O **Spring Initializr** é uma ferramenta online que facilita a configuração inicial de um projeto Spring Boot. Siga os passos abaixo para configurar seu projeto Spring Boot corretamente:

Passos para Configurar pelo Spring Initializr:

1. **Acesse o Spring Initializr**:
   - URL: https://start.spring.io/

2. **Configure as opções principais**:
   - **Project**: Escolha entre **Maven** (para projetos baseados em Maven) ou **Gradle**.
   - **Language**: Selecione **Java**.
   - **Spring Boot Version**: Escolha a versão do Spring Boot (recomendo usar a versão mais recente estável).
   - **Project Metadata**:
     - **Group**: O grupo ao qual seu projeto pertence, por exemplo: `com.exemplo`.
     - **Artifact**: Nome do seu artefato/projeto, por exemplo: `sistema-atas`.
     - **Name**: Nome do seu projeto (igual ao artifact).
     - **Description**: Descreva o projeto, por exemplo: `Sistema de emissão de atas de reunião`.
     - **Package name**: Será gerado automaticamente, mas você pode ajustá-lo se necessário.
     - **Packaging**: Selecione **jar** (o padrão).
     - **Java version**: Selecione a versão do Java com a qual você está trabalhando, por exemplo: **17** (ou a versão mais recente instalada no seu sistema).

3. **Adicionar Dependências**:
   Agora, você deve adicionar as dependências necessárias ao projeto. Clique no botão **"Add Dependencies"** e busque pelas seguintes dependências:

   - **Spring Web**: Para criar APIs RESTful e servidores web.
   - **Spring Data JPA**: Para integrar o banco de dados com JPA (Java Persistence API).
   - **MySQL Driver** (ou outro banco de dados que você usar): Para se conectar ao banco de dados MySQL (ou outra base de dados que você está usando).
   - **Lombok** (opcional): Para reduzir o código boilerplate, como getters, setters, e construtores.

   Se estiver planejando usar outra base de dados (como PostgreSQL), substitua o driver de MySQL pelo respectivo driver do banco que utilizará.

4. **Gerar o projeto**:
   Depois de configurar tudo, clique no botão **Generate**. Isso fará o download de um arquivo **.zip** contendo o esqueleto do projeto configurado.

5. **Importar o Projeto na IDE**:
   - Extraia o arquivo ZIP gerado.

- Abra sua IDE favorita (IntelliJ, Eclipse, ou VS Code) e importe o projeto como um projeto **Maven** ou **Gradle**, dependendo da escolha feita no Spring Initializr.
- Após a importação, a IDE irá automaticamente baixar as dependências especificadas no `pom.xml` (para Maven) ou `build.gradle` (para Gradle).

Configuração de Banco de Dados
Após importar o projeto, configure as credenciais de conexão com o banco de dados. Abra o arquivo `application.properties` (ou `application.yml`) e insira as informações necessárias.

Exemplo de configuração com MySQL:

```properties
# Configuração do banco de dados MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/sistema_atas
spring.datasource.username=seu_usuario
spring.datasource.password=sua_senha

# JPA configuração
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Dependências Adicionais:
Se você precisar adicionar dependências no futuro, edite o arquivo **`pom.xml`** (para projetos Maven) e adicione dependências lá, como no exemplo:

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Depois de configurar, você pode rodar o projeto Spring Boot com o comando:

```bash
./mvnw spring-boot:run
```

Ou diretamente da sua IDE.

Com o Spring Initializr, o processo de configuração se torna muito simples, e a partir daí você pode começar a desenvolver suas funcionalidades específicas.

Para transformar essas classes em um sistema Spring Boot, precisamos considerar as seguintes dependências e ajustes:

1. **Anotações Spring Boot**
- Usaremos anotações do **Spring Data JPA** para o mapeamento das entidades no banco de dados.
- Cada classe será mapeada para uma tabela do banco, e adicionaremos repositórios para gerenciar as operações de CRUD.

2. **Dependências Spring Boot**
Para o uso de Spring Boot com **Spring Data JPA** e **MySQL** (ou outro banco de dados), você precisará adicionar as seguintes dependências no arquivo `pom.xml`:

```xml
<dependencies>
    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- MySQL Driver (se usar MySQL) -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
```

```
        </dependency>

        <!-- Lombok (opcional, para reduzir código boilerplate como getters/setters) -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <scope>provided</scope>
        </dependency>

        <!-- Spring Boot DevTools (opcional, para desenvolvimento) -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>

        <!-- Spring Boot Test -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    ```

3. **Correção das Classes com Spring Boot (Entidades e Repositórios)**

Classe `Ata` com JPA
```java
import jakarta.persistence.*;
import java.util.List;

@Entity
public class Ata {

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

    private String titulo;
    private String dataEmissao;
    private String inicio;
    private String termino;
    private String pauta;
```

```
    @Column(length = 3000)
    private String descricao;

    @ElementCollection
    private List<String> palavrasChave;

    private String tipo; // "publica" ou "privada"
    private String status; // "Emissão", "Revisão", "Conclusão", "Emitida"

    // Getters, Setters, e Construtores

    public Ata() {}

    public Ata(String titulo, String dataEmissao, String inicio, String termino, String pauta, String
descricao, List<String> palavrasChave, String tipo, String status) {
        this.titulo = titulo;
        this.dataEmissao = dataEmissao;
        this.inicio = inicio;
        this.termino = termino;
        this.pauta = pauta;
        this.descricao = descricao;
        this.palavrasChave = palavrasChave;
        this.tipo = tipo;
        this.status = status;
    }

    // Métodos como enviarRevisao() e concluir(), com a lógica de negócio
}
```

Classe `Setor` com JPA
```java
import jakarta.persistence.*;

@Entity
public class Setor {

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

    private String nome;
```

```
    // Construtores, Getters e Setters

    public Setor() {}

    public Setor(String nome) {
        this.nome = nome;
    }

    // Método selecionar() para lógica de negócio
}
```

Classe `Funcionario` com JPA
```java
import jakarta.persistence.*;

@Entity
public class Funcionario {

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

    private String nome;
    private String matricula;
    private char sexo; // 'm' ou 'f'
    private String nascimento;
    private String email;

    // Construtores, Getters e Setters

    public Funcionario() {}

    public Funcionario(String nome, String matricula, char sexo, String nascimento, String email)
{
        this.nome = nome;
        this.matricula = matricula;
        this.sexo = sexo;
        this.nascimento = nascimento;
        this.email = email;
    }

    // Método selecionar() para lógica de negócio
}
```

```

#### **Classe `ParticipanteExterno` com JPA**
```java
import jakarta.persistence.*;

@Entity
public class ParticipanteExterno {

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

    private String nome;
    private String empresa;
    private String email;

    // Construtores, Getters e Setters

    public ParticipanteExterno() {}

    public ParticipanteExterno(String nome, String empresa, String email) {
        this.nome = nome;
        this.empresa = empresa;
        this.email = email;
    }

    // Método manter() e selecionar() para lógica de negócio
}
```

Classe `Sugestao` com JPA
```java
import jakarta.persistence.*;

@Entity
public class Sugestao {

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

    private String data;
```
```

```
    @Column(length = 500)
    private String descricao;

    // Construtores, Getters e Setters

    public Sugestao() {}

    public Sugestao(String data, String descricao) {
        this.data = data;
        this.descricao = descricao;
    }

    // Métodos como selecionar() e emitir() com a lógica de negócio
}
```

4. **Repositórios (Interfaces)**
Cada classe precisa de um **Repository** para realizar as operações de banco de dados (CRUD). O Spring Data JPA facilita isso com a criação de interfaces que estendem `JpaRepository`.

Exemplo de Repositório para `Ata`
```java
import org.springframework.data.jpa.repository.JpaRepository;

public interface AtaRepository extends JpaRepository<Ata, Long> {
    // Métodos de consulta personalizados, se necessário
}
```

Repita esse padrão para as outras classes, como `Setor`, `Funcionario`, `ParticipanteExterno`, e `Sugestao`.

5. **Configurações de Banco de Dados**
No arquivo `application.properties` ou `application.yml`, configure as credenciais do banco de dados:

Exemplo (`application.properties`)
```properties
spring.datasource.url=jdbc:mysql://localhost:3306/sistema_atas
spring.datasource.username=root
spring.datasource.password=senha
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

```

### 6. **Controladores (Opcional para API RESTful)**
Para expor os endpoints da aplicação, você pode criar controladores com as anotações do Spring MVC.

#### Exemplo de Controlador para `Ata`
```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/atas")
public class AtaController {

    @Autowired
    private AtaRepository ataRepository;

    @GetMapping
    public List<Ata> listarAtas() {
        return ataRepository.findAll();
    }

    @PostMapping
    public Ata criarAta(@RequestBody Ata ata) {
        return ataRepository.save(ata);
    }

    // Métodos adicionais para atualizar, deletar, etc.
}
```

Com isso, a estrutura de classes e dependências para o Spring Boot está pronta. Se precisar de mais alguma parte ou tiver dúvidas na implementação de algum componente, estou à disposição!

Aqui estão as classes Java baseadas no documento de modelagem. Cada classe inclui seus atributos e métodos principais, de acordo com as especificações do documento.
```

1. **Classe `Ata`**
```java
public class Ata {
    private String titulo;
    private String dataEmissao;
    private String inicio;
    private String termino;
    private String pauta;
    private String descricao;
    private String[] palavrasChave; // Mínimo de 1, máximo de 5
    private String tipo; // "publica" ou "privada"
    private String status; // "Emissão", "Revisão", "Conclusão", "Emitida"

    // Construtor
    public Ata(String titulo, String dataEmissao, String inicio, String termino, String pauta, String
descricao, String[] palavrasChave, String tipo, String status) {
        this.titulo = titulo;
        this.dataEmissao = dataEmissao;
        this.inicio = inicio;
        this.termino = termino;
        this.pauta = pauta;
        this.descricao = descricao;
        this.palavrasChave = palavrasChave;
        this.tipo = tipo;
        this.status = status;
    }

    // Métodos
    public void enviarRevisao() {
        this.status = "Revisão";
        // Lógica para envio de revisão
    }

    public void concluir() {
        this.status = "Concluída";
        // Lógica para conclusão
    }

    public void selecionar() {
        // Lógica para selecionar a ata
    }

    public void manter() {
        // Lógica para incluir, alterar e excluir ata
```

```
    }

    // Getters e Setters (gerar automaticamente para cada atributo, se necessário)
}
```

2. **Classe `Setor`**
```java
public class Setor {
    private String nome;

    // Construtor
    public Setor(String nome) {
        this.nome = nome;
    }

    // Método
    public void selecionar() {
        // Lógica para selecionar o setor
    }

    // Getter e Setter
    public String getNome() {
        return nome;
    }

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

3. **Classe `Funcionario`**
```java
public class Funcionario {
    private String nome;
    private String matricula;
    private char sexo; // 'm' ou 'f'
    private String nascimento;
    private String email;

    // Construtor
    public Funcionario(String nome, String matricula, char sexo, String nascimento, String email)
{
```

```
    this.nome = nome;
    this.matricula = matricula;
    this.sexo = sexo;
    this.nascimento = nascimento;
    this.email = email;
}

// Método
public void selecionar() {
    // Lógica para selecionar funcionário
}

// Getters e Setters
public String getNome() {
    return nome;
}

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

public String getMatricula() {
    return matricula;
}

public void setMatricula(String matricula) {
    this.matricula = matricula;
}

public char getSexo() {
    return sexo;
}

public void setSexo(char sexo) {
    this.sexo = sexo;
}

public String getNascimento() {
    return nascimento;
}

public void setNascimento(String nascimento) {
    this.nascimento = nascimento;
}
```

```
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

4. **Classe `ParticipanteExterno`**
```java
public class ParticipanteExterno {
    private String nome;
    private String empresa;
    private String email;

    // Construtor
    public ParticipanteExterno(String nome, String empresa, String email) {
        this.nome = nome;
        this.empresa = empresa;
        this.email = email;
    }

    // Métodos
    public void selecionar() {
        // Lógica para selecionar participante externo
    }

    public void manter() {
        // Lógica para incluir, alterar ou excluir participante externo
    }

    // Getters e Setters
    public String getNome() {
        return nome;
    }

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

    public String getEmpresa() {
```

```
        return empresa;
    }

    public void setEmpresa(String empresa) {
        this.empresa = empresa;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

5. **Classe `Sugestao`**
```java
public class Sugestao {
    private String data;
    private String descricao;

    // Construtor
    public Sugestao(String data, String descricao) {
        this.data = data;
        this.descricao = descricao;
    }

    // Métodos
    public void selecionar() {
        // Lógica para selecionar sugestão
    }

    public void emitir() {
        // Lógica para emitir sugestão
    }

    // Getters e Setters
    public String getData() {
        return data;
    }

    public void setData(String data) {
```

```
        this.data = data;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
}
```

Essas classes representam a base do sistema. Podemos agora prosseguir com a implementação de métodos específicos, como a lógica de interface gráfica, manipulação de banco de dados ou integração com o sistema de notificações por e-mail. Qual parte você gostaria de desenvolver primeiro?