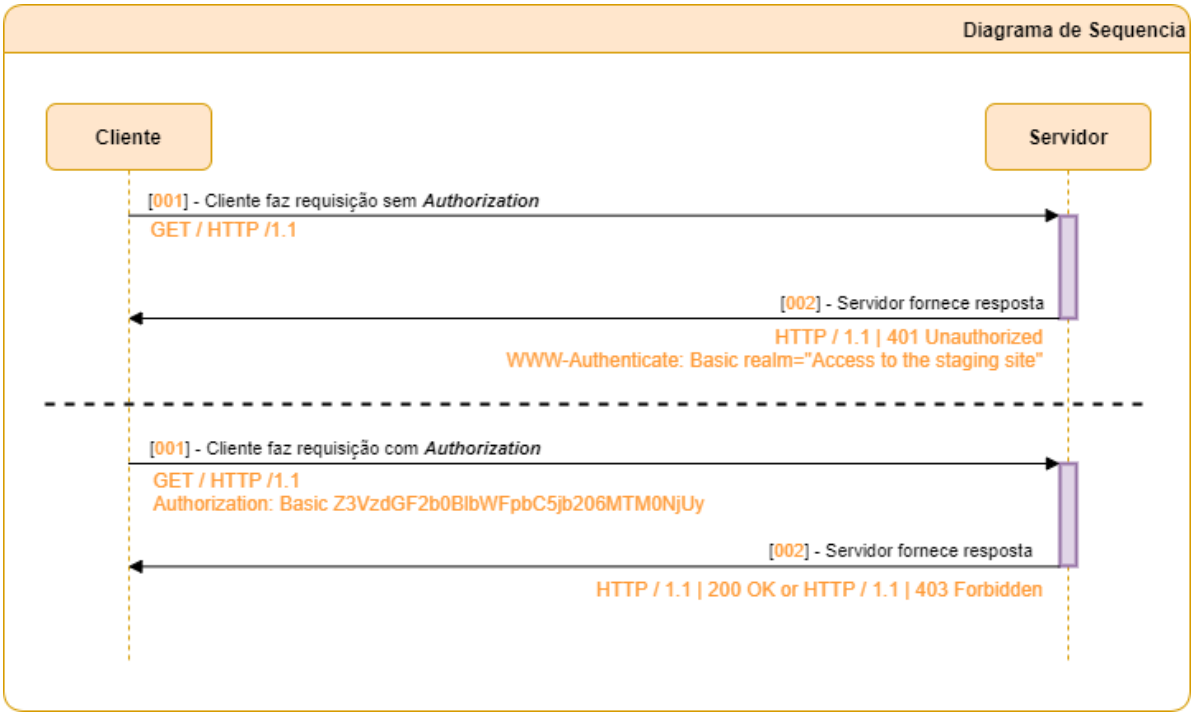
Projeto 02 - Blog Pessoal - Security

O que veremos por aqui:

1. Conhecendo HTTP authentication
2. Esquema de autenticação Basic
3. Spring Security
    1. Ecossistema de Usuario
    2. Ecossistema de Segurança
4. Testando no Postman

1. Conhecendo HTTP authentication

O IETF (*Internet Engineering Task Force*) tem como missão identificar e propor soluções a questões/problemas relacionados à utilização da Internet, além de propor padronização das tecnologias e protocolos envolvidos. O mesmo define a estrutura de autenticação HTTP que pode ser usada por um servidor para definir uma solicitação do cliente. O servidor responde ao cliente com uma mensagem do tipo `401` (Não autorizado) e fornece informações de como autorizar com um cabeçalho de resposta `WWW-Authenticate` contendo ao menos uma solicitação. Um cliente que deseja autenticar-se com um servidor pode fazer isso incluindo um campo de cabeçalho de solicitação `WWW-Authenticate` com as credenciais. No diagrama de seuquência abaixo pode se observar este relacionamento:



No caso de uma autorização "Basic" (como a mostrada na figura), a troca deve acontecer por meio de uma conexão HTTP (TLS) para ser segura. Se um servidor recebe credenciais válidas, mas que não são adequadas para ter acesso a um determinado recurso, o servidor responderá com o código de status `Forbidden` `403`. Ao contrário de `401` `Unauthorized` ou `407` `Proxy Authentication Required`, a autenticação é impossível para este usuário. O cabeçalho de solicitação `Authorization` contém as credenciais para autenticar um agente de usuário com um servidor. Aqui o tipo é novamente necessário, seguido pelas credenciais, que podem ser codificadas ou criptografadas dependendo do esquema de autenticação usado. No caso acima foi utilizado o esquema de autenticação Basic que sera explicado abaixo.

2. Esquema de autenticação Basic

A estrutura geral de autenticação HTTP é usado por vários esquemas de autenticação. Os esquemas podem divergir na força da segurança e na disponibilidade do software cliente ou servidor. O esquema mais comum de autenticação é o "Basic", mas existem outros esquemas oferecidos por serviços de hospedagem, como Amazon AWS, Google ou Microsoft. Os esquemas de autenticação mais comuns são:

Esquema de autenticação	Documentação
Basic	[RFC7617]
Bearer	[RFC6750]
Digest	[RFC7616]
HOBA	[RFC7486, Section 3]
Mutual	[RFC8120]
Negotiate	[RFC4559, Section 3]
OAuth	[RFC5849, Section 3.5.1]
SCRAM-SHA-1	[RFC7804]
SCRAM-SHA-256	[RFC7804]
vapid	[RFC 8292, Section 3]

ATENÇÃO: Para melhor compreeção no momento, focar apenas no necessario que seria o entendimento do formato Basic, que por sua vés é considerado o principal esquema para compriender os demais meios de autorização. Vale mencionar que para aprender os demais é necessario tempo e dedicação.

O esquema **Basic** segundo sua documentação, consiste em um conjunto de caracteres que posicionados após a palavra **"Basic "** no formato **"email:senha"** criptografado utilizando **Base64**, forma um ***Authorization*** para ser passado ao sistema. Abaixo veremos um exemplo de código para gerar a estrutura em Java:

Dependência:

```
<!-- Dependência para Codificação do Token -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
</dependency>
```

Exemplo de código em java:

```
import java.nio.charset.Charset;
import org.apache.commons.codec.binary.Base64;

/**
 * Método privado estatico responsavel por gerar o token em formato Basic
 *
 * ex.
 * estrutura: email@dominio.com:134652
 * estruturaBase64: ZW1haWxAZG9taW5pby5jb206MTM0NjUy
 * return: "Basic ZW1haWxAZG9taW5pby5jb206MTM0NjUy"
 *
 * @param email,    String format.
 * @param senha,    String format.
 * @return String
 * @author Generation
 * @since 1.0
 * @see Base64
 *
 */
private static String geradorBasicToken(String email, String senha) {
    String estrutura = email + ":" + senha;
    byte[] estruturaBase64 =
        Base64.encodeBase64(estrutura.getBytes(Charset.forName("US-ASCII")));
    return "Basic " + new String(estruturaBase64);
}
```

O resultado do código acima proporciona o retorno de um esquema de autenticação Basic respeitando as regras ja definidas pela IETF (*Internet Engineering Task Force*).

ALERTA DE BSM: Mantenha a Atenção aos Detalhes ao escrever o formato Basic, o mesmo é representado da palabra "*Basic* " com um espaço na frente + a criptografia de um conjunto de caracteres (*email:senha*) fornecidos ao se autenticar no sistema. Sua combinação irá deixar o seguinte resultado como exemplo: "*Basic ZW1haWxAZG9taW5pby5jb206MTM0NjUy*".

Com isso podemos utilizar no header de uma requisição ao servidor devidamente configurado para acessar seus recurços.

3. Spring Security

Analisando nossos projetos podemos perceber que nossa Api não possui nenhuma segurança, ou seja, qualquer pessoa pode acessar nossos end point e ter acesso aos nossos recursos. Precisamos entender que algumas aplicações contem informações como, dados pessoais, dados bancários, login e senha, precisamos garanti que nossa Api e estes dados estejam devidamente protegidos. E para isto podemos contar com a dependência do spring chamada **Spring Security**.

O Spring Segurity é um framework para Java que provê autenticação, autorização e diversas outras funcionalidades para aplicações corporativas. Iniciado em 2003 por Ben Alex o Spring Security é distribuído sob a licença Apache Licence. Ele oferece diversos recursos que permitem muitas práticas comuns de segurança serem declaradas ou configuradas de uma forma direta.

O esquema de autenticação que utilizaremos e o Basic, onde entraremos com o email e a senha de usuário através de um end point liberado, o Spring Security irá encriptar a senha e fazer uma consulta no nosso banco para saber se o usuário existe com a senha no nosso banco de dados, isto deverá ser feito através de uma camada de service. Se a consulta conseguir localizar o usuário e a senha, o Spring security devolverá como resposta um Authorization com o prefixo Basic + token. Este token ficara registrado na nossa aplicação na camada de Security, e apenas por meio desta chave o usuário poderá consumir a Api.

Para melhor compreenção deste sistema, é necessario dividilo em 2 ecosistemas um de Usuario e outro de Segurança, mas antes de começar é importante adcionar as depêndencias pricipais para a implementação:

Dependência spring security

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

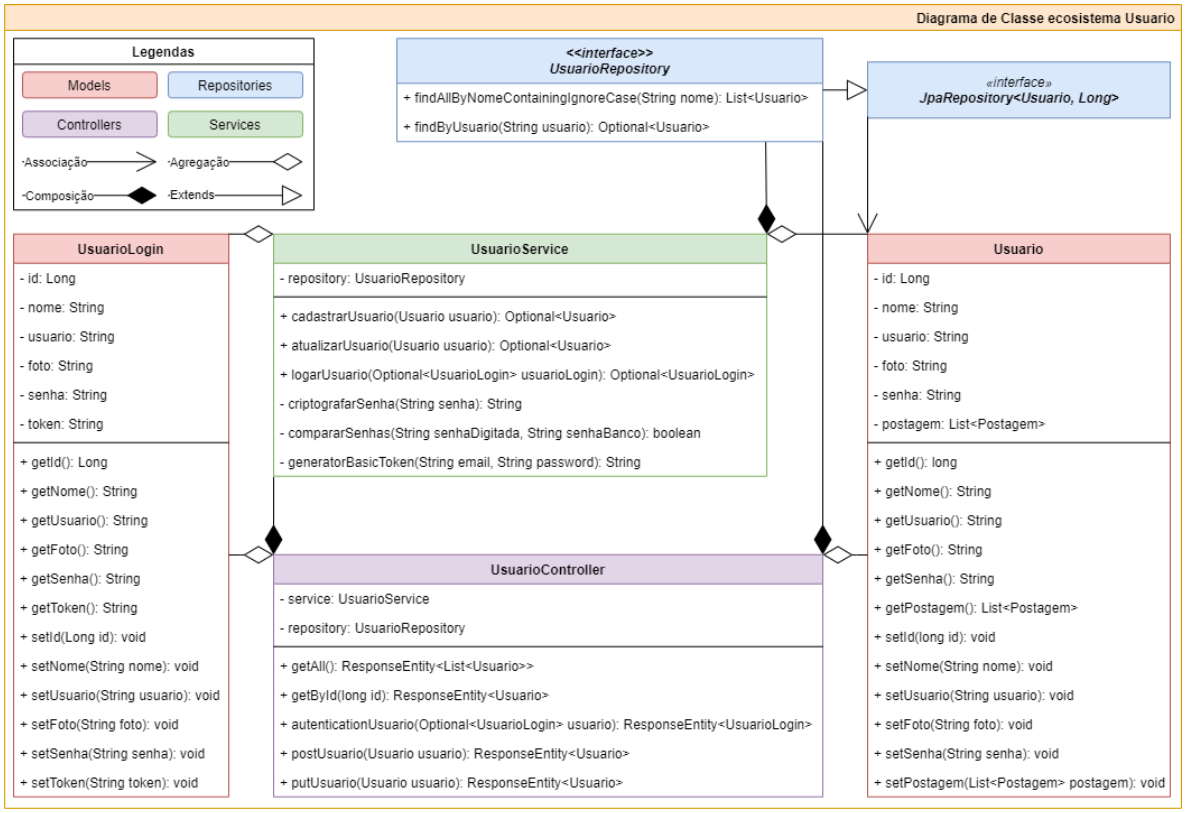
Esta dependência é responsavel por carregar todo o ambiente de segurança para recursos fornecidos na aplicação spring. Ao inserir ela a aplicação esta completamente protegida com uma configuração ja definida por padrão. Para poder acessar é necessario implementar uma classe de configuração e sobrescrever alguns métodos. Este procedimento sera visto neste documento.

Dependência commons codec

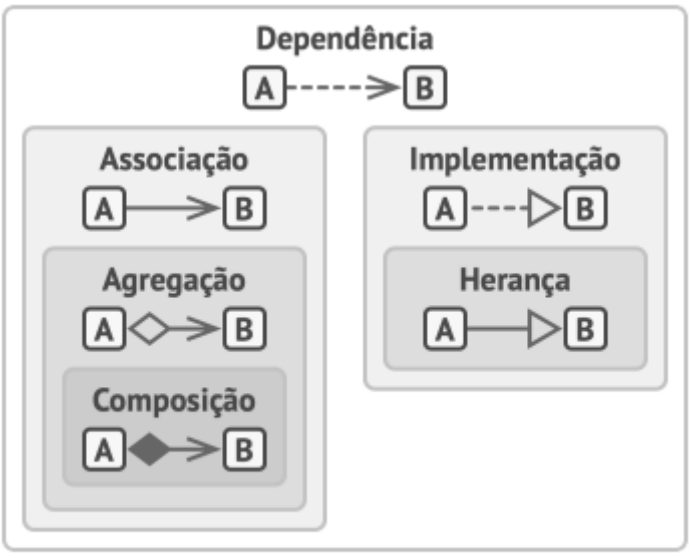
```
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
</dependency>
```

Esta dependência é responsavel por trazer bibliotécas de criptografica para o token.

3.1 Ecossistema de Usuario



No diagrama de classe acima, é possivel visualizar por completo o ecossistema de Usuario, que representa uma das principáis implementações para que o sistema de autenticação baseado em detalhes do usuario(UserDetail) funcione. A cor de cada uma das classe representa o pacote aonde deve ser construida segundo informações da Legênda. Para melhor compreenção das relaçoes entre classes veja uma breve definição sobre, relações entre objetos:



Relações entre objetos e classes: da mais fraca a mais forte.

Relação	Descrição
Dependência	Classe A pode ser afetada por mudanças na classe B;
Associação	Objeto A sabe sobre objeto B. Classe A depende de B;
Agregação	Objeto A sabe sobre objeto B, e consiste de B. Classe A depende de B;
Composição	Objeto A sabe sobre objeto B, consiste de B, e gerencia o ciclo de vida de B. Classe A depende de B;
Implementação	Classe A define métodos declarados na interface B. objetos de A podem ser tratados como B. Classe A depende de B;
Herança	Classe A herda a interface e implementação da classe B mas pode estendê-la. Objetos de A podem ser tratados como B. Classe A depende de B.

Após compreender o diagrama de classe acima e aprender um pouco sobre UML segue a aplicação do código em cada uma de suas classe

ATENÇÃO: Aproveite para validar seu código do blog pessoal, o projeto abaixo serve como espelho para o projeto. Tenha bastante atenção aos detalhes, pois existem implementações de muita importancia no sistema. Caso surja duvida de algo não exite em buscar os Instrutores Generation para um orientação.

Implementação Classe Usuario:

A classe **Usuario** é uma representação de uma tabéla no banco. A mesma possui notações que fazem com que o sistema identifique que é uma "entidade" no banco de dados. A seguir veja sua implementação:

```
package com.generation.blogpessoal.model;

import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Entity
@Table(name = "tb_usuarios")
public class Usuario {
```

```java
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull(message = "O atributo Nome é Obrigatório!")
    private String nome;

    @Size(max = 5000,
          message = "O link da foto não pode ser maior do que 5000 caractéres")
    private String foto;

    @NotNull(message = "O atributo Usuário é Obrigatório!")
    @Email(message = "O atributo Usuário deve ser um email válido!")
    private String usuario;

    @NotBlank(message = "O atributo Senha é Obrigatório!")
    @Size(min = 8, message = "A Senha deve ter no mínimo 8 caracteres")
    private String senha;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.REMOVE)
    @JsonIgnoreProperties("usuario")
    private List<Postagem> postagem;

    /* Insira os Getters and Setters */
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getFoto() {
        return foto;
    }

    public void setFoto(String foto) {
        this.foto = foto;
    }

    public String getUsuario() {
        return usuario;
    }

    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }

    public String getSenha() {
```

```
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

    public List<Postagem> getPostagem() {
        return postagem;
    }

    public void setPostagem(List<Postagem> postagem) {
        this.postagem = postagem;
    }
}
```

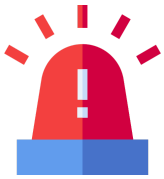
	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, o atributo senha definido acima, é importante que não contenha um atributo size que limite seu tamanho, pois isso pode provocar um erro 500, devido ao limite de caracteres que serão utilizados na criptografia. Deixe apenas com size minimo e não coloque maxímo.**
	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, o atributo de relacionamento definido como postagem esta com uma notação de cascade type do tipo REMOVE, se o seu projeto estiver com ALL é provavel que ao deletar uma postagem você delete também o usuario, pois este relacionamento tem uma dependência de pai e filho. Mantenha como REMOVE para que isso não aconteça**

Implementação Classe UsuarioLogin

A classe UsuarioLogin é responsavel por fazer com que o cliente ao tentar efetuar uma autenticação sómente forneça valores correspondentes e pertinentes ao Login. Essa classe também pode ser definida como uma DTO (*Data trasfer object*), é uma classe que é utilizada para transitar dados do sistema sem revelar sua classe de model para o cliente. A seguir veja sua implementação:

```
package com.generation.blogpessoal.model;

public class UsuarioLogin {

    private Long id;
    private String nome;
    private String usuario;
    private String foto;
    private String senha;
    private String token;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
```

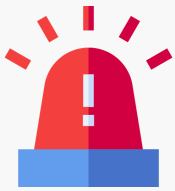
```
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getUsuario() {
        return usuario;
    }

    public String getFoto() {
        return foto;
    }

    public void setFoto(String foto) {
        this.foto = foto;
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }

    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }

    public String getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }
}
```

	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, não se esqueça de passar o id do usuario como atributo da classe, pois este atributo sera utilizado como crdencial pelo front-end para pegar o usuario pelo id.**
	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, não esquecer também do atributo token, pois o mesmo será passado no cabeçalho de todas as requisições que do front-end. Este atributo é fundamental para o funcionamento do consumo da api.**

Implementação Interface UsuarioRepository

A interface UsuarioRepositotory é responsavel por carregar os métodos CRUD para o sistema. Ao extender JpaRepository, através de herança podemos reduzir a quantidade de código escrito. Junto com esta interface é possivel customizar métodos de pesquisa especificos para algumas colunas, no caso abaixo temos uma pesquisa pela coluna usuario que será utilizada para fazer autenticação no sitema. A seguir veja sua implementação:

```
package com.generation.blogpessoal.repository;

import java.util.List;
import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.generation.blogpessoal.model.Usuario;

@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long>{

    public Optional<Usuario> findByUsuario(String usuario);

    public List<Usuario> findAllByNomeContainingIgnoreCase(String nome);
}
```



ALERTA DE BSM: Mantenha a Atenção aos Detalhes, tome muito cuidado pois se a escrita dos métodos findBy ou findAllBy estiver errada, pode saltar um erro no console do eclipse, tenha valide a docuemntação caso seja necessario.


[Documentação: *JpaRepository - Spring Doc*]


[Documentação: *JpaRepository - Java Doc*]

Implementação Classe UsuarioService

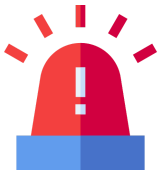
A classe UsuarioService é responsavel por manipular as regras de negócio de usuario no sistema. Esta classe deve ser notada com @Service, para que o spring identifique que é uma classe que serviços e carregue ela sempre que puder. Vale mencionar que alguns métodos definidos abaixo sãod e estrema inportância para o sistema. A seguir veja sua implementação:

```
package com.generation.blogpessoal.service;

import java.nio.charset.Charset;
import java.util.Optional;

import org.apache.commons.codec.binary.Base64;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;
```

```java
import org.springframework.web.server.ResponseStatusException;

import com.generation.blogpessoal.model.Usuario;
import com.generation.blogpessoal.model.UsuarioLogin;
import com.generation.blogpessoal.repository.UsuarioRepository;

@Service
public class UsuarioService {

    private @Autowired UsuarioRepository usuarioRepository;

    /**
     * Método utilizado para cadastrar um usuario no banco de dados, o mesmo é
     * responsavel por retornar vazio caso Usuario exista
     *
     * @param usuario
     * @return Optional<Usuario>
     * @since 1.0
     * @author Generation
     */
    public Optional<Usuario> cadastrarUsuario(Usuario usuario) {
        if (usuarioRepository.findByUsuario(usuario.getUsuario()).isPresent())
            throw new
            ResponseStatusException(HttpStatus.BAD_REQUEST,"Usuário já
existe!");
        usuario.setSenha(criptografarSenha(usuario.getSenha()));
        return Optional.of(usuarioRepository.save(usuario));
    }

    /**
     * Metodo utilizado para atualizar um usuario fornecido pelo FRONT, O mesmo
     * retorna um Optional com entidade Usuario dentro e senha criptografada.
Caso
     * falho retorna um Optional.empty()
     *
     * @param usuario
     * @return Optional<Usuario>
     * @since 1.0
     * @author Generation
     */
    public Optional<Usuario> atualizarUsuario(Usuario usuario) {
        if (usuarioRepository.findById(usuario.getId()).isPresent()) {
            Optional<Usuario> buscaUsuario =
                usuarioRepository.findByUsuario(usuario.getUsuario());
            if (buscaUsuario.isPresent()) {
                if (buscaUsuario.get().getId() != usuario.getId())
                    throw new
                    ResponseStatusException(HttpStatus.BAD_REQUEST,"Usuário já
existe!");
            }
            usuario.setSenha(criptografarSenha(usuario.getSenha()));
            return Optional.of(usuarioRepository.save(usuario));
        }
        throw new
            ResponseStatusException(HttpStatus.NOT_FOUND,"Usuário não
encontrado!");
    }
```

```
    /**
     * Metodo utilizado para pegar credenciais do usuario com Tokem (Formato
Basic),
     * este método sera utilizado para retornar ao front o token utilizado para
ter
     * acesso aos dados do usuario e mantelo logado no sistema
     *
     * @param usuarioLogin
     * @return Optional<UsuarioLogin>
     * @since 1.0
     * @author Generation
     */
    public Optional<UsuarioLogin> logarUsuario(Optional<UsuarioLogin>
usuarioLogin) {
        Optional<Usuario> usuario =
            usuarioRepository.findByUsuario(usuarioLogin.get().getUsuario());

        if (usuario.isPresent()) {
            if (compararSenhas(usuarioLogin.get().getSenha(),
usuario.get().getSenha())){
                usuarioLogin.get().setId(usuario.get().getId());
                usuarioLogin.get().setNome(usuario.get().getNome());
                usuarioLogin.get().setFoto(usuario.get().getFoto());
                usuarioLogin.get().setSenha(usuario.get().getSenha());
                usuarioLogin.get().setToken(
                    generatorBasicToken(

 usuarioLogin.get().getUsuario(),usuarioLogin.get().getSenha()));
                return usuarioLogin;
            }
        }
        throw new ResponseStatusException(
                HttpStatus.UNAUTHORIZED, "Usuário ou senha inválidos!", null);
    }

    /**
     * Método privado responsavel por gerar o criptografia para senha
     *
     * @param senha
     * @return String
     * @author Generation
     * @since 1.0
     * @see BCryptPasswordEncoder
     */
    private String criptografarSenha(String senha) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        String senhaEncoder = encoder.encode(senha);
        return senhaEncoder;
    }

    /**
     * Método privado responsavel por comparar senha de entrada com senha no
banco
     *
     * @param senhaDigitada
     * @param senhaBanco
     * @return boolean
     * @author Generation
```

```
     * @since 1.0
     * @see BCryptPasswordEncoder
     */
    private boolean compararSenhas(String senhaDigitada, String senhaBanco) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        return encoder.matches(senhaDigitada, senhaBanco);
    }

    /**
     * Método privado responsavel por gerar o token em formato Basic
     *
     * ex.
     * estrutura: email@dominio.com:134652
     * estruturaBase64: ZW1haWxAZG9taW5pby5jb20MTM0NjUy
     * return: "Basic ZW1haWxAZG9taW5pby5jb20MTM0NjUy"
     *
     * @param email
     * @param password
     * @return String
     * @author Generation
     * @since 1.0
     * @see Base64
     */
    private String generatorBasicToken(String email, String password) {
        String structure = email + ":" + password;
        byte[] structureBase64 =
            Base64.encodeBase64(structure.getBytes(Charset.forName("US-
ASCII")));
        return "Basic " + new String(structureBase64);
    }
}
```

	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, o cadastro de um novo usuario no sistema necessita ser validado no banco. Caso o usuario ja exista não permitir que seja criado, pois um usuario duplicado no sistema ocasionara um erro 500.**
	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, ao atualizar um usuario é importante que seja validado novamente a criptografia da senha. Caso não seja validado ocasionara um problema ao tentar pegar as credenciais pelo front-end da aplicação.**
	**ALERTA DE BSM: Mantenha a Atenção aos Detalhes, ao utilizar o método para logarUsuario, se atentar de que seja passado todos os parametros do usuarioLogin, pois o mesmo sera utilizado pelo front-end da aplicação.**

 [Documentação: *Base64 - Java Doc Commons*]

 [Documentação: *BCryptPasswordEncoder - Java Doc Spring*]

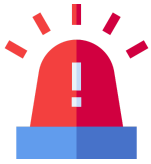
Implementação Classe UsuarioController

A classe UsuarioCotnroller, é responsavel por fornecer o acessos aos recursos do sistema. Uma de suas funcionalidades abaixo é promover o CRUD do usuario. Alen de permitir total manipulação do usuario, esta consumindo serviços para cadastrar usuario e pegar sua autenticação

```
package com.generation.blogpessoal.controller;

import java.util.List;
import java.util.Optional;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.generation.blogpessoal.model.Usuario;
import com.generation.blogpessoal.model.UsuarioLogin;
import com.generation.blogpessoal.repository.UsuarioRepository;
import com.generation.blogpessoal.service.UsuarioService;

@RestController
@RequestMapping("/usuarios")
@CrossOrigin(origins = "*", allowedHeaders = "*")
public class UsuarioController {

    private @Autowired UsuarioService service;
    private @Autowired UsuarioRepository repository;

    @GetMapping("/all")
    public ResponseEntity <List<Usuario>> getAll() {
        return ResponseEntity.ok(repository.findAll());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Usuario> getById(@PathVariable Long id) {
        return repository.findById(id)
            .map(resp -> ResponseEntity.ok(resp))
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping("/logar")
    public ResponseEntity<UsuarioLogin> autenticationUsuario(
        @RequestBody Optional<UsuarioLogin> usuario) {
        return service.logarUsuario(usuario)
            .map(resp -> ResponseEntity.ok(resp))
            .orElse(ResponseEntity.status(HttpStatus.UNAUTHORIZED).build());
```

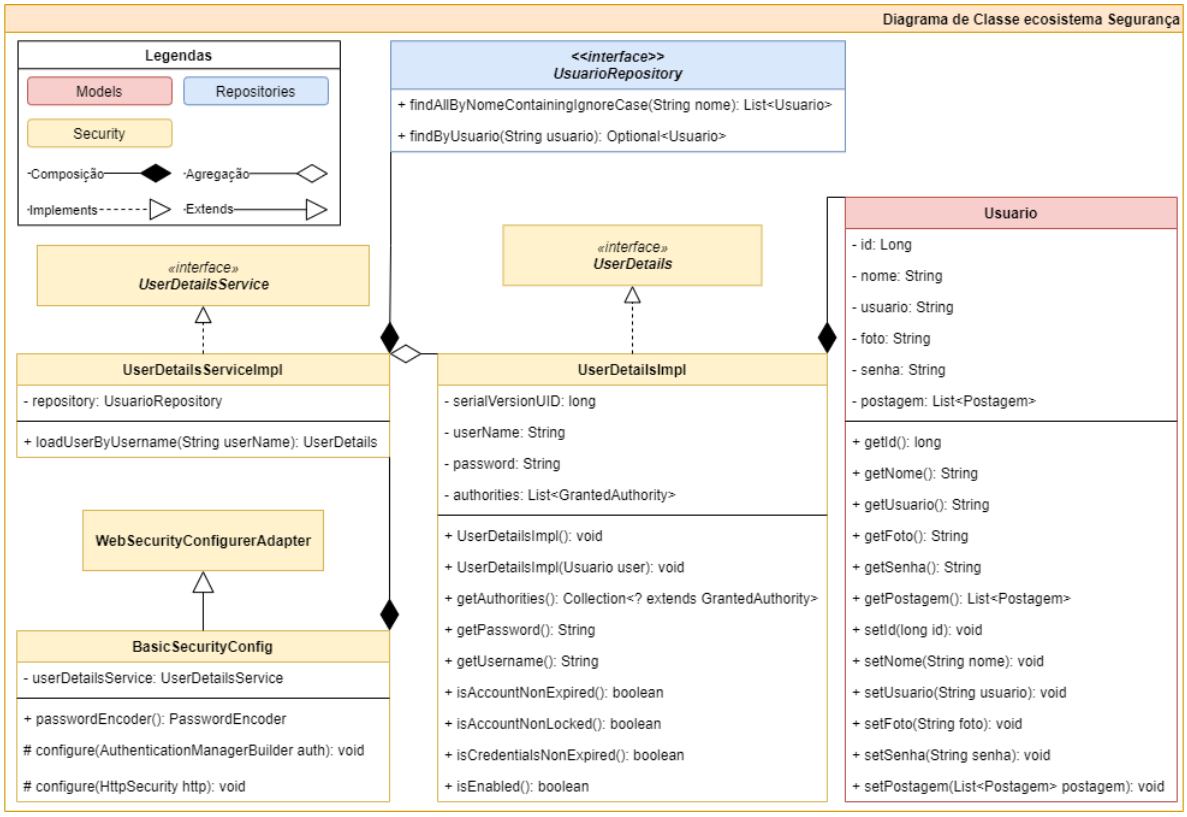
```
    }

    @PostMapping("/cadastrar")
    public ResponseEntity<Usuario> postUsuario(@Valid @RequestBody Usuario
usuario) {
        return service.cadastrarUsuario(usuario)
            .map(resp -> ResponseEntity.status(HttpStatus.CREATED).body(resp))
            .orElse(ResponseEntity.status(HttpStatus.BAD_REQUEST).build());
    }

    @PutMapping("/atualizar")
    public ResponseEntity<Usuario> putUsuario(@Valid @RequestBody Usuario
usuario){
        return service.atualizarUsuario(usuario)
            .map(resp -> ResponseEntity.status(HttpStatus.OK).body(resp))
            .orElse(ResponseEntity.status(HttpStatus.BAD_REQUEST).build());
    }
}
```

ALERTA DE BSM: Mantenha a Atenção aos Detalhes, nos mmétodos postUsuario e putUsuario não esquecer da notação @Valid. Caso não for fornecida é ao efetuar testes e fornecer parametros invalidos o servidor retornara status 500. Caso contrario fornece status 400.

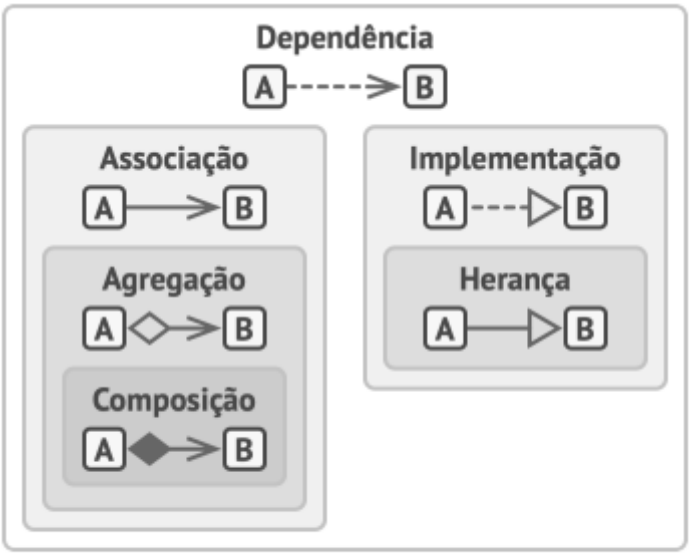


ALERTA DE BSM: Mantenha a Atenção aos Detalhes, não esquecer da notação @CrossOrigin(origins = "", *allowedHeaders* = "").

3.2 Ecosistema de Segurança



No diagrama de classe acima, é possivel visualizar por completo o ecossistema de Segurança, que representa uma das principáis implementações de segurança do sistema. Nesta implementação é possivel realizar a autenticação do usuario se baseando em uma base de dados. Para isso é necessario passar as informações necessarias para que o spring valide a permição do usuario no sistema. O ecosistema de segurança principal consiste de uma classe de configuração (BasicSecurityConfig), uma de serviço (UserDetailsService) e uma model (UserDetailsImpl). A cor de cada uma das classes representa o pacote onde deve ser construida segundo informações da legenda. Para este diagrama repare que a model de Usuario e o Repositorio de Usuario ja foram criada anteriormente. Para melhor compreenção das relaçoes entre classes veja uma breve definição sobre, relações entre objetos:



Relações entre objetos e classes: da mais fraca a mais forte.

Relação	Descrição
Dependência	Classe A pode ser afetada por mudanças na classe B;
Associação	Objeto A sabe sobre objeto B. Classe A depende de B;
Agregação	Objeto A sabe sobre objeto B, e consiste de B. Classe A depende de B;
Composição	Objeto A sabe sobre objeto B, consiste de B, e gerencia o ciclo de vida de B. Classe A depende de B;
Implementação	Classe A define métodos declarados na interface B. objetos de A podem ser tratados como B. Classe A depende de B;
Herança	Classe A herda a interface e implementação da classe B mas pode estendê-la. Objetos de A podem ser tratados como B. Classe A depende de B.

Após compreender o diagrama de classe acima e aprender um pouco sobre UML segue a aplicação do código em cada uma de suas classe

ATENÇÃO: Aproveite para validar seu código do blog pessoal, o projeto abaixo serve como espelho para o projeto. Tenha bastante atenção aos detalhes, pois existem implementações de muita importancia no sistema. Caso surja duvida de algo não exite em buscar os Instrutores Generation para um orientação.

Implementação Classe UserDetailsImpl

Esta classe implementa a interface UserDetails, que tem como principal funcionalidade fornecer informações básicas do usuário para o spring. Ao criar esta classe, será permitido que um serviço preencha a mesma e retorne apenas o necessario para que o spring possa validar o usuario que esta tentando acessar o sistema. Vejamos abaixo sua implementação:

```
package com.generation.blogpessoal.security;

import java.util.Collection;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.generation.blogpessoal.model.Usuario;

public class UserDetailsImpl implements UserDetails{

    private static final long serialVersionUID =1L;
    private String userName;
    private String password;
    private List<GrantedAuthority> authorities;

    public UserDetailsImpl (Usuario user){
        this.userName = user.getUsuario();
        this.password = user.getSenha();
    }

    public UserDetailsImpl (){}

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return userName;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
```

```
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

ALERTA DE BSM: Mantenha a Atenção aos Detalhes, é necessario ter um construtor adaptado para receber atributos que serão utilizados para logar no sistema. Os atributos em questão são usuario e senha, mas poderia ser também email e senha. Tudo irá depender de como a model de Usuario esta elaborada.

[Documentação: *UserDetails - Java Doc Spring*]

Implementação da Classe UserDetailsServiceImpl

Esta classe é uma implementação de UserDetailsService, interfasse responsavel por fornecer a estrutura de um método utilizado pelo spring security para validar a existencia de um usuario no banco de dados e retornar um UserDetails. Com os dados fornecidos ao sistema, será possivel autenticar um usuario baseandosse na sua existencia no banco. Vale lembrar que para isso é necessario que ao salvar o usuario, sua senha esteja criptografada, utilizadno uma bibliotéca de criptografia valida para o sistema de segurança. Vejamos abaixo sua implementação:

```
package com.generation.blogpessoal.security;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.generation.blogpessoal.model.Usuario;
import com.generation.blogpessoal.repository.UsuarioRepository;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    private @Autowired UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Optional<Usuario> optional = repository.findByUsuario(username);
        if (optional.isPresent()) {
            return new UserDetailsImpl(optional.get());
```

```
        } else {
            throw new UsernameNotFoundException(userName + " not found.");
        }
    }
}
```

ALERTA DE BSM: Mantenha a Atenção aos Detalhes, ao criar este serviço não esquecer de colocar a notação @Service. Assim notado o spring entenderá que a classe é um serviço e fará o carregamento com prioridade para esta classe.

 [Documentação: *UserDetailsService - Java Doc Spring*]

Implementação da Classe BasicSecurityConfig

Esta classe é utilizada para sobreescrver a configuração já predefinida pelo spring para segurança. A mesma é notada de **@EnableWebSecurity** que habilita e sobrescreve métodos que iram ajudar com que as regras de segurança sejam redefinidas para melhor definição de seu negocio.

```
package com.generation.blogpessoal.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpMethod;
import
org.springframework.security.config.annotation.authentication.builders.Authentic
ationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecuri
ty;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConf
igurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@EnableWebSecurity
public class BasicSecurityConfig extends WebSecurityConfigurerAdapter {

    private @Autowired UserDetailsService userDetailsService;

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
```

```
            auth.userDetailsService(userDetailsService);

            auth.inMemoryAuthentication()
                .withUser("root")
                .password(passwordEncoder().encode("root"))
                .authorities("ROLE_ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/usuarios/logar").permitAll()
                .antMatchers("/usuarios/cadastrar").permitAll()
                .antMatchers(HttpMethod.OPTIONS).permitAll()
                .anyRequest().authenticated()
            .and().httpBasic()
            .and().sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and().cors()
            .and().csrf().disable();
    }
}
```

No esquema acima ao notar com **@Bean** o método ***passwordEncoder()*** , informamos ao spring security que a aplicação esta baseada em alguma criptografia qeu implement *PasswordEncoder*. Para esta aplicação estamos utilizando o encoder *BCryptPasswordEncoder()*. Vale resaltar que esta criptografia esta sendo esperada para analizar a senha que estamos guardadno no Banco de dados.

Quando sobreescrevemos o método ***configure(AuthenticationManagerBuilder auth)***, estamos informando ao spring que a configuração de authenticação sera definida no formato passado em sua implementação. No caso acima este método esta fornecendo duas maneiras de autenticação.

Com o método:

`.userDetailsService(userDetailsService)`, sera possivel utilizar um serviço para pesquisar a credencial do usuario no banco de dados

Com o método:

`.inMemoryAuthentication()`, será possivel validar com um usuario em memória, no caso definido ali mesmo como **root** e senha **root**.

Para mais detalhes sobre esta definição, o mais correto é olhar sua fonte no link abaixo:

 **Documentação: *AuthenticationManagerBuilder - Java Doc Spring***

Quando sobreescrevemos o método ***configure(HttpSecurity http)***, estamos informando ao spring que a configuração por padrão definida sera alterada. Nesta configuração é possivel customizar de diferentes maneiras sua aplicação.

Com a confuguração:

`.authorizeRequests()`, podemos definir qual rota sera possivel acessar no sistema sem precisar de altenticação alguma. No caso acima foi definido que as rotas de de logar e cadastrar seja autorizada para todos os usuarios, para todas as demasi rotas, é necessario fazer uma autenticação.

Com as confugurações:

`.antMatchers("/usuarios/logar").permitAll()`
`.antMatchers("/usuarios/cadastrar").permitAll()`, é possivel passar qual rota será possivel acessar sem ter uma autenticação definida.

Com o configuração:

`anyRequest().authenticated()`, é passado ao sistema que todas as demais rotas que não estiverem especificadas no escopo do *authorizeRequests()*, deveram ser autenticadas.

Com a configuração:

`.httpBasic()`, estamos informando ao sistema que o servidor ira receber requisições que dever ter o esquema Basica de autenticação. Desta maneira habilitamos o servidor para que as requisições tenham um formato de autenticação.

Com a configuração:
`.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)`, é possivel definir que nosso sistema não guardara sessões para o cliente.

Com a configuração:

`.cors()`, estamos liberando o acesso de outras origens, desta maneira nossas APIs seram possivel ser acessadas de outros dominios.

Com a configuração:

`.csrf().disable()`, estamos desabilitando uma proteção que vem ativa contra ataque de CSRF.

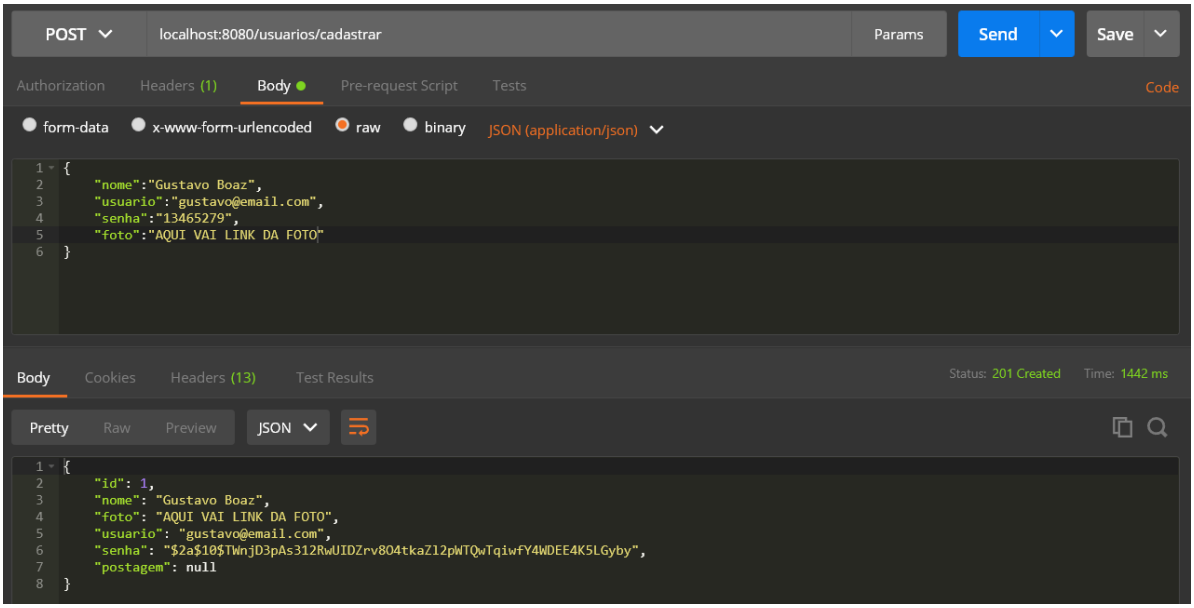
Para mais detalhes sobre esta definição, o mais correto é olhar sua fonte no link abaixo:


Documentação: *HttpSecurity - Java Doc Spring*

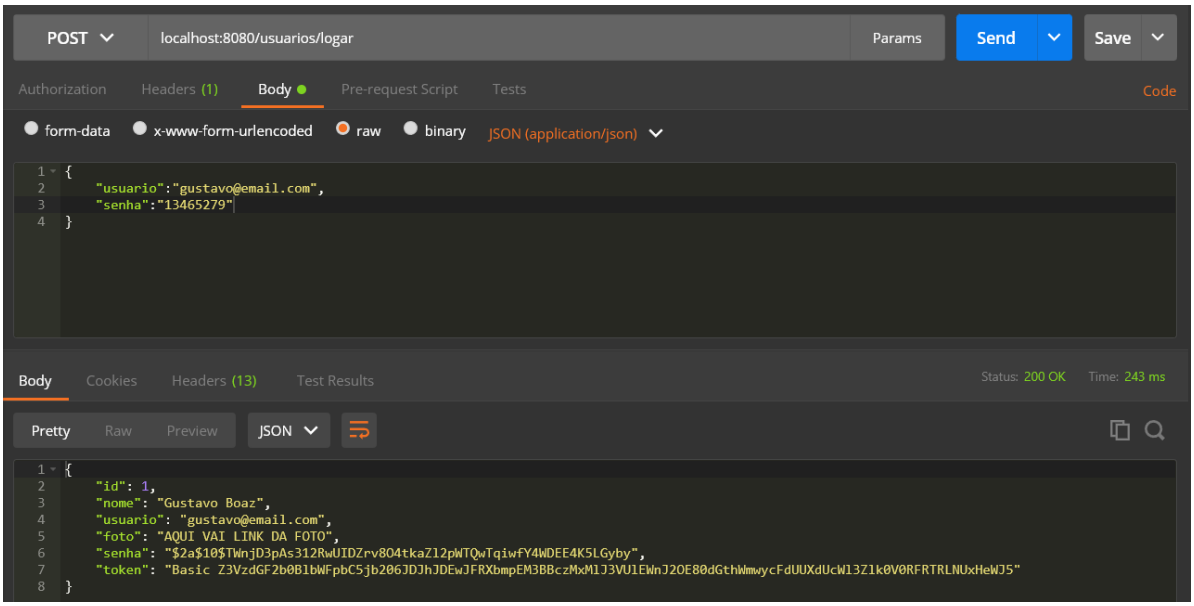
4. Testando no Postman

Para testar com Postman basta seguir os passos abaixo:

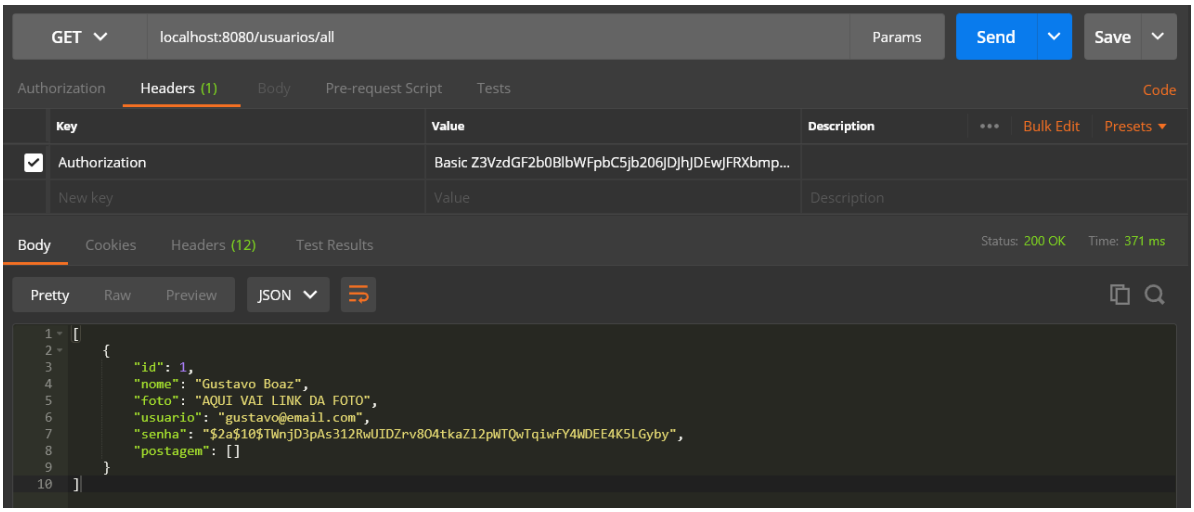
4.1 Cadastre novo usuario:



4.2 Fazer Login com o usuario:



4.3 Busque com o Token no Autorization:



4.4 Busque com a autenticação em memoria: