backend code	🔴	🔴	🔵
Scale Servers	🔴	🔵	🔵
Manage Servers	🔴	🔵	🔵
Deploy Code	🔴	🔵	🔵
Storage + Networking	🔵	🔵	🔵
Data Center + Servers	🔵	🔵	🔵

🔵 Available 🔴 Not Available


Google Voice
Voicemail

DATA
AI
Firebase

Technical Advantage of BaaS

- Frontend Development
- Boilerplate Code
- Standard Code Environment
- Features and Settings
- Clone Apps for Testing

When and Why to used Backend as a Service

- ✓ Stand Alone Application
- ✓ Minimum Viable Product
- ✓ Uncritical Enterprise Application

A solution for:

- ▪ Manage and Scale Cloud Infrastructure
- ▪ Speed-up Backend Development

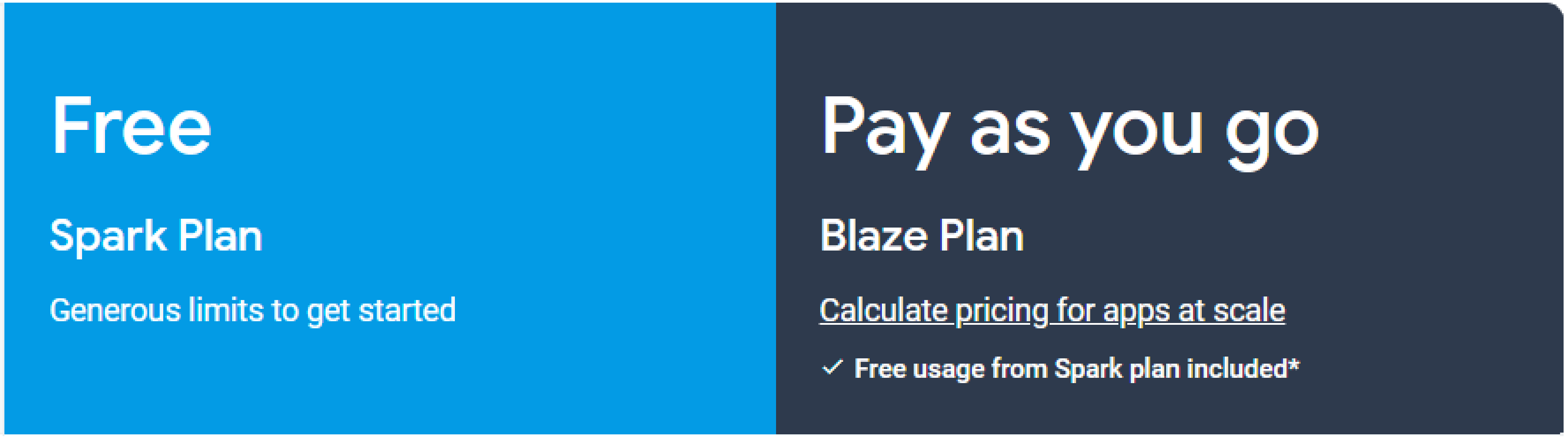
The Key Features of Google Firebase

- Authentication
- Realtime Database
- Cloud Storage
- Notification
- Community
- Synchronize
- Hosting
- Analytics

- Multiple Clients
- Performance
- Minimal Integration with 3rd-party Services
- Dynamic Database
- Time Constraint

- Cache
- Data Authority Issue
- Limited Query
- Migration Problem

✕ FLAME PLAN



Free

Spark Plan

Generous limits to get started

Pay as you go

Blaze Plan

Calculate pricing for apps at scale

✓ Free usage from Spark plan included*

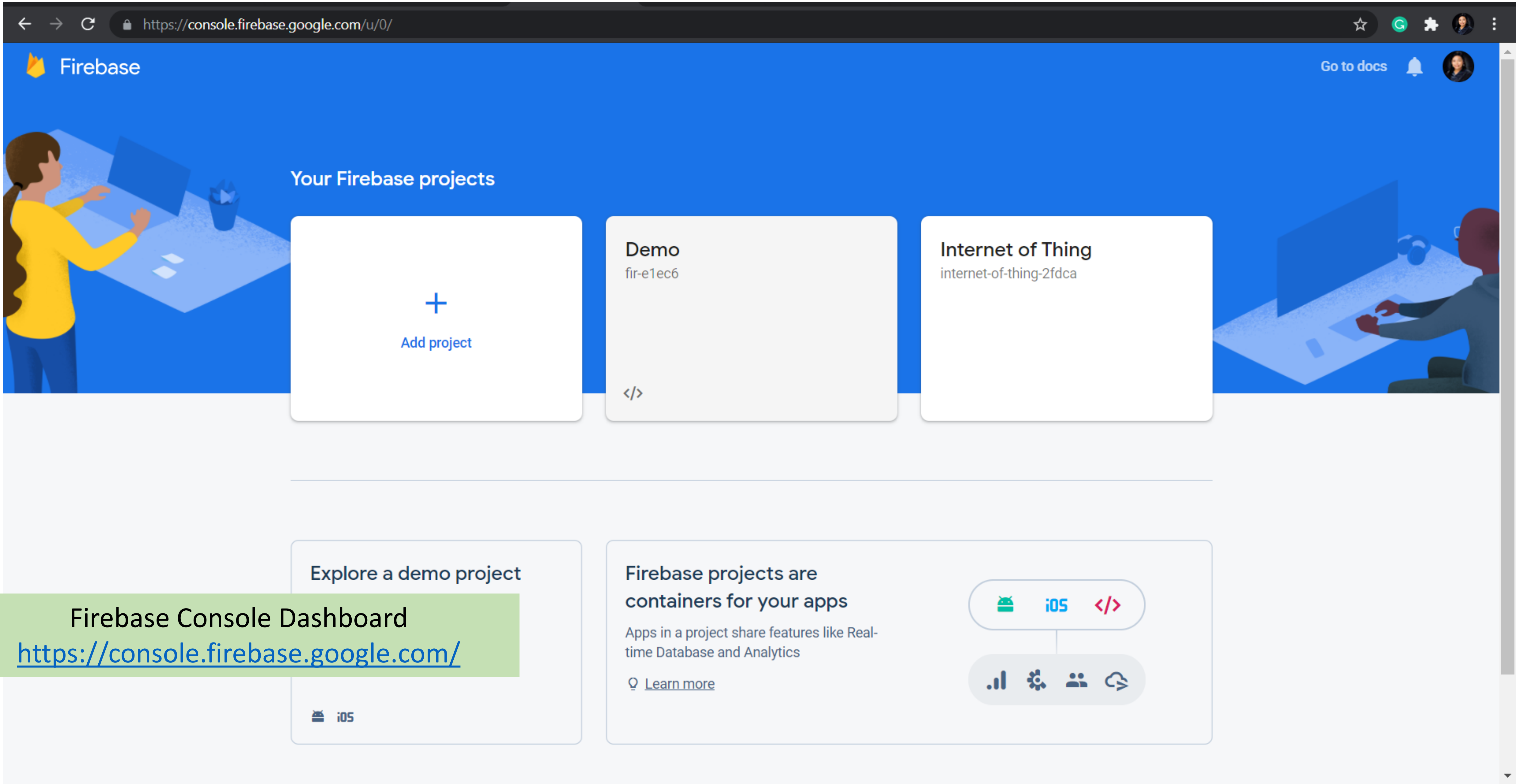
https://firebase.google.com/pricing

- ✓ JSON Tree
- ✓ Basic
- ✓ Stable
- ✓ Required Sharding
- ✓ Separate Validation
- ✓ Bandwidth and Storage

Realtime Database

Cloud Firestore

- ✓ Documents in Collection
- ✓ Indexes Queries
- ✓ Atomic
- ✓ Charges include operation performance



Your Firebase projects

Demo
fir-e1ec6

Internet of Thing
internet-of-thing-2fdca

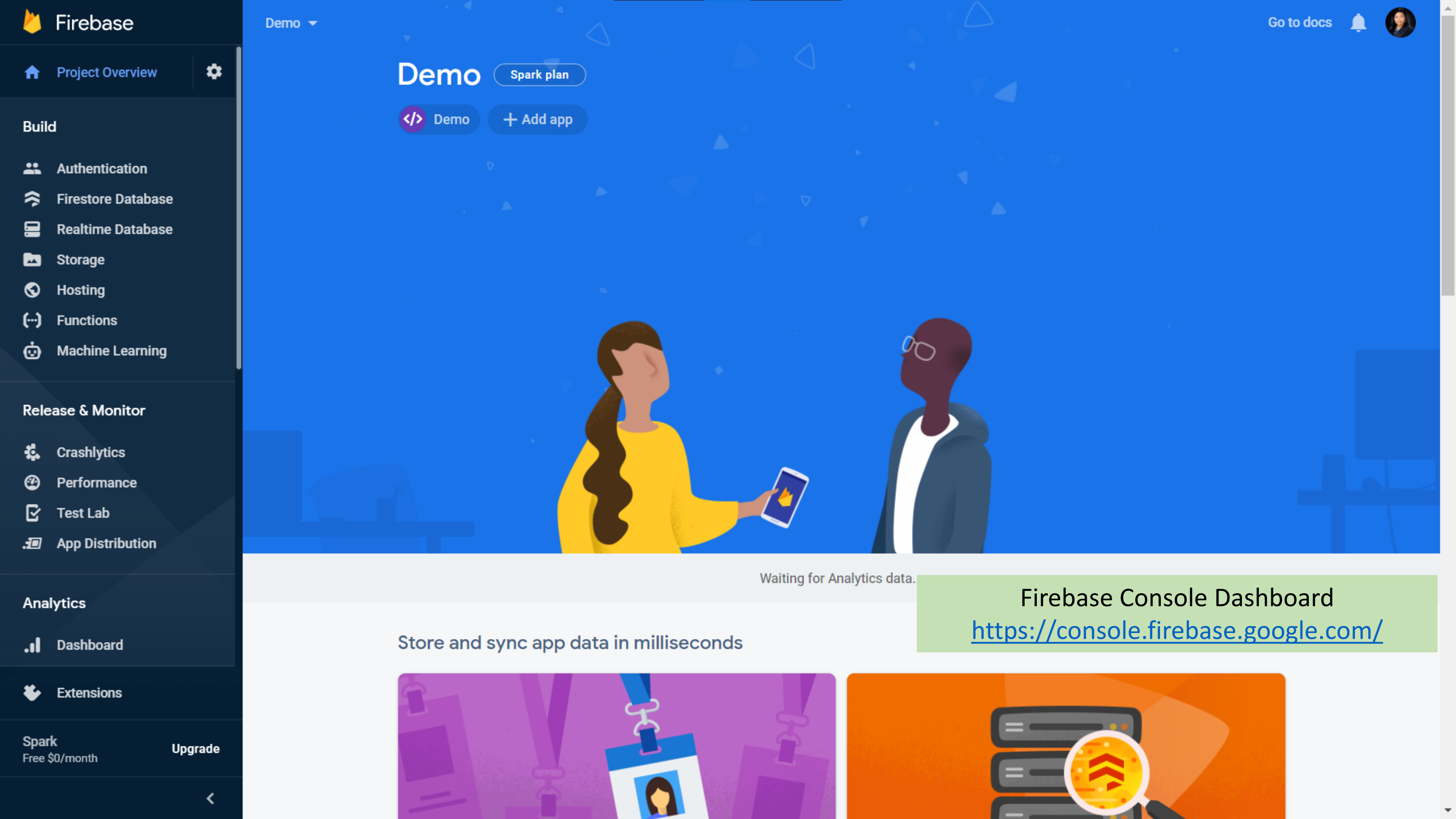
Explore a demo project

Firebase projects are containers for your apps

Apps in a project share features like Real-time Database and Analytics

Firebase Console Dashboard

https://console.firebase.google.com/

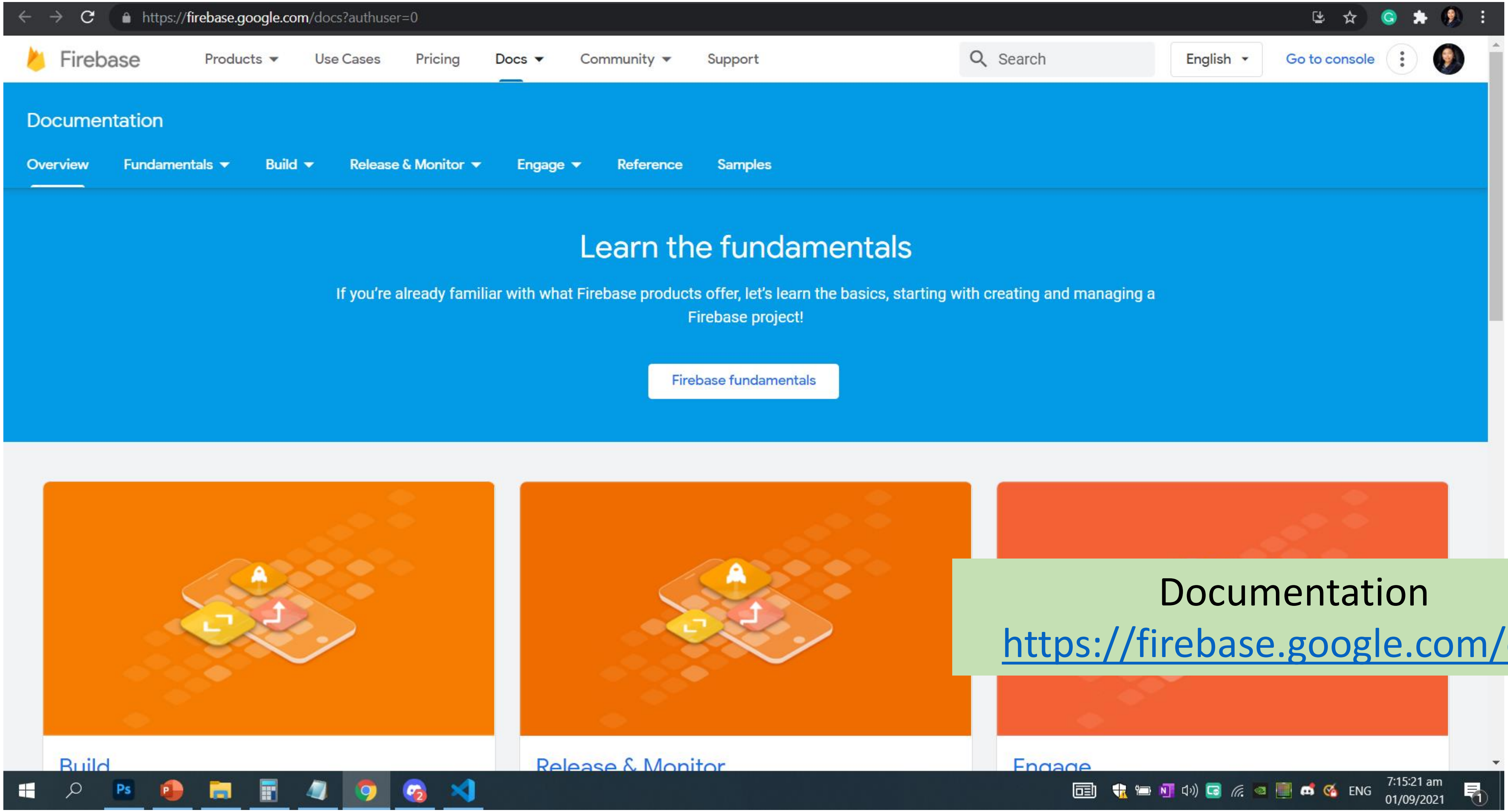


Waiting for Analytics data.

Store and sync app data in milliseconds

Firebase Console Dashboard

https://console.firebase.google.com/



Documentation

Overview | Fundamentals | Build | Release & Monitor | Engage | Reference | Samples

Learn the fundamentals

If you're already familiar with what Firebase products offer, let's learn the basics, starting with creating and managing a Firebase project!

Firebase fundamentals

Documentation

https://firebase.google.com/docs

TOPIC COVERAGE

1. Promise
2. Async & Await
3. Try .. Catch

PROMISE

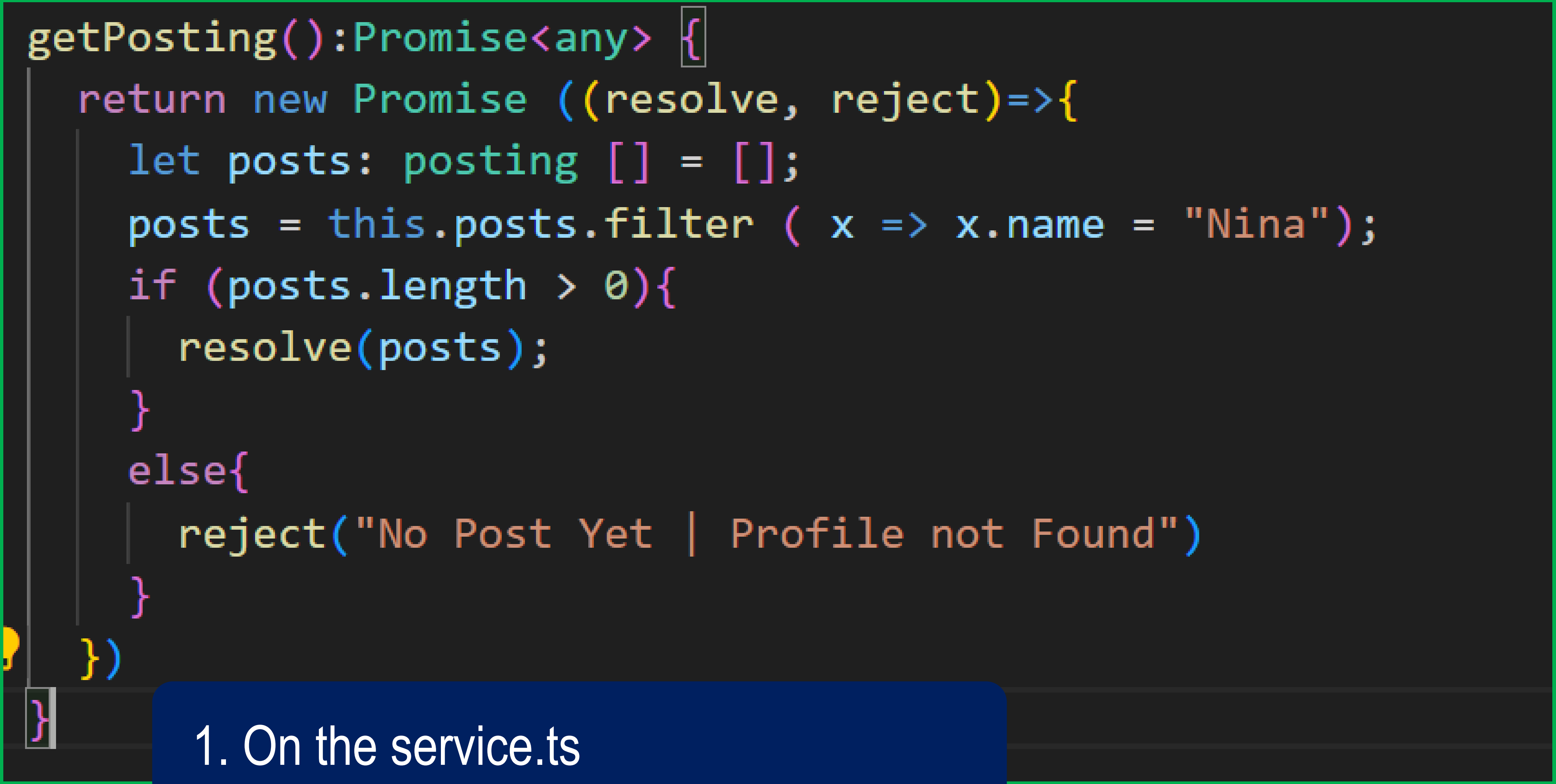
- You can create custom services that return promises, which can be used in controllers or other services for better code organization and error management.

- Promises allow for chaining `.then()` and `.catch()` methods for improved readability and error handling, and they work well with the `async`/`await` syntax for more synchronous-looking code.

PROMISE

- handle asynchronous operations, providing a mechanism to execute code once an operation completes successfully or fails.
- Promises are commonly used with the `$http` service for making HTTP requests, allowing you to handle responses and errors cleanly.

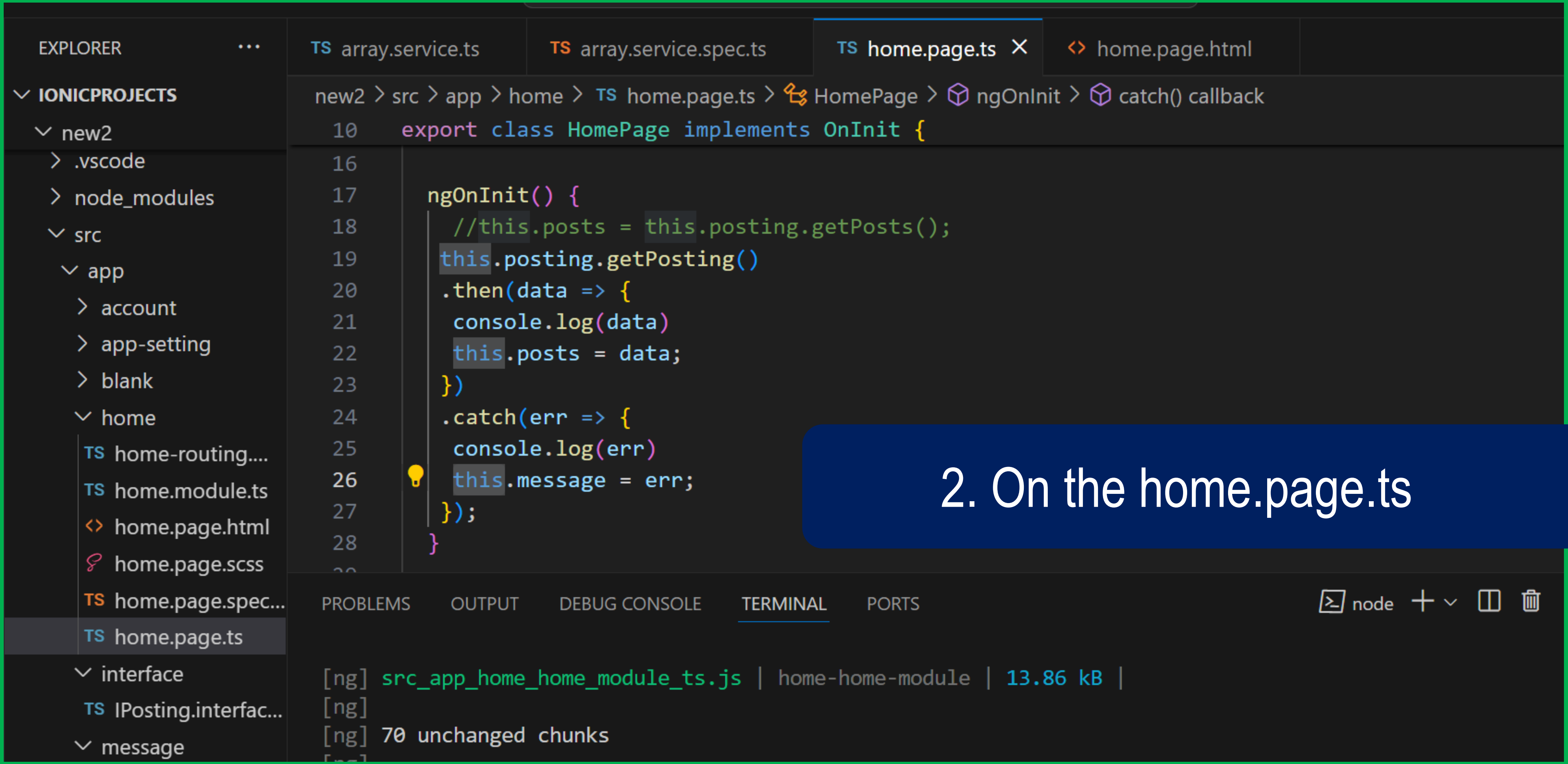
PROMISE



```
getPosting():Promise<any> {
  return new Promise ((resolve, reject)=>{
    let posts: posting [] = [];
    posts = this.posts.filter ( x => x.name = "Nina");
    if (posts.length > 0){
      resolve(posts);
    }
    else{
      reject("No Post Yet | Profile not Found")
    }
  })
}
```

1. On the service.ts

PROMISE



```
export class HomePage implements OnInit {

  ngOnInit() {
    //this.posts = this.posting.getPosts();
   this.posting.getPosting()
   .then(data => {
    console.log(data)
    this.posts = data;
   })
   .catch(err => {
    console.log(err)
    this.message = err;
   });
  }
```

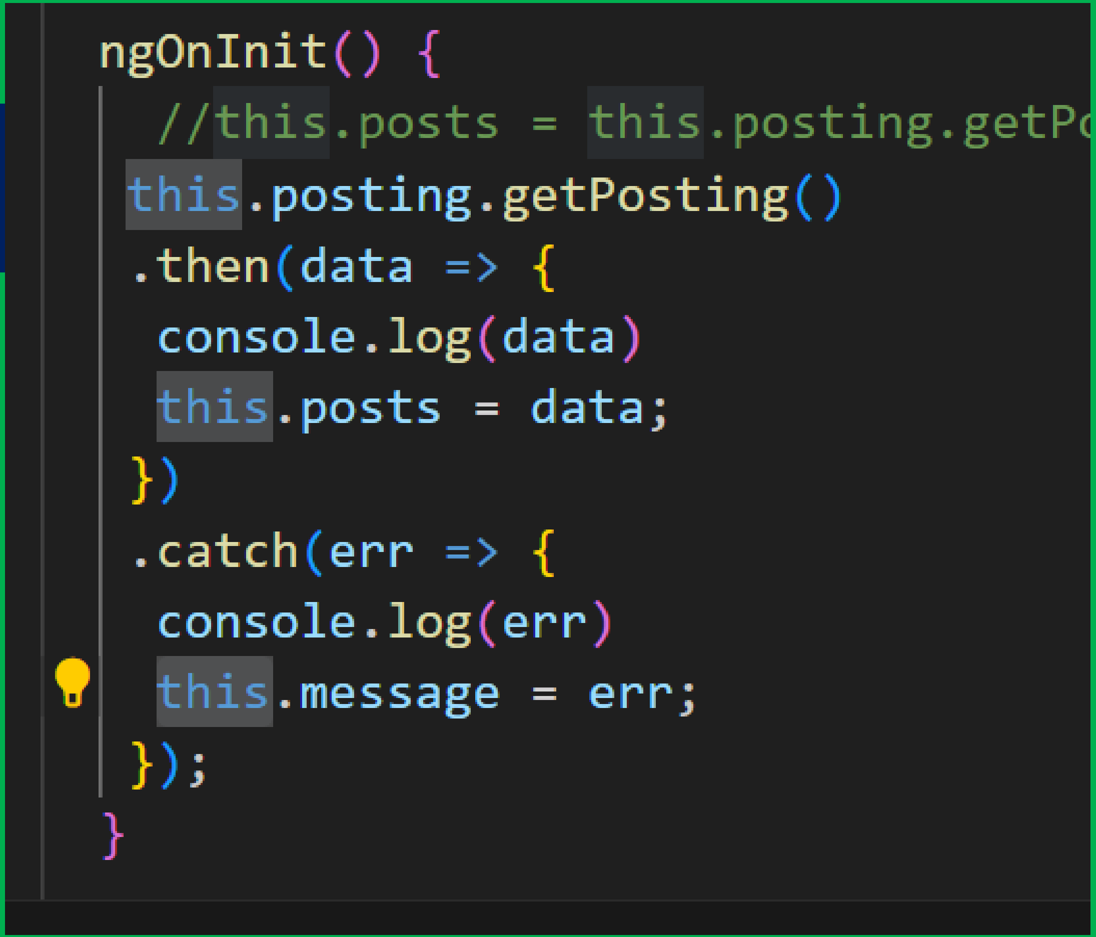
```
[ng] src_app_home_home_module_ts.js | home-home-module | 13.86 kB |
[ng]
[ng] 70 unchanged chunks
```

2. On the home.page.ts

THEN CATCH METHOD

.then()

- Handles success.
- Can be chained for sequential asynchronous operations.
- Each `.then()` receives the return value of the previous `.then()`.

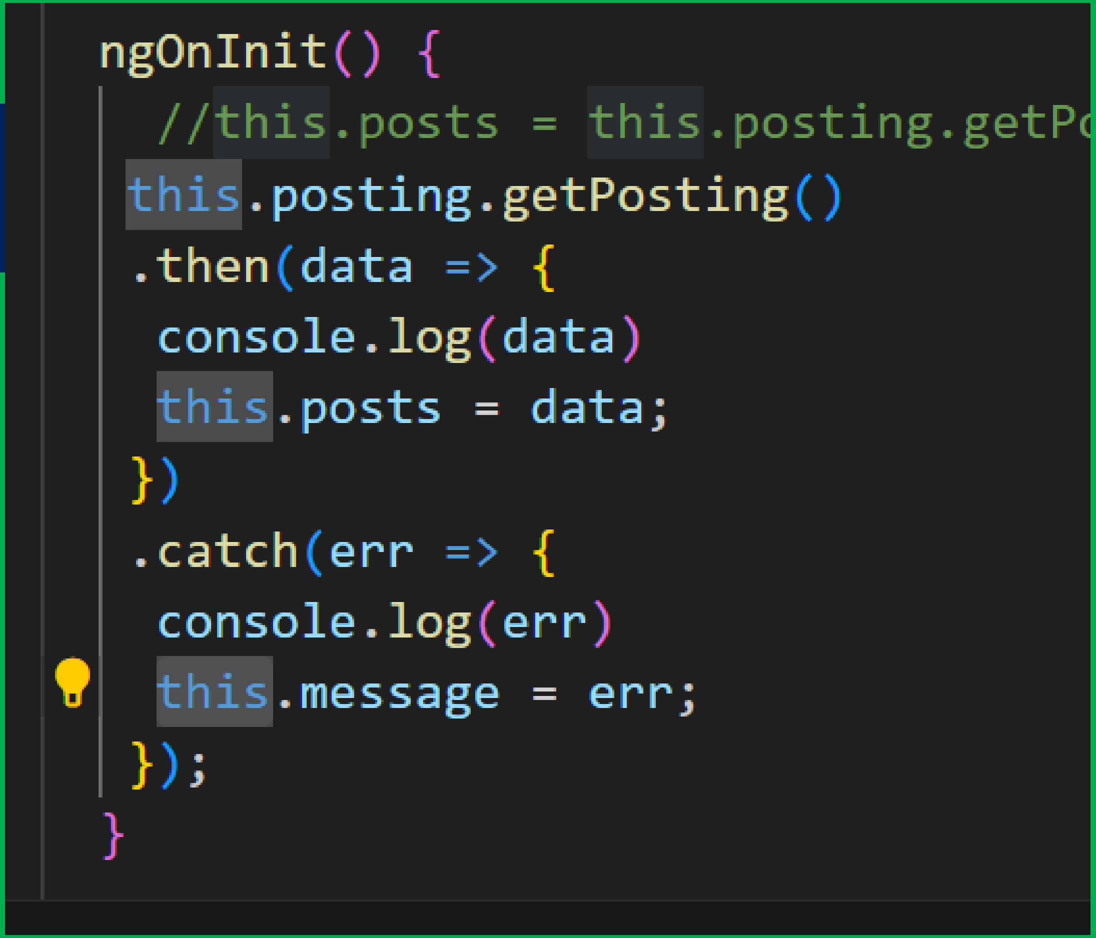


```
ngOnInit() {
  //this.posts = this.posting.getPo
 this.posting.getPosting()
 .then(data => {
  console.log(data)
  this.posts = data;
 })
 .catch(err => {
  console.log(err)
  this.message = err;
 });
}
```

THEN CATCH METHOD

.catch()

- Handles errors and rejections.
- Can catch errors from any previous `.then()` in the chain.
- Should be placed at the end of the promise chain to handle any errors from the preceding operations.



```
ngOnInit() {
  //this.posts = this.posting.getPo
 this.posting.getPosting()
 .then(data => {
  console.log(data)
  this.posts = data;
 })
 .catch(err => {
  console.log(err)
  this.message = err;
 });
}
```

ASYNC

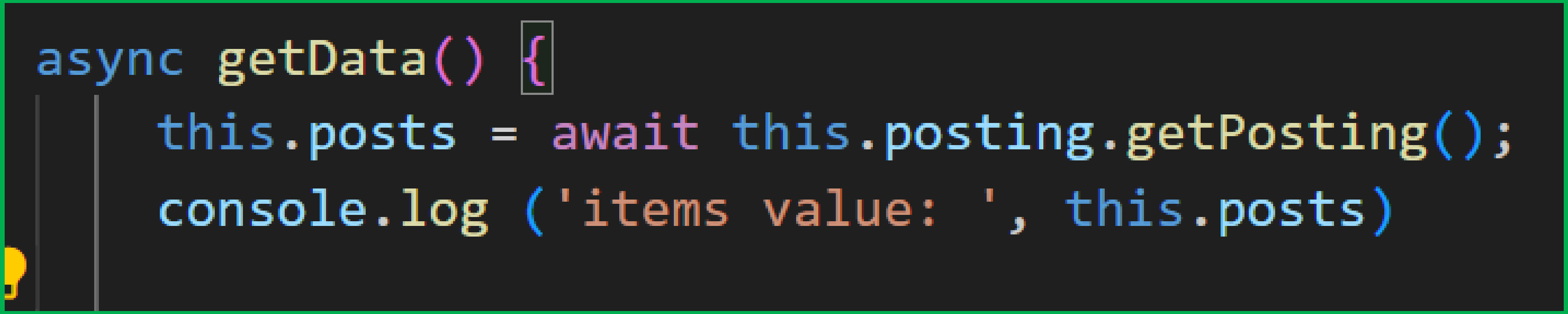
- keyword in JavaScript that is used to define asynchronous functions, allowing you to write code that handles asynchronous operations more easily and readably. Along with the await keyword, it simplifies working with promises.

- Function Declaration: When you declare a function with async, it automatically returns a promise.

ASYNC

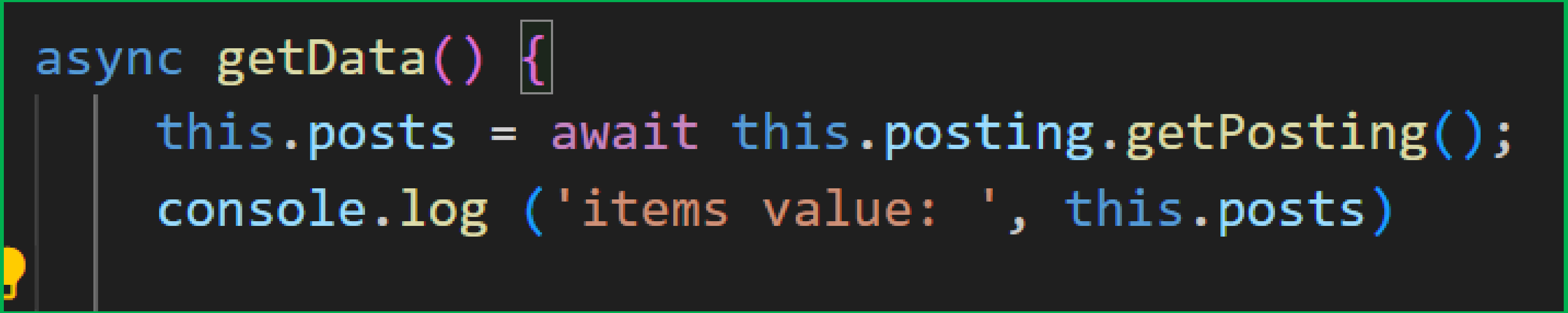
- Return Value: If the function returns a value, it is wrapped in a resolved promise.

- Exceptions: If the function throws an exception, it is wrapped in a rejected promise.



```
async getData() {
    this.posts = await this.posting.getPosting();
    console.log ('items value: ', this.posts)
```

AWAIT

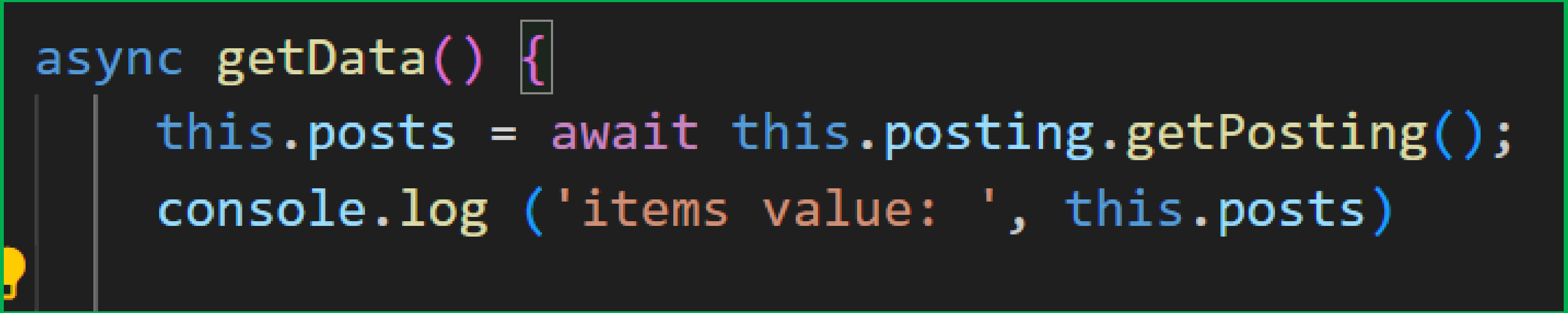


```
async getData() {
    this.posts = await this.posting.getPosting();
    console.log ('items value: ', this.posts)
```

- Purpose: await is used inside async functions to pause the execution of the function until the promise is resolved or rejected.

- Behavior: It makes the function wait for the promise and returns the resolved value or throws the rejected value.

AWAIT

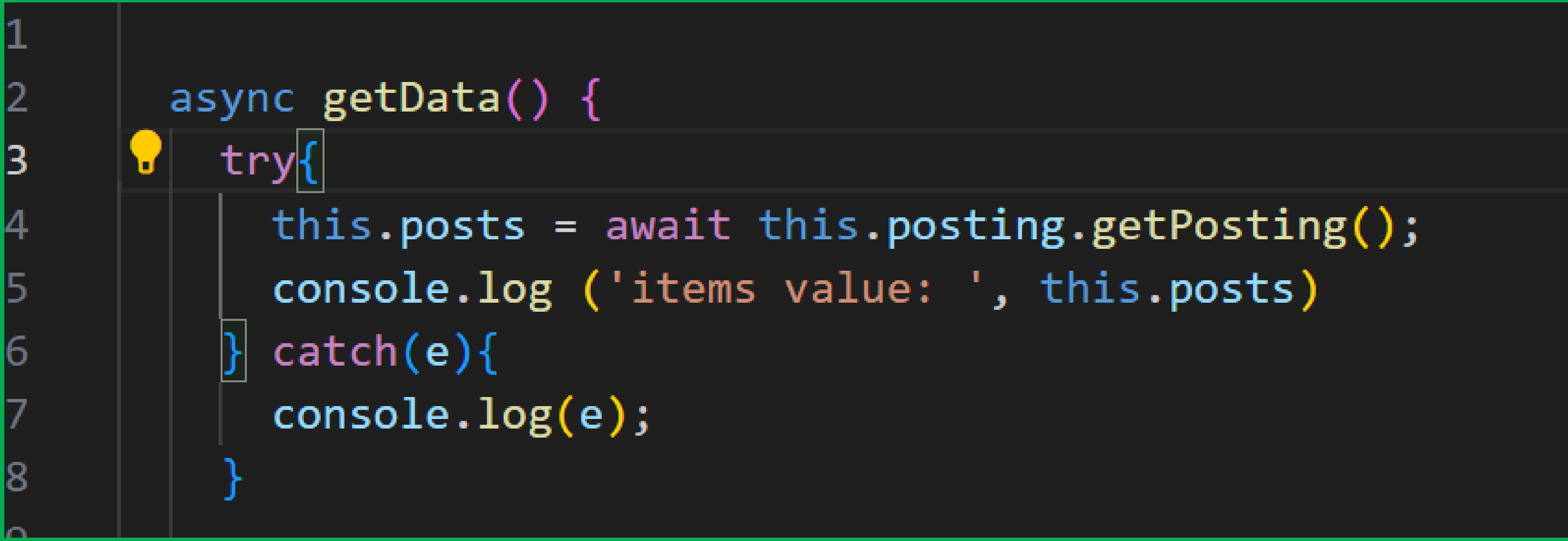


```
async getData() {
    this.posts = await this.posting.getPosting();
    console.log ('items value: ', this.posts)
```

- Purpose: await is used inside async functions to pause the execution of the function until the promise is resolved or rejected.

- Behavior: It makes the function wait for the promise and returns the resolved value or throws the rejected value.

TRY CATCH METHOD



```
 async getData() {
   try{
     this.posts = await this.posting.getPosting();
     console.log ('items value: ', this.posts)
   } catch(e){
     console.log(e);
   }
```

- try...catch is used for handling exceptions in synchronous code.
- It allows you to run code that might throw an error and handle that error gracefully.
- In asynchronous code with async/await, try...catch can be used to handle rejected promises and exceptions.