P.PORTO

REST API & WebSockets

Programação em Ambiente Web

Índice

Introdução

REST (Representational State Transfer)

Roy Fielding definiu REST na sua dissertação de doutoramento em 2000 "Architectural Styles and the Design of Network-based Software Architectures“

Fonte: https://en.wikipedia.org/wiki/File:Roy_Fielding_at_OSCON_2008.jpg

Introdução

API (Application Program Interface): é um conjunto de funções e procedimentos que permitem a criação de aplicações que acedem a recursos ou dados de um sistema, aplicação ou outro serviço

REST API: definição de apis baseadas no protocolo HTTP universal

Introdução

Um serviço REST, contêm um padrão de software arquitetural que define um conjunto de restrições e propriedades baseados em métodos HTTP

Web Services que obedecem ao padrão arquitetural REST, ou web services RESTful, fornecem interoperabilidade entre sistemas na Internet através da representação dos recursos sobre a forma de texto

Existem outro tipo de web services, como web services SOAP, expõem seus próprios conjuntos arbitrários de operações

Introdução

Num web service RESTful, pedidos feitos a um URI de um recurso obterá uma resposta que pode estar em XML, HTML, JSON ou algum outro formato

Os métodos HTTP mais usados em serviços REST, são GET, POST, PUT, DELETE e que predefinem o CRUD em HTTP

Introdução

Os serviços REST proporcionam

- desempenho rápido,
- confiabilidade
- reutilização de componentes

O REST ignora os detalhes da implementação de componentes e a sintaxe de protocolos com o objetivo de se focar

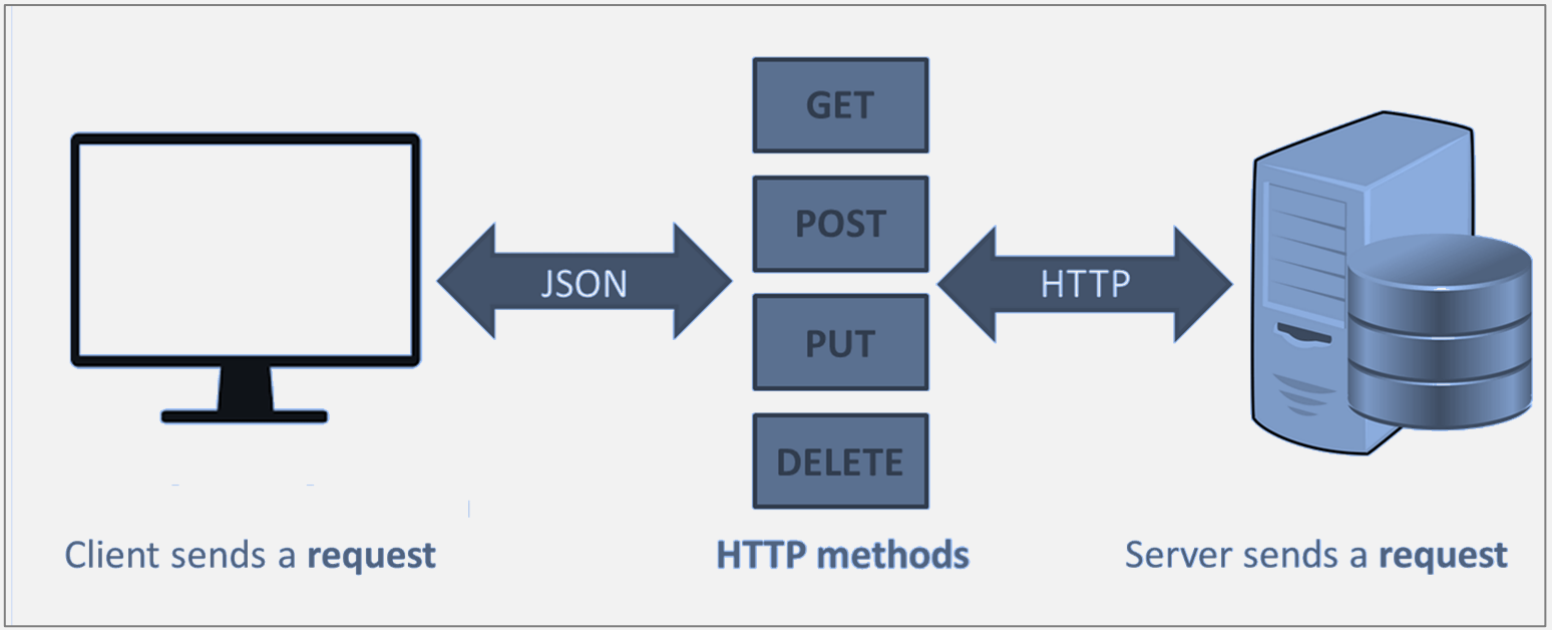
- no papel dos componentes
- nas restrições sobre sua interação com outros componentes

Introdução

Propriedades de serviços REST

- Arquitetura cliente-servidor
- Statelessness
- Cacheability
- Layered system
- Uniform interface

Introdução


GET
POST
PUT
DELETE
JSON
HTTP
Client sends a **request**
HTTP methods
Server sends a **request**

REST API

Os quatro métodos HTTP a seguir são comumente usados na arquitetura baseada em REST:

- GET - é usado para fornecer acesso somente leitura a um recurso.
- PUT - é usado para criar um novo recurso.
- DELETE - é usado para remover um recurso.
- POST - é usado para atualizar um recurso existente ou criar um novo recurso.

Introdução

Métodos HTTP comumente utilizados por API REST

Método	Objectivo
GET	Obtêm recursos do servidor
POST	Cria um novo recurso no servidor
PUT	Atualiza um recurso no servidor
DELETE	Elimina um recurso no servidor

REST API

Exemplo em node.js :

```
/* GET product listing. */
router.get('/products', productController.getAllProducts);
router.post('/products', productController.createProduct);

router.get('/product/:productId', productController.getOneProduct);
router.put('/product/:productId', productController.updateProduct);
router.delete('/product/:productId', productController.deleteProduct);
```

Serviços REST não exigem que o cliente saiba alguma coisa sobre a estrutura da API

O servidor especifica a localização do recurso e os campos obrigatórios. O browser não sabe antecipadamente onde enviar as informações e não sabe de antemão quais informações enviar

REST API

Problema:

- Vamos considerar o problema de gerir o stock de um produto
- Iremos criar um conjunto de serviços web armazenar informação sobre produtos e as suas quantidades

REST API

Antes de começar um projeto que defina uma API do tipo REST devemos refletir sobre o tipo de operações que devemos permitir

De seguida devemos associar os métodos do protocolo HTTP a cada uma dessas ações

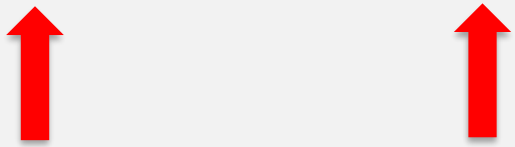
Por fim definimos o endereço de cada uma destas ações

REST API

Método HTTP	Path	Input	Output
GET	/products	-	lista de produtos
POST	/products	produto	-
GET	/product/prodID	prodID	Produto
DELETE	/product/prodID	prodID	-
PUT	/product/prodID	/product/prodID	-

REST API

Método HTTP	Path	Input	Output
GET	/products	-	lista de produtos
POST	/products	produto	-
GET	/product/prodID	prodID	Produto
DELETE	/product/prodID	prodID	-
PUT	/product	produto	-



REST API

Podemos agora criar o projeto REST API

User o gerador Express:

- "express RESTAPI"

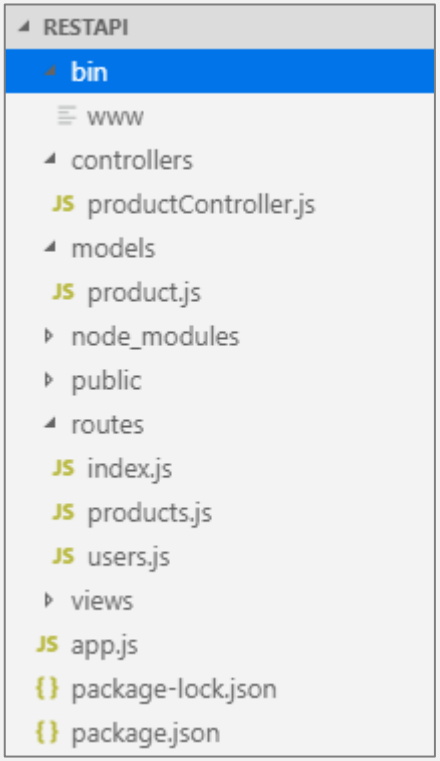
Criar o padrão MVC:

- Criar controladores
- Criar modelos
- As views serão criadas pelo middleware no formato JSON

REST API

O projeto ter a estrutura exemplificada na figura ao lado

Devemos sempre que possível estruturar a aplicação da melhor forma possível



REST API

Criar o ficheiro product.js na pasta models

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var ProductSchema = new Schema({
    name: { type: String  },
    description: {type: String},
    quantity: {type: Number}
  });

  module.exports = mongoose.model('Product', ProductSchema);
```

REST API

Criamos o ficheiro productController.js dentro da pasta controllers

Definimos todas as ações do controlador do produto e a interação com uma base de dados em mongoDB

```
var mongoose = require("mongoose");
var Product = require("../models/product");

var productController = {};

productController.createProduct = function (req, res, next) {
    var product = new Product(req.body);

    product.save(function (err) {
      if (err) {
        next(err);
      } else {
        res.json(product);
      }
    });
  };

productController.updateProduct = function (req, res, next) {
  Product.findByIdAndUpdate(req.body._id, req.body, {new: true}, function
  (err, product) {
      if (err) {
        next(err);
      } else {
        res.json(product);
      }
    });
  };
```

REST API

Criamos o ficheiro productController.js dentro da pasta controllers

Definimos todas as ações do controlador do produto e a interação com uma base de dados em mongoDB

```
productController.deleteProduct = function (req, res, next) {
    req.product.remove(function (err) {
      if (err) {
        next(err);
      } else {
        res.json(req.product);
      }
    });
  };

productController.getAllProducts = function (req, res, next) {
  Product.find(function (err, products) {
      if (err) {
        next(err);
      } else {
        res.json(products);
      }
    });
  };

productController.getOneProduct = function (req, res) {
    res.json(req.product);
  };
```

REST API

Criamos o ficheiro productController.js dentro da pasta controllers

Definimos todas as acções do controloador do produto e a interação com uma base de dados em mongoDB

```
productController.getByIdProduct = function (req, res, next, id) {
  Product.findOne({_id: id}, function (err, product) {
      if (err) {
        next(err);
      } else {
        req.product = product;
        next();
      }
    });
  };


module.exports = productController;
```

REST API

Criar o ficheiro product.js dentro da pasta routes

Aqui definimos os métodos http para cada ação

```
var express = require('express');
var router = express.Router();
var productController = require("../controllers/productController.js");

/* GET product listing. */
router.get('/products', productController.getAllProducts);
router.post('/products', productController.createProduct);

router.get('/product/:productId', productController.getOneProduct);
router.put('/product/:productId', productController.updateProduct);
router.delete('/product/:productId', productController.deleteProduct);

router.param('productId', productController.getByIdProduct);

module.exports = router;
```

REST API

Atualizamos o ficheiro app.js com a informação relevante para servir a API REST

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

var productsRouter = require('./routes/products');
var mongoose = require('mongoose');
mongoose.Promise = global.Promise;

mongoose.connect('mongodb://localhost:27017/product-demo')
  .then(() =>  console.log('connection succesful'))
  .catch((err) => console.error(err));

var app = express();

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/api/v1', productsRouter);
```

REST API

Atualizamos o ficheiro app.js com a informação relevante para servir a API REST

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

REST API

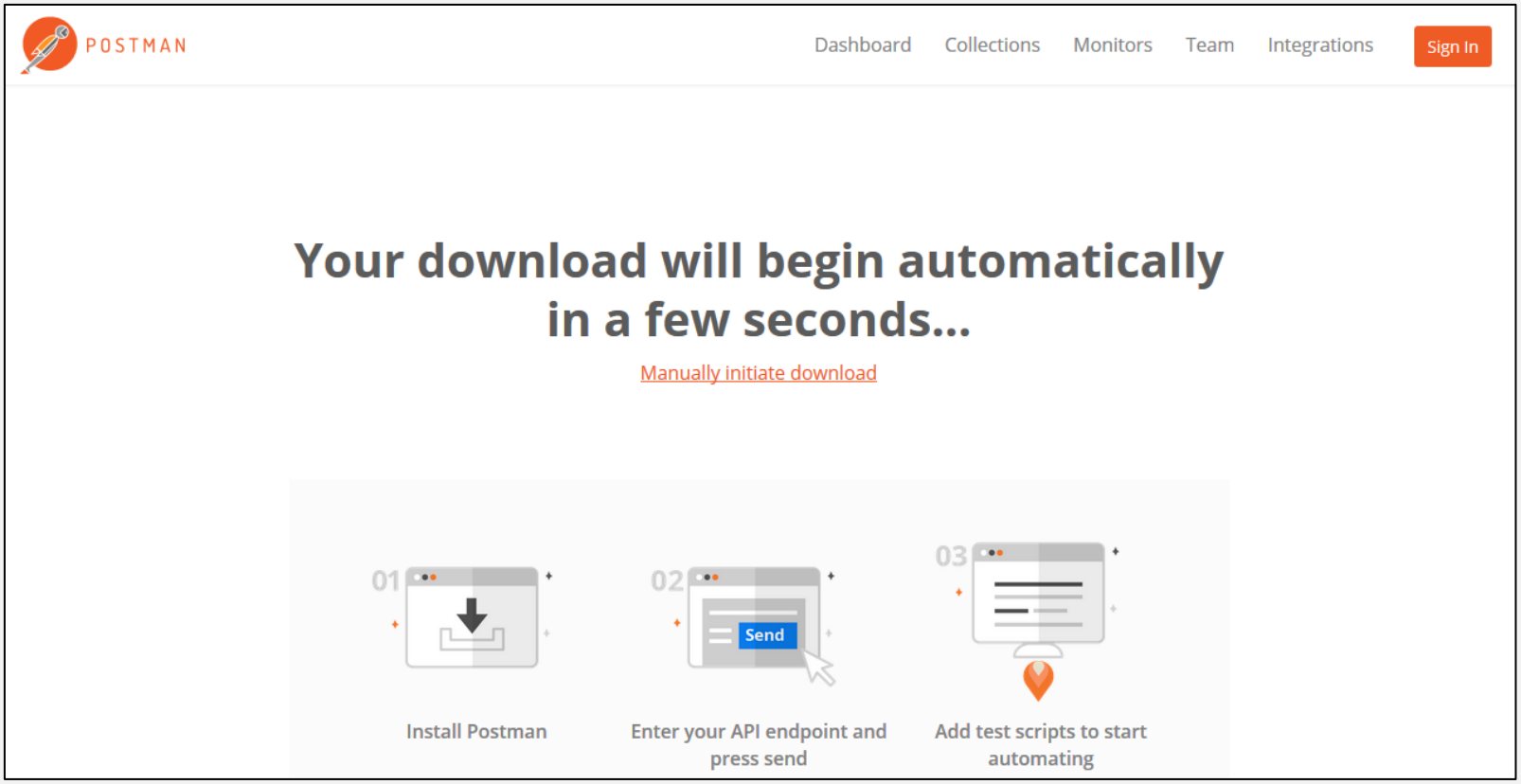
Podemos usar ferramentas externas para o teste da nossa API

Uma das mais conhecidas é a extensão postman que permite:

- Enviar pedidos REST
- Definir parâmetros na área Hearder e Body dos pedidos HTTP
- Verificar as respostas do servidor
- Criar projetos para teste e validação de APIs

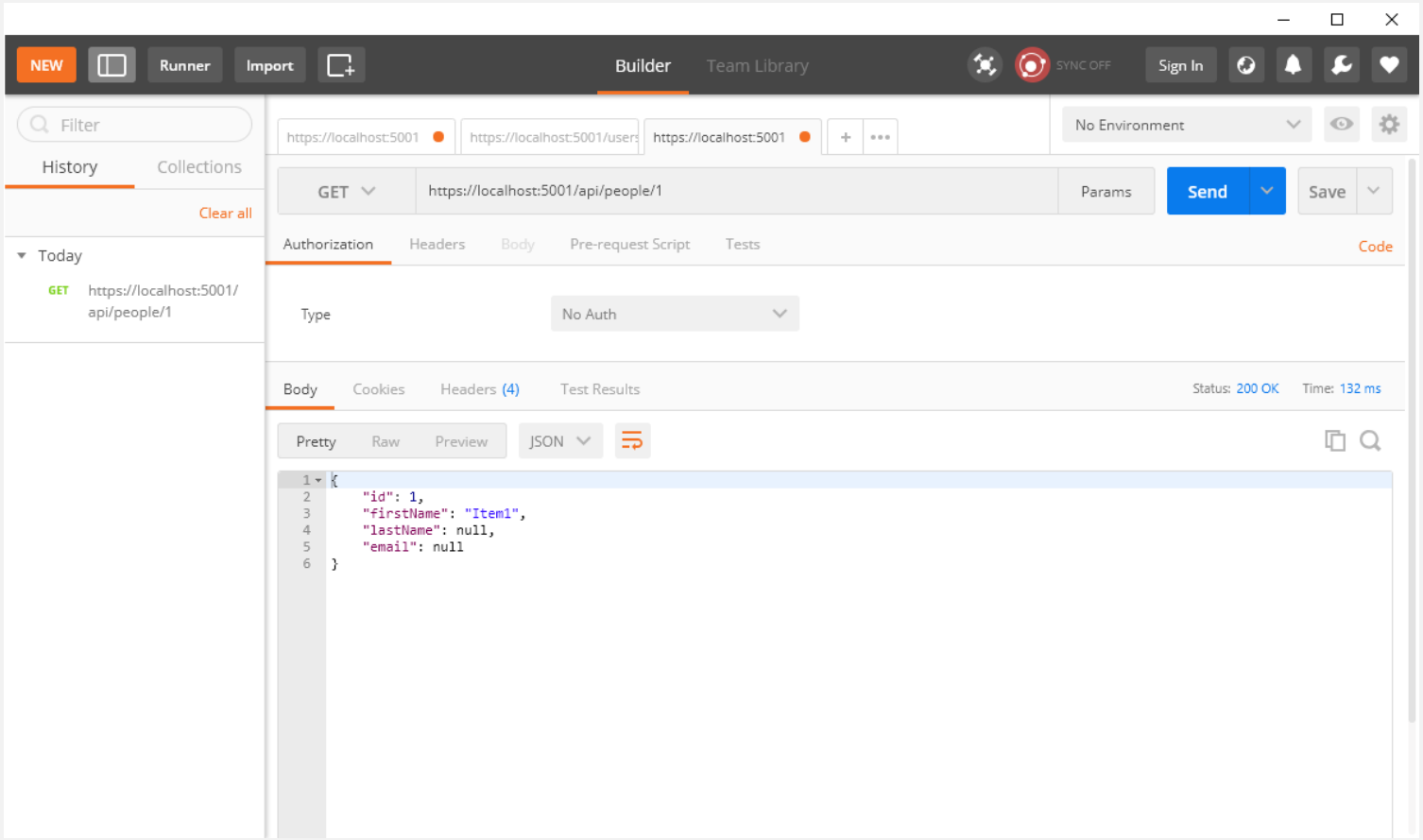
https://www.getpostman.com/

REST API



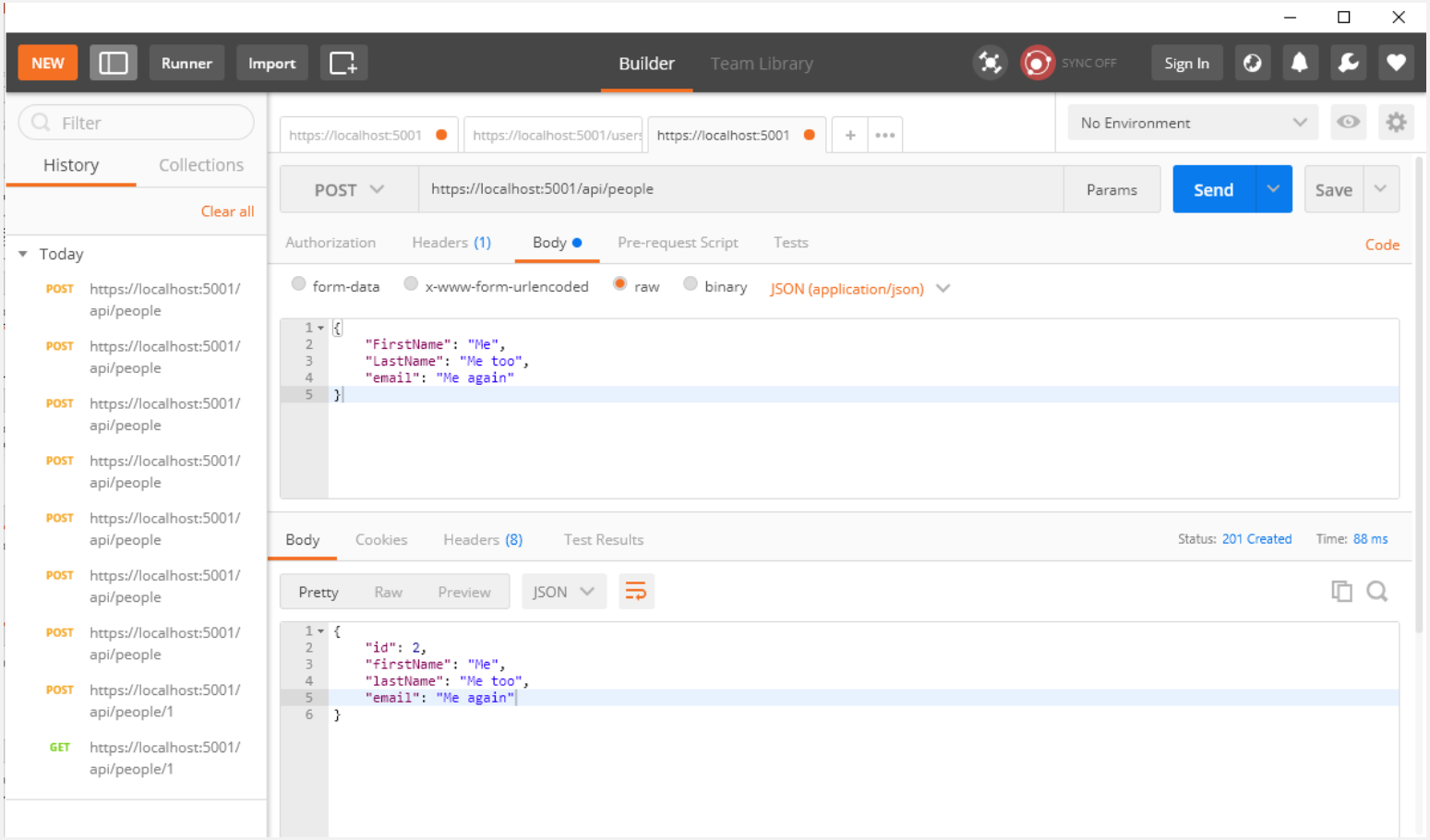
Fonte: https://app.getpostman.com/app/download/win64

REST API



Teste de um pedido get

REST API



Teste de um pedido post com parâmetros no Body do request HTTP

REST API

Demonstração Prática

(Postman)

REST API

Os serviços REST podem ser diretamente acedidos por páginas HTML simples usando pedidos AJAX ao servidor

Contudo é necessário conhecimento sobre a API REST para interagir com serviços REST

Devemos sempre documentar os nossos serviços REST

REST API

Por defeito, os browsers irão bloquear pedidos a servidores com domínio diferente de onde a página foi carregada

Para que os nossos web services REST sejam acessíveis por domínios de outras páginas web é preciso que se definam headers específicos nas mensagens HTTP

REST API

Devemos instalar o package cors com o comando:

- “npm i cors --save"

Dentro do ficheiro app.js

```
var express = require('express');
var cors = require('cors');
var app = express();

app.use(cors());
```

REST API

Podemos também apenas aplicar CORS a apenas um único tipo de pedido com funções de middleware diretamente no router em express

```
var express = require('express');
var cors = require('cors');
var app = express();

app.get('/products/:id', cors(), function (req, res, next) {
    res.json({msg: 'This is CORS-enabled for a Single Route'});
});
```

REST API

Uma explicação completa sobre o funcionamento de politicas CORS para estudo futuro pode ser encontrado em:

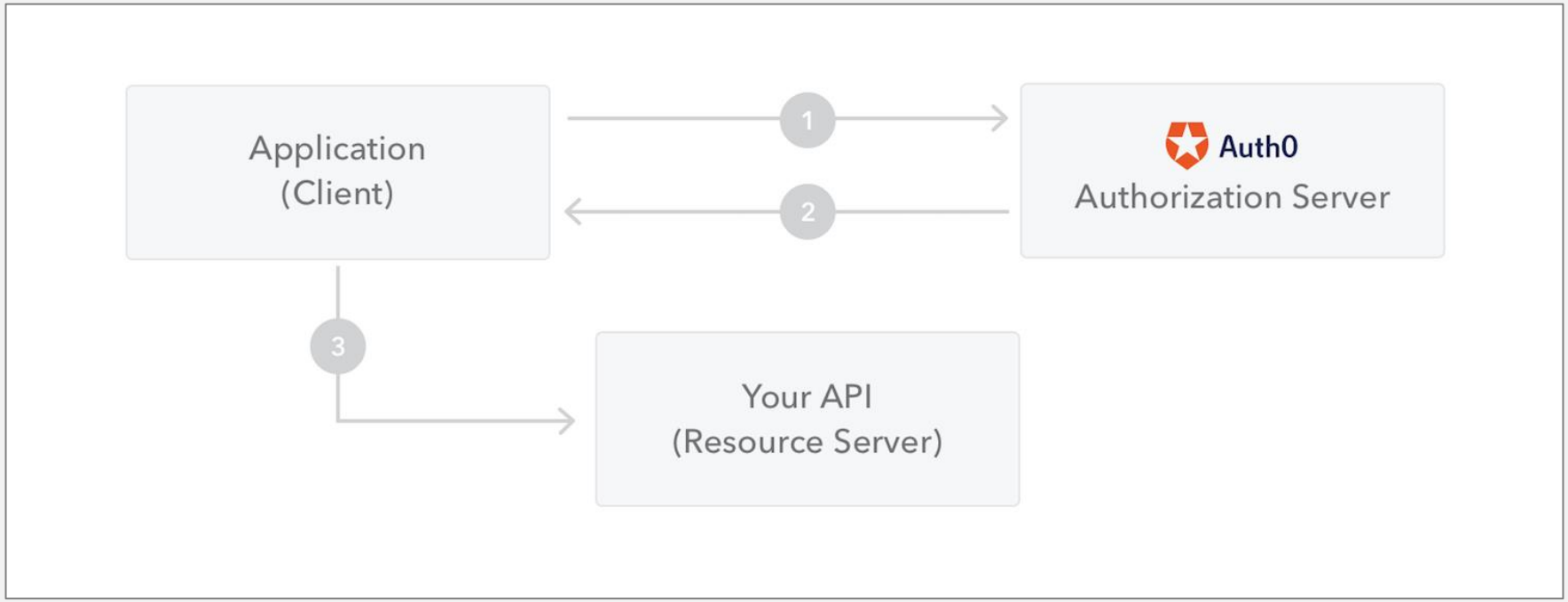
- https://flaviocopes.com/cors/

REST API e JWT

JSON Web Token (JWT) é um open standard (RFC 7519) que define um método compacto e autocontido para transmitir com segurança informações entre as partes num objeto JSON.

As JWTs podo ser assinados utilizando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA.

REST API e JWT



REST API e JWT

Embora JWTs possam ser criptografadas para também fornecer sigilo entre as partes, iremos focar-nos no use de signed tokens

Os signed tokens podem verificar a integridade da informação contidas nele, enquanto os tokens criptografados ocultam essas declarações de outras partes

Quando os tokens são assinados usando pares de chaves pública/privada, a assinatura também certifica que a parte que é prorietária da chave privada é aquela que a assinou

REST API e JWT

Em node.js podemos adicionar a verificação por JWT com o módulo:

- npm install jsonwebtoken --save

Devemos criar um controlador para tratar a autenticação e um segredo para criar tokens

REST API e JWT

Controlador para criar a autenticação:

```
// AuthController.js
var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
router.use(bodyParser.urlencoded({ extended: false }));
router.use(bodyParser.json());
var User = require('../user/User');
var jwt = require('jsonwebtoken');
var config = require('../config');
router.post('/register', function(req, res) {

  User.create({
    name : req.body.name,
    email : req.body.email,
    password : req.body.password
  },
  function (err, user) {
    if (err) return res.status(500).send("There was a problem registering
    the user.")
    // create a token
    var token = jwt.sign({ id: user._id }, config.secret, {
      expiresIn: 86400 // expires in 24 hours
    });
    res.status(200).send({ auth: true, token: token });
  });
});

module.exports = router;
```

REST API e JWT

Config.js

```
// config.js
module.exports = {
  'secret': 'supersecret'
};
```

Adicionar a app.js

```
// app.js
var AuthController = require('./auth/AuthController');
app.use('/api/auth', AuthController);
module.exports = app;
```

REST API e JWT

Verificar o acesso a um recurso que necessita de autenticação

```
router.get('/me', function(req, res) {
  var token = req.headers['x-access-token'];
  if (!token) return res.status(401).send({ auth: false, message: 'No token
  provided.' });

  jwt.verify(token, config.secret, function(err, decoded) {
    if (err) return res.status(500).send({ auth: false, message: 'Failed to
    authenticate token.' });

    res.status(200).send(decoded);
  });
});
```

Verificar a existência de um token nos header do pedido http e autenticar junto do módulo JWT

OpenAPI e Swagger

OpenAPI e Swagger

- Foram criados por um consórcio do setor que reconhecem o valor da padronização de como as APIs REST são descritas.
- Como uma estrutura de governance aberta sob a Linux Foundation, a OAI está focada na criação, evolução e promoção de um formato de descrição de API REST neutra de fornecedor.
- Criar um formato de descrição aberto para serviços de API que seja neutro, portátil e aberto é importante para melhorar o acesso e interoperabiliade dos sistemas

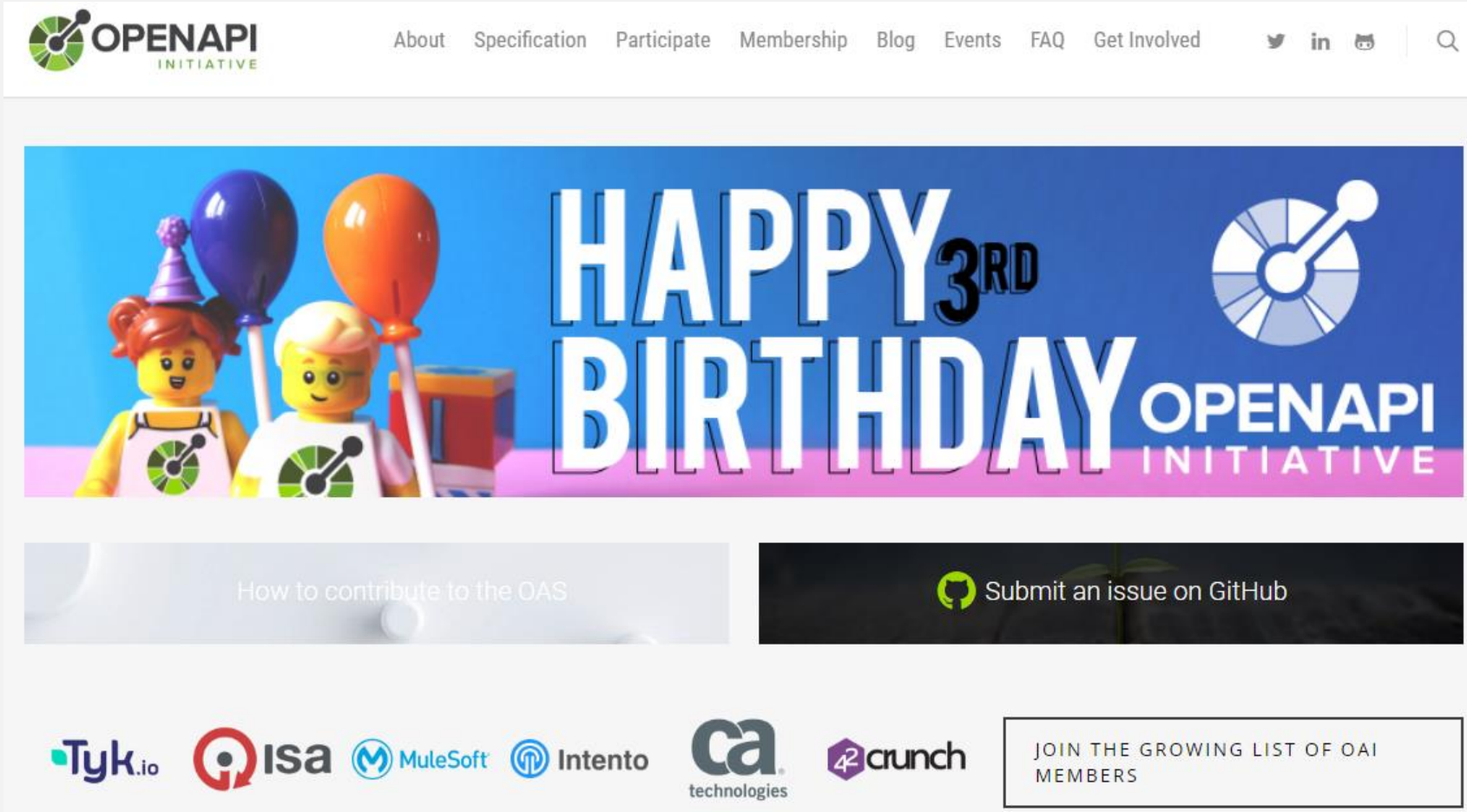
OpenAPI e Swagger

O openAPI define um conjunto de métodos e boas práticas na definição de webservices que são seguidos pela plataforma swagger

Principais características presentes na plataforma swagger:

- Design
- Build
- Document
- Test
- Standardize

OpenAPI e Swagger



Fonte:
https://www.openapis.org/

OpenAPI e Swagger

Fonte: https://swagger.io

OpenAPI e Swagger

Podemos adicionar módulos que nos ajudam com a tarefa de documentar a nossa API

Um deles é o “swagger-ui-express” desenhado para funcionar com o middleware express de node.js

OpenAPI e Swagger

Antes de o utilizar devemos-o instalar com o comando

- "npm i swagger-ui-express"

Devemos no projeto anterior sobre a API REST de produtos adicionar ao ficheiro app.js

```
var swaggerUi = require('swagger-ui-express');
var swaggerDocument = require('./swagger.json');
```

O ficheiro "swagger.json" será criado na raiz do projeto e serve para configurar os métodos da nossa api na documentação swagger

OpenAPI e Swagger

Devemos também adicionar o endereço onde a documentação sobre o nosso web servisse irá estar

```
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
app.use('/api/v1', productsRouter);
```

Finalmente necessitamos apenas de configurar o ficheiro swagger.json

OpenAPI e Swagger

Começamos por definir informação genérica sobre a API

```
{
  "swagger": "2.0",
  "info": {
    "version": "1.0.0",
    "title": "Product API",
    "description": "REST API for products"
  },
  "host": "localhost:3000",
  "basePath": "/api/v1",
  "tags": [
    {
      "name": "Products",
      "description": "API for products in the system"
    }
  ],
  "schemes": [
    "http"
  ],
  "consumes": [
    "application/json"
  ],
  "produces": [
    "application/json"
  ],
```

OpenAPI e Swagger

Para cada endereço da nossa api é especificado

- o método http
- parametros
- resposta e modelo da resposta
- o que o serviços produz

```
"paths": {
  "/products": {
    "post": {
      "tags": [
        "Products"
      ],
      "description": "Create new user in system",
      "parameters": [
        {
          "name": "product",
          "in": "body",
          "description": "Product that we want to create",
          "schema": {
            "$ref": "#/definitions/Product"
          }
        }
      ],
      "produces": [
        "application/json"
      ],
      "responses": {
        "200": {
          "description": "New product is created",
          "schema": {
            "$ref": "#/definitions/Product"
          }
        }
      }
    },
```

OpenAPI e Swagger

Para cada endereço da nossa api é especificado

- o método http
- parametros
- resposta e modelo da resposta
- o que o serviços produz

```
  "get": {
    "tags": [
      "Products"
    ],
    "summary": "Get all products in system",
    "responses": {
      "200": {
        "description": "OK",
        "schema": {
          "$ref": "#/definitions/Products"
        }
      }
    }
  }
},
"/product/{productId}": {
  "parameters": [
    {
      "name": "productId",
      "in": "path",
      "required": true,
      "description": "ID of product that we want to find",
      "type": "string"
    }
  ],
```

OpenAPI e Swagger

Para cada endereço da nossa api é especificado

- o método http
- parametros
- resposta e modelo da resposta
- o que o serviços produz

```
"get": {
  "tags": [
    "Products"
  ],
  "summary": "Get product with given ID",
  "responses": {
    "200": {
      "description": "Product is found",
      "schema": {
        "$ref": "#/definitions/Product"
      }
    }
  }
},
"delete": {
  "summary": "Delete product with given ID",
  "tags": [
    "Products"
  ],
  "responses": {
    "200": {
      "description": "Product is deleted",
      "schema": {
        "$ref": "#/definitions/Product"
      }
    }
  }
},
```

OpenAPI e Swagger

Para cada endereço da nossa api é especificado

- o método http
- parametros
- resposta e modelo da resposta
- o que o serviços produz

```
    "put": {
      "summary": "Update product with give ID",
      "tags": [
        "Products"
      ],
      "parameters": [
        {
          "name": "product",
          "in": "body",
          "description": "Product with new values of properties",
          "schema": {
            "$ref": "#/definitions/Products"
          }
        }
      ],
      "responses": {
        "200": {
          "description": "Product is updated",
          "schema": {
            "$ref": "#/definitions/Product"
          }
        }
      }
    }
  }
},
```

OpenAPI e Swagger

É definido no final também os modelos de dados usados nos webservices

```
"definitions": {
  "Product": {
    "required": [
      "name",
      "_id"
    ],
    "properties": {
      "_id": {
        "type": "string",
        "uniqueItems": true
      },
      "name": {
        "type": "string",
        "uniqueItems": true
      },
      "description": {
        "type": "string"
      },
      "quantity": {
        "type": "Number"
      }
    }
  },
  "Products": {
    "type": "array",
    "$ref": "#/definitions/Product"
  }
}
}
```

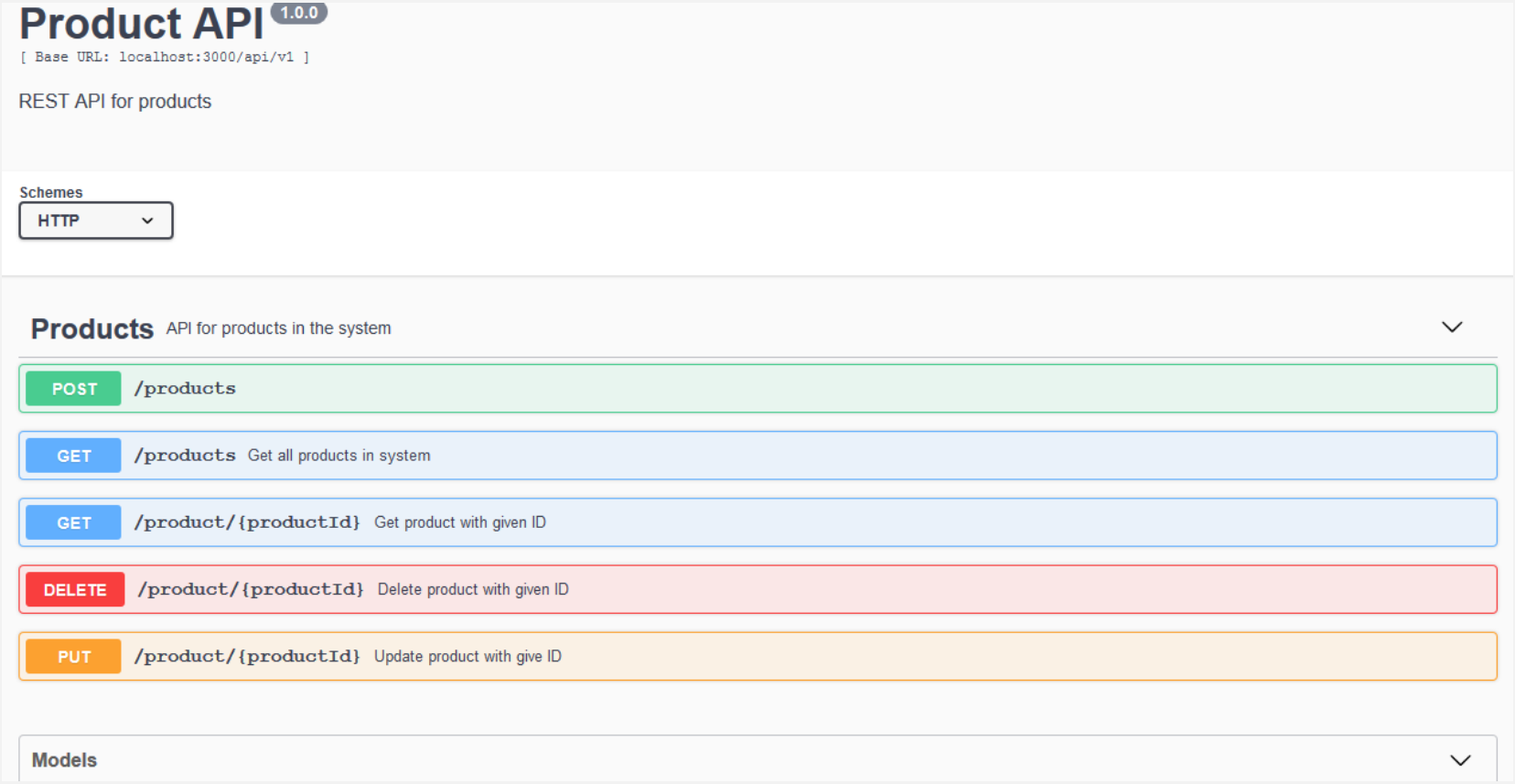
OpenAPI e Swagger

Após as alterações ao projeto, devemos ser agora capazes de usar a plataforma swagger para documentação teste da nossa api

Podemos consultar a informação através do endereço:

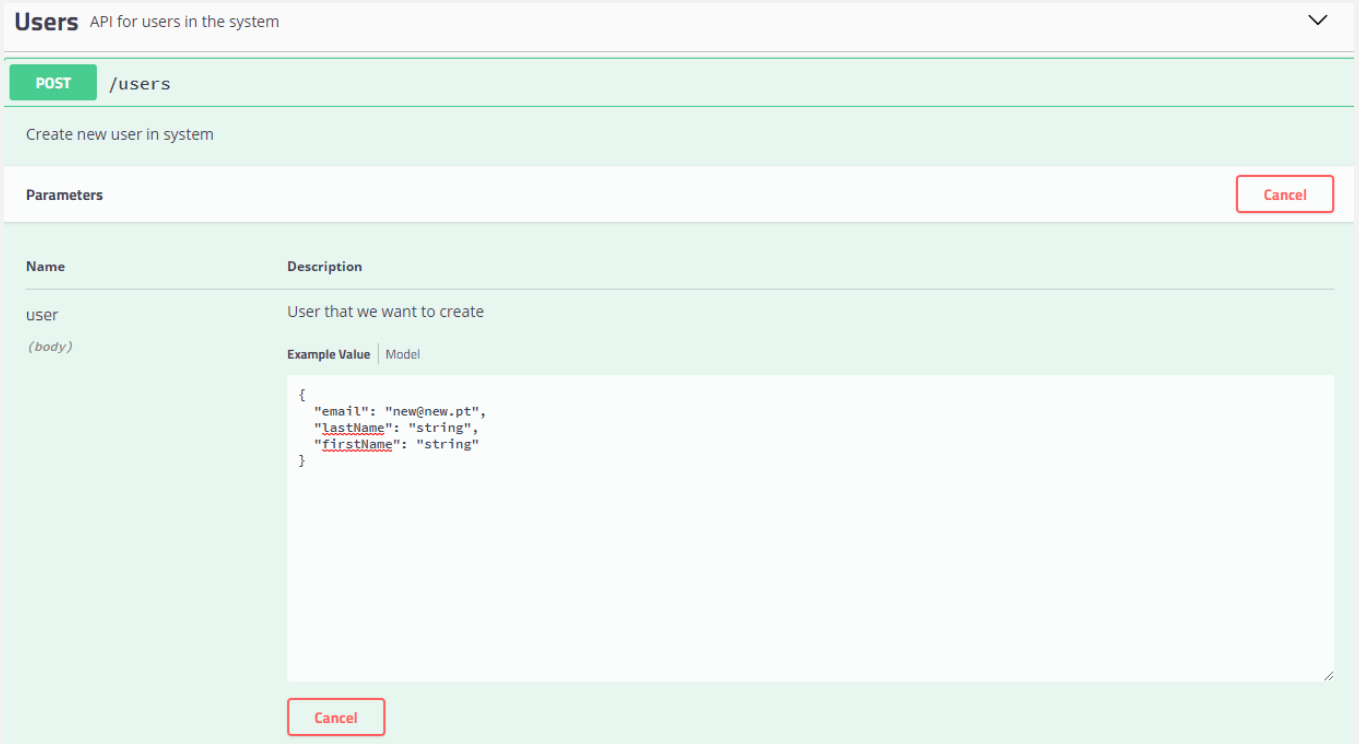
- http://localhost:3000/api-docs

OpenAPI e Swagger



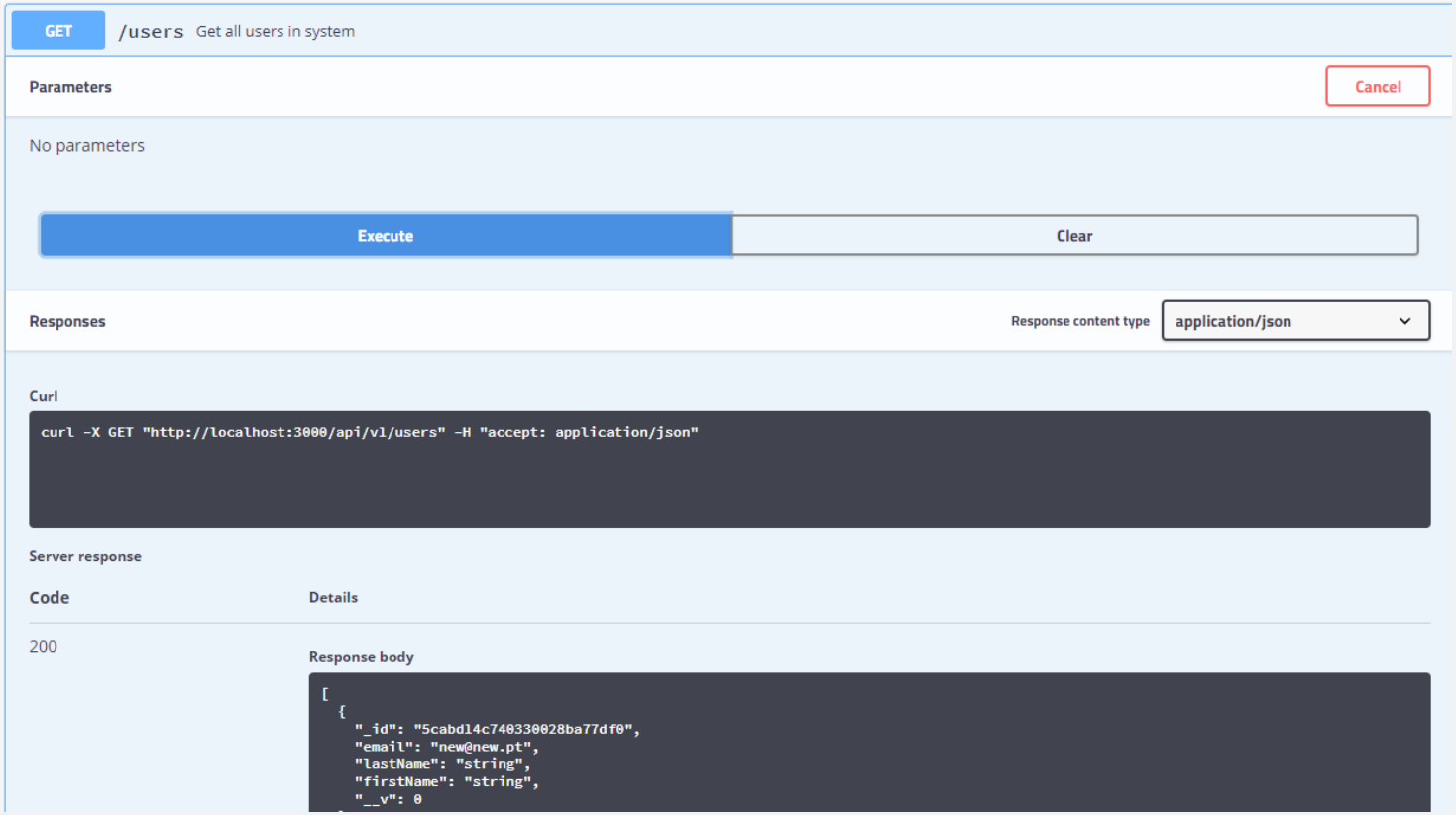
OpenAPI e Swagger

Podemos invocar diretamente a partir do swagger os nossos serviços REST e obter o feedback sobre eles



OpenAPI e Swagger

Podemos invocar diretamente a partir do swagger os nossos serviços REST e obter o feedback sobre eles



API Requests

Podemos também invocar web services de terceiros a partir de um backend

Para este efeito necessitamos de usar o módulo https e definir um callback para o sucesso ou insucesso da operação

Podemos encapsular chamadas a APIs de terceiros dentro das nossas APIs REST

API Requests

Exemplo de pedido GET ao endereço exemplo.com

```
const https = require('https')
const options = {
  hostname: 'exemplo.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}

const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})

req.on('error', (error) => {
  console.error(error)
})

req.end()
```

API Requests

Exemplo de pedido POST ao endereço exemplo.com

Os métodos put e delete também são possíveis alterando o method no objeto options

```
const https = require('https')
const data = JSON.stringify({
  item: 'item1'
})

const options = {
  hostname: 'exemplo.com',
  port: 443,
  path: '/item',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length
  }
}

const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})

req.on('error', (error) => {
  console.error(error)
})

req.write(data)
req.end()
```

WebSockets

WebSocket é uma tecnologia que permite a comunicação bidirecional por canais full-duplex sobre um único socket TCP (Transmission Control Protocol)

Foi desenhado para ser usado em browsers e servidores web que suportem o HTML5

A API WebSocket é definida pelo W3C e o protocolo WebSocket é definido pelo IETF

O protocolo Websocket é um protocolo independente baseado em TCP

Usa apenas o HTTP para fazer "handshake", é interpretado por servidores HTTP como uma requisição de Upgrade

WebSockets

Todos os browsers atuais devem suportar a última especificação do protocolo Websocket.

Para referência uma lista dos browser que suportam websockets:

- Internet Explorer 10+
- Mozilla Firefox 4+
- Safari 5+
- Google Chrome 4+
- Opera 11+[7]

WebSockets

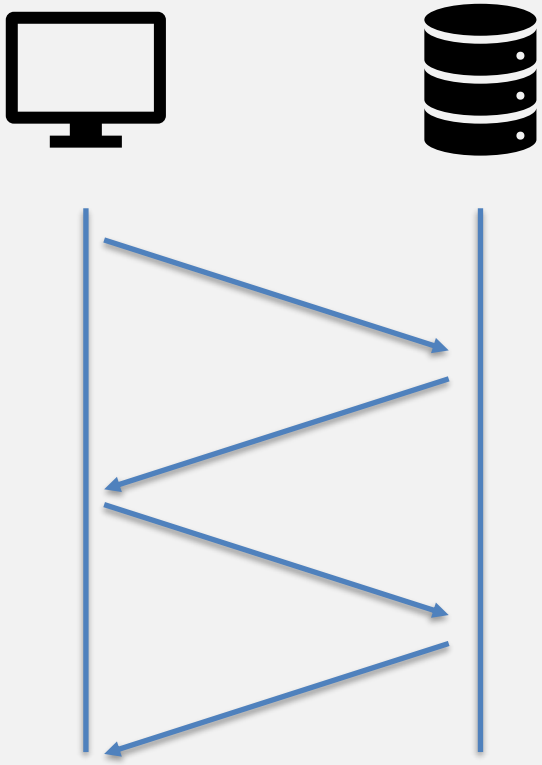
O protocolo WebSocket é uma extensão do ecossistema HTTP que permite criar conexões ativas entre um servidor da web e um browser

Permite que os aplicações da web troquem dados eficientemente e em tempo real sem a sobrecarga das conexões HTTP convencionais

Há muitas maneiras de usar WebSockets e há muitos módulos que você encontrará no NPM (por exemplo, Socket.io, ws, etc.)

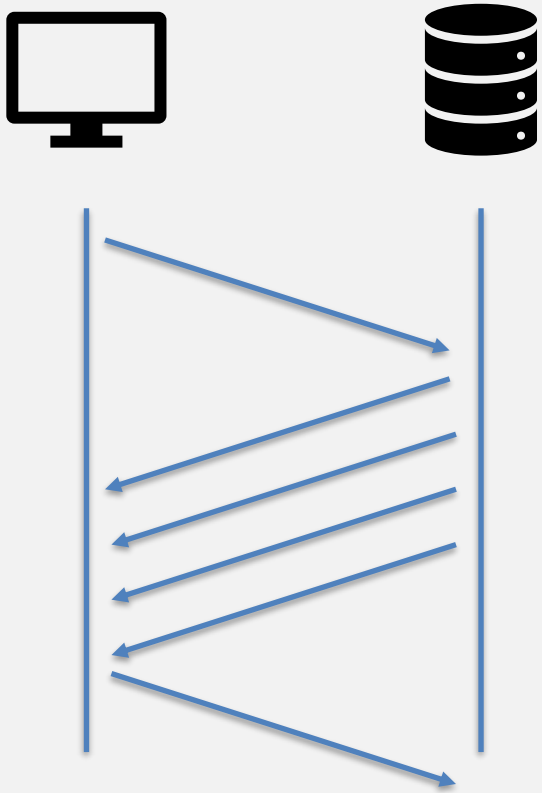
WebSockets

Modelo cliente-servidor



WebSockets

Websockets



WebSockets

Conceitualmente, o protocolo WebSocket é uma extensão do HTTP que permite aos clientes "atualizar" uma conexão HTTP com uma conexão bidirecional:

- O cliente abre conexão HTTP ao servidor e pede documento
- O servidor responde com um documento HTML
- A conexão HTTP está fechada
- O código JavaScript no documento HTML abre outra conexão HTTP na qual ele pede ao servidor para atualizar essa conexão para uma conexão WebSocket
- Uma conexão WebSocket entre cliente e servidor é estabelecida e permanece aberta para envio e recebimento de dados em ambas as direções

Tecnicamente, uma conexão WebSocket é simplesmente uma conexão TCP na porta 80, com a diferença de que o cliente e o servidor tratam a conexão de uma maneira especial.

WebSockets

Em node.js será usada a biblioteca ws para criar Websockets

Podemos criar um servidor de webscokets com o comando:

- const wss = new SocketServer ({server}) ;.

WebSockets

Ponto de partida para construir um servidor WebSockets

```
const SocketServer = require('ws').Server;
var express = require('express');
var path = require('path');
var connectedUsers = [];

//init Express
var app = express();

//init Express Router
var router = express.Router();
var port = process.env.PORT || 80;

//return static page with websocket client
app.get('/', function(req, res) {
    res.sendFile(path.join(__dirname + '/static/index-with-websockets.html'));
});

var server = app.listen(port, function () {
    console.log('node.js static server listening on port: ' + port + ", with websockets listener")
})

const wss = new SocketServer({ server });

//init Websocket ws and handle incoming connect requests
wss.on('connection', function connection(ws) {
    console.log("connection ...");

    //on connect message
    ws.on('message', function incoming(message) {
        console.log('received: %s', message);
        connectedUsers.push(message);
    });

    ws.send('message from server at: ' + new Date());
});
```

WebSockets

No Browser:

1. Abrir uma conexão WebSocket (wss)
2. Callback opcional caso haja algum erro
3. Callback opcional caso a coneção termine
4. Callback opcional quando a conexão inicia
5. Mensagem do cliente para o servidor
6. Callback para cada mensagem recebida do servidor
7. Distinguir entre mensagem de texto ou binária

```
<script>
  var ws = new WebSocket('wss://localhost:3000/socket');  1

  ws.onerror = function(error){ console.log('websocket error.')} 2
  ws.onclose = function(){ console.log('websocket closed.')} 3
  ws.onopen = function(){  4
    console.log('websocket open.');
    ws.send('Hello Server from client'); //msg to server  5
  }

  ws.onmessage = function(msg){ 6
    if (msg instanceof Blob){ 7

    }else{
      let temp = document.createElement('p');
      temp.innerText = msg.data;
      document.getElementById('serverMessages').appendChild(temp);
    }
  }
</script>
```

Referências

REST

- https://developer.mozilla.org/en-US/docs/Glossary/REST

Swager and OpenAPI

- https://github.com/swagger-api/swagger-node
- https://github.com/swagger-api/swagger-node/blob/master/docs/quick-start.md

WebScokets

- https://github.com/websockets/ws
- https://www.npmjs.com/package/websocket

P.PORTO

REST API & WebSockets

Programação em Ambiente Web