

# Guida per la creazione di un backend

---

## Indice

---

1. Introduzione
2. Creazione di un progetto Spring Boot
3. Configurazione di un database
  - Configurazione di un database HSQLDB
  - Configurazione di un database Postgres
4. Creazione di un'entità, un repository, un servizio e un controller
  - Creazione di un'entità
  - Creazione di un listener per le entità (opzionale)
  - Creazione di un DTO (Data Transfer Object)
  - Creazione di un repository (DAO)
  - Creazione di un servizio
    - Creazione di un'interfaccia per il servizio
    - Creazione dell'implementazione del servizio
  - Creazione di un controller
5. Configurazione dell'applicazione Spring Boot
  - Auditor
    - AuditorConfig
    - UserAuditorAware
  - I18n
    - MessageLang
    - LanguageResolver
    - Internationalization
  - Filter
    - AuthDummyFilter
    - GenericServletInterceptor
    - LogginFilter
  - handler
    - GlobalExceptionHandler
  - CacheConfig
  - ModelMapperConfig
  - Security
6. Exception
7. DbGenerator

# Introduzione

Un backend è la parte di un'applicazione che si occupa di elaborare i dati e di fornire le risposte alle richieste provenienti dal frontend. Questa guida illustra i passaggi necessari per creare un backend utilizzando Java, Spring Boot, Spring Data JPA e Postgres.

## Creazione di un progetto Spring Boot

Per creare un progetto Spring Boot, è possibile utilizzare Spring Initializr. In questo caso, è necessario selezionare le dipendenze Spring Boot DevTools, Lombok, Spring Web, Spring Security, JDBC API, Spring Data JPA, HyperSQL Database e Validation.

## Configurazione di un database

Un database è un'entità che memorizza i dati in modo strutturato. Ad esempio, un database può contenere tabelle, colonne e righe. In questo caso, è possibile utilizzare un database HSQLDB per memorizzare i dati.

### Configurazione di un database HSQLDB

Per configurare un database HSQLDB, è necessario aggiungere le seguenti proprietà al file `application.properties`:

```
server.port = 8080

# swagger-ui custom path
# springdoc.swagger-ui.path = /swagger-ui.html

# Percorso del file di configurazione di HSQLDB
spring.datasource.url=jdbc:hsqldb:file:./hsqldb/db;hsqldb.write_delay=false
spring.datasource.username = SA
spring.datasource.password =
spring.jpa.hibernate.ddl-auto = create
spring.datasource.show-sql = true

spring.messages.basename = messages, i18n.list
spring.messages.fallback-to-system-locale = false

# Enable actuator endpoints
management.info.env.enabled = true
management.endpoints.web.exposure.include = health,info,metrics
```

## Configurazione di un database Postgres

Per configurare un database Postgres, è necessario aggiungere le seguenti proprietà al file `application.properties`:

```
# URL di connessione al database
spring.datasource.url=jdbc:postgresql://localhost:5432/dbname

# Nome utente del database
spring.datasource.username=username

# Password del database
# La password può essere impostata come variabile d'ambiente o come
# parametro di avvio dell'applicazione.
spring.datasource.password=${DB_PASSWORD}

# Driver JDBC
spring.datasource.driver-class-name=org.postgresql.Driver

# Dialect specifico per Postgres
# Il dialect serve per definire il tipo di database che si sta utilizzando
# e permette di generare le query SQL corrette per quel database.
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# Mostra SQL per debugging, disattivalo in produzione per la sicurezza e
# performance
spring.jpa.show-sql=false

# Strategia di gestione dello schema del database
spring.jpa.hibernate.ddl-auto=validate

# Formatta l'output SQL nel log, utile per il debugging
spring.jpa.properties.hibernate.format_sql=true

# Impostazioni di HikariCP
# HikariCP è un pool di connessioni JDBC ad alte prestazioni per Java che
# offre prestazioni migliori rispetto ai pool di connessioni JDBC
# tradizionali.
spring.datasource.hikari.connection-timeout=20000
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=300000
spring.datasource.hikari.max-lifetime=1800000
spring.datasource.hikari.auto-commit=false # Disabilita l'autocommit per
# prevenire operazioni non intenzionali sul db
```

# Creazione di un'entità, un repository, un servizio e un controller

## Creazione di un'entità

Il passo successivo consiste nella creazione delle entità. Un'entità rappresenta una tabella del database. Ad esempio, la seguente classe rappresenta un'entità `User` nel package `com.example.backend.data.entity`:

```
package com.example.backend.data.entity;

import lombok.Data; // Importa Data da Lombok che genera i getter e i setter
import jakarta.persistence.*; // Importa le annotazioni di JPA
import lombok.Data; // Importa Data da Lombok che genera i getter e i setter
import lombok.NoArgsConstructor; // Importa NoArgsConstructor da Lombok che genera un costruttore vuoto
import org.springframework.data.jpa.domain.support.AuditingEntityListener;
// Importa AuditingEntityListener che permette di registrare le modifiche alle entità nel database
import com.example.backend.core.entityAuditTrailListener.UserListener; // Importa UserListener che permette di registrare le modifiche alle entità User nel database

@Entity // Indica che la classe è un'entità
@Data // Genera i getter e i setter
public class User {
    @Id // Indica che il campo è la chiave primaria
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Genera un valore univoco per la chiave primaria
    private Long id;

    @Basic(optional = false) // Indica che il campo è obbligatorio
    @Column(name = "username", nullable = false) // Indica che il campo è una colonna con il nome "username" e non può essere nullo
    private String username;

    @Basic(optional = false) // Indica che il campo è obbligatorio
    @Column(name = "password", nullable = false) // Indica che il campo è una colonna con il nome "password" e non può essere nullo
    private String password;
}
```

## Creazione di un listener per le entità (opzionale)

Per registrare le modifiche alle entità nel database, è possibile creare un listener. Ad esempio, la seguente classe rappresenta un listener `UserListener` nel package

`com.example.backend.config.entityAuditTrailListener`:

```
package com.example.backend.config.entityAuditTrailListener;

import com.example.backend.data.entity.User; // Importa l'entità User
import jakarta.persistence.*; // Importa le annotazioni di JPA
import org.apache.commons.logging.Log; // Importa Log da Apache Commons
Logging che permette di registrare le modifiche alle entità nel database
import org.apache.commons.logging.LogFactory; // Importa LogFactory da
Apache Commons Logging che permette di creare un logger

public class UserListener {
    private static final Log log = LogFactory.getLog(UserListener.class);

    @PrePersist // Indica che il metodo viene eseguito prima di persistere
l'entità
    public void prePersist(User user) {
        log.info("User " + user.getUsername() + " is being persisted");
    }

    @PostPersist // Indica che il metodo viene eseguito dopo aver
persistito l'entità
    public void postPersist(User user) {
        log.info("User " + user.getUsername() + " has been persisted");
    }

    @PreUpdate // Indica che il metodo viene eseguito prima di aggiornare
l'entità
    public void preUpdate(User user) {
        log.info("User " + user.getUsername() + " is being updated");
    }

    @PostUpdate // Indica che il metodo viene eseguito dopo aver
aggiornato l'entità
    public void postUpdate(User user) {
        log.info("User " + user.getUsername() + " has been updated");
    }

    @PreRemove // Indica che il metodo viene eseguito prima di rimuovere
l'entità
    public void preRemove(User user) {
        log.info("User " + user.getUsername() + " is being removed");
    }

    @PostRemove // Indica che il metodo viene eseguito dopo aver rimosso
l'entità
    public void postRemove(User user) {
        log.info("User " + user.getUsername() + " has been removed");
    }
}
```

## Creazione di un DTO (Data Transfer Object)

Un DTO (Data Transfer Object) è un oggetto che trasporta i dati tra il frontend e il backend. Un DTO può contenere solo i campi necessari per la visualizzazione dei dati e non deve contenere i campi sensibili come la password. Ad esempio, la seguente classe rappresenta un DTO `UserDto` nel package `com.example.backend.data.dto`:

```
package com.example.backend.data.dto;

import lombok.Data;
import lombok.NoArgsConstructor;
import jakarta.validation.constraints.*;

@Data
@NoArgsConstructor
public class UserDto {
    private Long id;

    @NotBlank
    @Size(min = 3, max = 50)
    private String username;

    @NotBlank
    @Size(min = 6, max = 100)
    private String password;
}
```

## Creazione di un repository (DAO)

Una volta creata l'entità, è necessario creare un repository per interagire con il database. Con Spring Data JPA, è possibile creare un repository estendendo l'interfaccia `JpaRepository`. I metodi di base come `save`, `findById`, `findAll`, `delete` e `count` sono già implementati in `JpaRepository`. Inoltre, è possibile definire query personalizzate utilizzando l'annotazione `@Query`.

Quindi ci spostiamo nel package `com.example.backend.data.dao` e creiamo un'interfaccia `UserDao`:

```
package com.example.backend.data.dao;

import com.example.backend.data.entity.User; // Importa User che
rappresenta un'entità User
import org.springframework.data.jpa.repository.JpaRepository; // Importa
JpaRepository da Spring Data JPA che permette di interagire con il
database
import org.springframework.data.jpa.repository.Query; // Importa Query da
Spring Data JPA che permette di definire query personalizzate
import org.springframework.stereotype.Repository; // Importa Repository da
Spring Framework che indica che l'interfaccia è un repository

@Repository
public interface UserDao extends JpaRepository<User, Long> {
    // Query personalizzata per cercare un utente per username
    @Query("SELECT u FROM User u WHERE u.username = ?1")
    User findByUsername(String username);

    // Non è necessario definire i metodi di base come save, findById,
    findAll, delete e count
}
```

## Creazione di un servizio

Successivamente, è necessario creare un servizio per gestire le operazioni sulle entità. Le operazioni possono includere la ricerca, l'aggiornamento e la rimozione delle entità.

### Creazione di un'interfaccia per il servizio

Quindi ci spostiamo nel package `com.example.backend.service` e creiamo un'interfaccia `UserService`:

```
package com.example.backend.service;

import com.example.backend.data.entity.User; // Importa User che
rappresenta un'entità User

public interface UserService {
    void save(User user);
    User findByUsername(String username);
}
```

## Implementazione del servizio

Poi dentro al package `com.example.backend.service.impl` creiamo una classe `UserServiceImpl` che implementa l'interfaccia `UserService`:

```
package com.example.backend.service.impl;

import com.example.backend.data.dao.UserDao; // Importa UserDao che
// permette di interagire con il database
import com.example.backend.data.entity.User; // Importa User che
// rappresenta un'entità User
import com.example.backend.service.UserService; // Importa UserService che
// rappresenta un servizio per gestire le operazioni sulle entità
import lombok.RequiredArgsConstructor; // Importa RequiredArgsConstructor
// da Lombok che genera un costruttore con i parametri richiesti
import org.springframework.stereotype.Service; // Importa Service da
// Spring Framework che indica che la classe è un servizio
import org.springframework.beans.factory.annotation.Autowired; // Importa
// Autowired da Spring Framework che permette di iniettare le dipendenze

@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    @Override
    public void save(User user) {
        userDao.save(user);
    }

    @Override
    public User findByUsername(String username) {
        return userDao.findByUsername(username);
    }
}
```



## Creazione di un controller

Successivamente, è necessario creare un controller per gestire le richieste HTTP provenienti dal frontend. Un controller può includere metodi per gestire le richieste di tipo GET, POST, PUT e DELETE.

Quindi ci spostiamo nel package `com.example.backend.controller` e creiamo una classe `UserController`:

```
package com.example.backend.controller;

import com.example.backend.data.entity.User; // Importa User che
rappresenta un'entità User
import com.example.backend.service.UserService; // Importa UserService che
rappresenta un servizio per gestire le operazioni sulle entità
import lombok.RequiredArgsConstructor; // Importa RequiredArgsConstructor
da Lombok che genera un costruttore con i parametri richiesti
import org.springframework.http.HttpStatus; // Importa HttpStatus da
Spring Framework che rappresenta lo stato HTTP
import org.springframework.http.ResponseEntity; // Importa ResponseEntity
da Spring Framework che rappresenta una risposta HTTP
import org.springframework.web.bind.annotation.*; // Importa
RequestMapping da Spring Framework che permette di mappare le richieste
HTTP ai metodi

@RestController
@RequestMapping("/api/users")
@RequiredArgsConstructor
public class UserController {
    private final UserService userService;

    @PostMapping
    public ResponseEntity<Void> save(@RequestBody User user) {
        userService.save(user);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    @GetMapping("/{username}")
    public ResponseEntity<User> findByUsername(@PathVariable String
username) {
        User user = userService.findByUsername(username);
        return new ResponseEntity<>(user, HttpStatus.OK);
    }
}
```

## Configurazione dell'applicazione Spring Boot

Il package `com.example.backend.config` contiene le classi di configurazione dell'applicazione. Queste classi sono utilizzate per configurare l'applicazione, ad esempio per configurare l'audit delle entità, la localizzazione delle risorse, la sicurezza dell'applicazione e la cache di Spring.

In questo caso specifico, il package `com.example.backend.config` contiene le seguenti classi e package:

- il package `auditor` che contiene le classi per l'audit delle entità.
- il package `i18n` che contiene le classi per la localizzazione delle risorse.
- il package `security` che contiene le classi per la sicurezza dell'applicazione.
- la classe `ModelMapperConfig` che configura `ModelMapper` per mappare le entità ai DTO.
- la classe `CacheConfig` che configura la cache di Spring.

## Auditor

L'audit delle entità è un meccanismo che registra le modifiche alle entità nel database. Ad esempio, quando un'entità viene creata, aggiornata o rimossa, l'audit delle entità registra chi ha effettuato l'operazione e quando è stata effettuata. Questo meccanismo è utile per tenere traccia delle modifiche alle entità e per garantire la conformità alle normative di sicurezza.

## AuditorConfig

Il package `com.example.backend.config.auditor` contiene le classi per l'audit delle entità. Prima di tutto, creiamo la classe `AuditorConfig` che configura l'audit delle entità:

```
package com.example.backend.config.auditor;

import org.springframework.context.annotation.Bean; // Importa Bean da
Spring Framework che permette di definire un bean
import org.springframework.context.annotation.Configuration; // Importa
Configuration da Spring Framework che indica che la classe è una classe di
configurazione
import org.springframework.data.domain.AuditorAware; // Importa
AuditorAware da Spring Data che permette di definire un auditor
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
// Importa EnableJpaAuditing da Spring Data JPA che abilita l'audit delle
entità

import java.util.Optional; // Importa Optional di Java che permette di
gestire i valori nulli

@Configuration // Indica che la classe è una classe di configurazione
@EnableJpaAuditing(auditorAwareRef = "auditorProvider") // Abilita l'audit
delle entità che utilizza l'auditorProvider come riferimento
public class AuditorConfig {
    @Bean // Definisce un bean di AuditorAware
    public AuditorAware<Long> auditorProvider() {
        return new UserAuditorAware();
    }
}
```

## UserAuditorAware

Successivamente creiamo la classe `UserAuditorAware()` che implementa l'interfaccia `AuditorAware` e restituisce l'ID dell'utente corrente:

```
package com.example.backend.config.auditor;

import org.springframework.data.domain.AuditorAware; // Importa
AuditorAware da Spring Data che permette di definire un auditor

import java.util.Optional; // Importa Optional di Java che permette di
gestire i valori nulli

public class UserAuditorAware implements AuditorAware<Long> {

    private static final Long AUTH_CODE = 1_000_001L; // ID dell'utente
corrente

    @Override
    public Optional<Long> getCurrentAuditor() {
        return Optional.of(AUTH_CODE); // Restituisce l'ID dell'utente
corrente
    }
}
```

## i18n

La localizzazione delle risorse è un meccanismo che consente di adattare l'applicazione a diverse lingue e culture. Ad esempio, è possibile creare file di proprietà per le diverse lingue e culture e utilizzare questi file per localizzare le risorse dell'applicazione. Questo meccanismo è utile per rendere l'applicazione più accessibile e per raggiungere un pubblico più ampio.

Il package `com.example.backend.config.i18n` contiene le classi per la localizzazione delle risorse.

## MessageLang

Prima di tutto, creiamo la classe `MessageLang` che configura il `MessageSource` per la localizzazione delle risorse:

```
package com.example.backend.config.i18n;

import lombok.RequiredArgsConstructor; // Importa RequiredArgsConstructor
da Lombok che genera un costruttore con i parametri richiesti
import org.springframework.context.i18n.LocaleContextHolder; // Importa
LocaleContextHolder da Spring Framework che permette di ottenere il locale
corrente
import org.springframework.context.support.ResourceBundleMessageSource; //
Importa ResourceBundleMessageSource da Spring Framework che permette di
caricare le risorse da un file di proprietà
import org.springframework.stereotype.Component; // Importa Component da
Spring Framework che indica che la classe è un componente

@Component // Indica che la classe è un componente
@RequiredArgsConstructor // Genera un costruttore con i parametri
richiesti
public class MessageLang {

    private final ResourceBundleMessageSource messageSource; // Carica le
risorse da un file di proprietà

    public String getMessage(String code) {
        return messageSource.getMessage(code, null,
LocaleContextHolder.getLocale()); // Restituisce il messaggio localizzato
    }

    public String getMessage(String code, Object... args) {
        return messageSource.getMessage(code, args,
LocaleContextHolder.getLocale()); // Restituisce il messaggio localizzato
con gli argomenti
    }
}
```

## LanguageResolver

Successivamente, creiamo la classe `LanguageResolver` che risolve il locale corrente:

```
package com.example.backend.config.i18n;

import jakarta.servlet.http.HttpServletRequest; // Importa
HttpServletRequest di Java che rappresenta una richiesta HTTP
import org.springframework.util.StringUtils; // Importa StringUtils da
Apache Commons Lang che fornisce metodi per la manipolazione delle
stringhe
import org.springframework.stereotype.Component; // Importa Component da
Spring Framework che indica che la classe è un componente
import org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver; //
Importa AcceptHeaderLocaleResolver da Spring Framework che risolve il
locale corrente

import java.util.List; // Importa List di Java che rappresenta una lista
import java.util.Locale; // Importa Locale di Java che rappresenta un
locale

@Component
public class LanguageResolver extends AcceptHeaderLocaleResolver {

    private static final List<Locale> LOCALES = List.of(new Locale("en"),
new Locale("it"));

    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        String language = request.getHeader("Accept-Language");
        List<Locale> supportedLocales = getSupportedLocales();
        Locale defaultLocale = getDefaultLocale();

        if (StringUtils.isEmpty(language)) {
            return defaultLocale;
        }
        Locale requestLocale = Locale.forLanguageTag(language);
        if (supportedLocales.contains(requestLocale)) {
            return requestLocale;
        } else {
            return defaultLocale;
        }
    }
}
```

## Internationalization

Infine, creiamo la classe `Internationalization` che configura la localizzazione delle risorse:

```
package com.example.backend.config.i18n;

import org.springframework.context.annotation.Bean; // Importa Bean da
Spring Framework che permette di definire un bean
import org.springframework.context.annotation.Configuration; // Importa
Configuration da Spring Framework che indica che la classe è una classe di
configurazione
import org.springframework.context.support.ResourceBundleMessageSource; //
Importa ResourceBundleMessageSource da Spring Framework che permette di
caricare le risorse da un file di proprietà
import org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver; //
Importa AcceptHeaderLocaleResolver da Spring Framework che risolve il
locale corrente

import java.util.Arrays; // Importa Arrays di Java che fornisce metodi per
manipolare gli array
import java.util.Locale; // Importa Locale di Java che rappresenta un
locale

@Configuration
public class Internationalization /*extends WebMvcConfigurerAdapter*/ {

    @Bean
    public AcceptHeaderLocaleResolver localeResolver() {
        final LanguageResolver resolver = new LanguageResolver();
        resolver.setSupportedLocales(Arrays.asList(Locale.ITALY,
Locale.US,Locale.UK));
        resolver.setDefaultLocale(Locale.ITALY);
        return resolver;
    }

    @Bean
    public ResourceBundleMessageSource messageSource() {
        final ResourceBundleMessageSource source = new
ResourceBundleMessageSource();
        source.setBasename("language/messages");
        source.setDefaultEncoding("UTF-8");
        return source;
    }
}
```

## Filter

Il package `com.example.backend.config.filter` contiene le classi per i filtri HTTP personalizzati. I filtri HTTP sono utilizzati per eseguire operazioni prima o dopo l'invio di una richiesta al controller. Ad esempio, è possibile utilizzare un filtro per autenticare le richieste, per registrare le richieste e le risposte, o per gestire le eccezioni.

## AuthDummyFilter

La classe `AuthDummyFilter` è un filtro HTTP personalizzato che viene utilizzato per l'autenticazione in un'applicazione web Java. È annotata con `@WebFilter`, il che significa che viene applicata a tutte le richieste HTTP che corrispondono al pattern URL specificato, in questo caso `"/**"` (tutte le richieste).

```
package com.example.backend.config.filter;

import jakarta.security.auth.message.AuthException; // Importa
AuthException di Java che rappresenta un'eccezione di autenticazione
import jakarta.servlet.FilterChain; // Importa FilterChain di Java che
rappresenta una catena di filtri
import jakarta.servlet.ServletException; // Importa ServletException di
Java che rappresenta un'eccezione di servlet
import jakarta.servlet.annotation.WebFilter; // Importa WebFilter di Java
che indica che la classe è un filtro HTTP
import jakarta.servlet.http.HttpServletRequest; // Importa
HttpServletRequest di Java che rappresenta una richiesta HTTP
import jakarta.servlet.http.HttpServletResponse; // Importa
HttpServletResponse di Java che rappresenta una risposta HTTP
import lombok.RequiredArgsConstructor; // Importa RequiredArgsConstructor
da Lombok che genera un costruttore con i parametri richiesti
import lombok.extern.slf4j.Slf4j; // Importa log4j da Lombok che permette
di registrare i messaggi di log
import org.springframework.core.Ordered; // Importa Ordered da Spring
Framework che permette di ordinare i filtri
import org.springframework.core.annotation.Order; // Importa Order da
Spring Framework che permette di ordinare i filtri
import org.springframework.data.domain.AuditorAware; // Importa
AuditorAware da Spring Data che permette di definire un auditor
import org.springframework.http.HttpHeaders; // Importa HttpHeaders da
Spring Framework che rappresenta gli header HTTP
import org.springframework.http.HttpStatus; // Importa HttpStatus da
Spring Framework che rappresenta lo stato HTTP
import org.springframework.http.HttpStatusCode; // Importa HttpStatusCode
da Spring Framework che rappresenta lo stato HTTP
import org.springframework.stereotype.Component; // Importa Component da
Spring Framework che indica che la classe è un componente
import org.springframework.web.filter.AbstractRequestLoggingFilter; //
Importa AbstractRequestLoggingFilter da Spring Framework che permette di
registrare le richieste HTTP
import org.springframework.web.filter.OncePerRequestFilter; // Importa
OncePerRequestFilter da Spring Framework che permette di eseguire il
filtro una volta per ogni richiesta

import java.io.IOException;

@WebFilter(urlPatterns = "/*")
@Order(1)
@RequiredArgsConstructor
public class AuthDummyFilter extends OncePerRequestFilter {

    private final AuditorAware<Long> currentUser;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws IOException
{
```

```

    try {
        if (request.getHeader(HttpHeaders.AUTHORIZATION) != null &&
            currentUser.getCurrentAuditor().isPresent()) {
            long auth =
                Long.valueOf(request.getHeader(HttpHeaders.AUTHORIZATION));
            if (currentUser.getCurrentAuditor().get() == auth) {
                filterChain.doFilter(request, response);
            } else {
                throw new AuthException("Wrong credential [" + auth + "]");
            }
        } else {
            throw new AuthException("Authorization Header not found");
        }
    } catch (Exception ex) {
        response.sendError(HttpStatus.FORBIDDEN.value(), ex.getMessage());
    }
}
}

```

## GenericServletInterceptor

La classe `GenericServletInterceptor` è un interceptor di Spring MVC. Gli interceptor sono utilizzati per eseguire operazioni prima o dopo l'invio di una richiesta al controller.

In particolare, questa classe implementa tre metodi:

- `preHandle`: viene chiamato prima che la richiesta venga inviata al controller. In questo caso, il metodo registra il messaggio "preHandle" e restituisce `true`, permettendo alla richiesta di procedere al controller.
- `postHandle`: viene chiamato dopo che il controller ha elaborato la richiesta, ma prima che la vista venga renderizzata. In questo caso, il metodo registra il messaggio "postHandle".
- `afterCompletion`: viene chiamato dopo che la vista è stata renderizzata e che la richiesta è stata completamente elaborata. In questo caso, il metodo registra il messaggio "afterCompletion".

```

package com.example.backend.config.filter;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

@Component
@Slf4j
public class GenericServletInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
        response, Object handler) throws Exception {

        log.info("preHandle");
    }
}

```



```

        return true; // mandiamo la richiesta al controller
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
        log.info("postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        HandlerInterceptor.super.afterCompletion(request, response, handler,
ex);
        log.info("afterCompletion");
    }
}

```

## LoginFilter

La classe `LoggingFilter` è un filtro di logging che registra le richieste HTTP. Estende `AbstractRequestLoggingFilter` e sovrascrive i metodi `beforeRequest` e `afterRequest` per registrare i messaggi di log prima e dopo l'invio della richiesta al controller.

```

package com.example.backend.config.filter;

package it.unical.backend.config.filter;

import jakarta.servlet.http.HttpServletRequest;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.AbstractRequestLoggingFilter;

@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
@Slf4j
public class LoggingFilter extends AbstractRequestLoggingFilter {

    public LoggingFilter() {
        setIncludePayload(true);
        setIncludeHeaders(true);
        setIncludeQueryString(true);
    }

    @Override
    protected void beforeRequest(HttpServletRequest request, String message)
    {
        log.info(message);
    }

    @Override
    protected void afterRequest(HttpServletRequest request, String message)
    {

```

```
        log.info(message);
    }
}
```

## Handler

Il package `com.example.backend.config.handler` contiene le classi per la gestione delle eccezioni. Le classi di gestione delle eccezioni sono utilizzate per gestire le eccezioni che si verificano durante l'esecuzione dell'applicazione. Ad esempio, è possibile utilizzare una classe di gestione delle eccezioni per gestire le eccezioni di autenticazione, le eccezioni di autorizzazione e le eccezioni di validazione.

## GlobalExceptionHandler

La classe `GlobalExceptionHandler` è una classe di gestione delle eccezioni globale che gestisce le eccezioni che si verificano durante l'esecuzione dell'applicazione. È annotata con `@RestControllerAdvice`, il che significa che viene applicata a tutti i controller dell'applicazione. La classe contiene metodi annotati con `@ExceptionHandler` per gestire le diverse eccezioni che possono verificarsi.

[illegible]

```

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ServiceError onMethodArgumentNotValid(WebRequest req,
    MethodArgumentNotValidException ex){

        String message = ex.getBindingResult().getFieldErrors().stream()
                                .map(viol ->
viol.getField().concat(" : ")
.concat(viol.getDefaultMessage()))
                                .collect(Collectors.joining("
, "));
        return ErrorResponse(req, message);
    }

    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ServiceError defaultErrorHandler(WebRequest req ,Exception ex){
        return ErrorResponse(req, ex.getMessage());
    }

    private ServiceError ErrorResponse (WebRequest req, String message) {
        HttpServletRequest httpreq = (HttpServletRequest)
req.resolveReference("request");
        final ServiceError output = new ServiceError(new Date(),
httpreq.getRequestURI(), message);
        log.error("Exception handler :::: {} ", output.toString());
        return output;
    }
}

```

## CacheConfig

La cache di Spring è un meccanismo che memorizza temporaneamente i dati in memoria per ridurre il tempo di risposta delle richieste. Ad esempio, è possibile memorizzare in cache i risultati delle query per evitare di eseguire la stessa query più volte. Questo meccanismo è utile per migliorare le prestazioni dell'applicazione e per ridurre il carico sul database.

Il package `com.example.backend.config` contiene la classe `CacheConfig` che configura la cache di Spring:

```

package com.example.backend.config;

import org.slf4j.Logger; // Importa Logger da SLF4J che permette di
registrare i messaggi di log
import org.slf4j.LoggerFactory; // Importa LoggerFactory da SLF4J che
permette di creare un logger
import org.springframework.cache.CacheManager; // Importa CacheManager da
Spring Framework che permette di gestire la cache
import org.springframework.cache.annotation.CacheEvict; // Importa
CacheEvict da Spring Framework che permette di rimuovere i dati dalla
cache
import org.springframework.cache.annotation.EnableCaching; // Importa
EnableCaching da Spring Framework che abilita la cache
import org.springframework.cache.concurrent.ConcurrentMapCacheManager; //

```

```

Importa ConcurrentMapCacheManager da Spring Framework che gestisce la
cache in memoria
import org.springframework.context.annotation.Bean; // Importa Bean da
Spring Framework che permette di definire un bean
import org.springframework.context.annotation.Configuration; // Importa
Configuration da Spring Framework che indica che la classe è una classe di
configurazione
import org.springframework.scheduling.annotation.EnableScheduling; //
Importa EnableScheduling da Spring Framework che abilita la pianificazione
delle attività
import org.springframework.scheduling.annotation.Scheduled; // Importa
Scheduled da Spring Framework che permette di pianificare l'esecuzione di
un metodo

import java.time.LocalDateTime; // Importa LocalDateTime di Java che
rappresenta una data e un'ora
import java.time.format.DateTimeFormatter; // Importa DateTimeFormatter di
Java che permette di formattare le date e le ore

@Configuration // Indica che la classe è una classe di configurazione
@EnableCaching // Abilita la cache
@EnableScheduling // Abilita la pianificazione delle attività
public class CacheConfig {
    private static final Logger logger =
LoggerFactory.getLogger(CacheConfig.class); // Crea un logger per la
classe

    public static final DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"); // Formatta la data e
l'ora

    public static final String CACHE_FOR_USERS = "USER"; // Nome della
cache per gli utenti

    @Bean("cacheManager")
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager(CACHE_FOR_USERS); // Crea un
gestore della cache per gli utenti
    }

    @CacheEvict(allEntries = true, value = {CACHE_FOR_USERS}) // Rimuove
tutti i dati dalla cache per gli utenti
    @Scheduled(fixedDelay = 10 * 60 * 1000, initialDelay = 500) //
Pianifica l'esecuzione del metodo ogni 10 minuti
    public void userCacheEvict() {
        logger.info(String.format("Flush Cache[%s] at [%s]",
CACHE_FOR_USERS, formatter.format(LocalDateTime.now()))); // Registra un
messaggio di log
    }
}

```

## InterceptorConfig

Un interceptor di Spring MVC è un oggetto che può essere utilizzato per eseguire operazioni prima o dopo l'invio di una richiesta al controller. Gli interceptor sono utilizzati per eseguire operazioni come l'autenticazione, l'autorizzazione, il logging e la gestione delle eccezioni.

```

package com.example.backend.config;

package it.unical.backend.config;

import it.unical.backend.config.filter.GenericServletInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class InterceptorConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new
GenericServletInterceptor()).order(Ordered.LOWEST_PRECEDENCE);
        // WebMvcConfigurer.super.addInterceptors(registry);
    }
}

```

## ModelMapperConfig

ModelMapper è una libreria che consente di mappare le entità ai DTO in modo automatico. Ad esempio, è possibile creare un oggetto `ModelMapper` e utilizzarlo per mappare le entità ai DTO e viceversa. Questo meccanismo è utile per ridurre il codice ripetitivo e per migliorare la manutenibilità dell'applicazione.

Il package `com.example.backend.config` contiene la classe `ModelMapperConfig` che configura `ModelMapper`:

```

package com.example.backend.config;

import it.unical.backend.data.entity.User; // Importa User che rappresenta
un'entità User
import it.unical.backend.dto.UserDto; // Importa UserDto che rappresenta
un DTO User
import org.modelmapper.ModelMapper; // Importa ModelMapper che permette di
mappare le entità ai DTO
import org.modelmapper.PropertyMap; // Importa PropertyMap da ModelMapper
che permette di definire le proprietà di mappatura
import org.springframework.context.annotation.Bean; // Importa Bean da
Spring Framework che permette di definire un bean
import org.springframework.context.annotation.Configuration; // Importa
Configuration da Spring Framework che indica che la classe è una classe di
configurazione

@Configuration // Indica che la classe è una classe di configurazione
public class ModelMapperConfig {

    @Bean // Definisce un bean di ModelMapper
    public ModelMapper modelMapper() {
        ModelMapper modelMapper = new ModelMapper(); // Crea un oggetto
ModelMapper
    }
}

```

```

modelMapper.getConfiguration().setFieldMatchingEnabled(true).setFieldAccessLevel(org.modelmapper.config.Configuration.AccessLevel.PRIVATE); //
Abilita il matching dei campi e imposta il livello di accesso ai campi

    return modelMapper; // Restituisce l'oggetto ModelMapper
}

// Genera il nome completo dell'utente
private String generateFullName(String firstName, String lastName) {
    return firstName + " " + lastName; // Restituisce il nome completo
    dell'utente
}
}

```

## Security

## Exception

## DbGenerator

Dentro al package `com.example.backend` allo stesso livello del Main creiamo la classe `DbGenerator` che permette di popolare il database con

```

package com.example.backend;

import com.example.backend.data.entity.User; // Importa User che
rappresenta un'entità User
import com.example.backend.service.UserService; // Importa UserService che
rappresenta un servizio per gestire le operazioni sulle entità
import org.springframework.beans.factory.annotation.Autowired; // Importa
Autowired da Spring Framework che permette di iniettare le dipendenze
import org.springframework.beans.factory.annotation.Value; // Importa
Value da Spring Framework che permette di ottenere i valori dalle
proprietà
import org.springframework.boot.ApplicationArguments; // Importa
ApplicationArguments da Spring Boot che rappresenta gli argomenti
dell'applicazione
import org.springframework.boot.ApplicationRunner; // Importa
ApplicationRunner da Spring Boot che permette di eseguire il codice
all'avvio dell'applicazione
import org.springframework.core.io.Resource; // Importa Resource da Spring
Framework che rappresenta una risorsa
import org.springframework.stereotype.Component; // Importa Component da
Spring Framework che indica che la classe è un componente
import org.apache.commons.csv.CSVFormat; // Importa CSVFormat da Apache
Commons CSV che permette di leggere e scrivere file CSV
import org.apache.commons.csv.CSVParser; // Importa CSVParser da Apache
Commons CSV che permette di analizzare un file CSV
import org.apache.commons.csv.CSVRecord; // Importa CSVRecord da Apache
Commons CSV che rappresenta un record CSV

import java.io.IOException; // Importa IOException di Java che rappresenta
un'eccezione di I/O
import java.io.InputStreamReader; // Importa InputStreamReader di Java che
legge i caratteri da un flusso di input

@Component
public class DbGenerator implements ApplicationRunner {

```

```

    @Value("classpath:data/users.csv") // Legge il file users.csv dalla
    cartella data
    private Resource usersCsv; // Rappresenta il file users.csv

    @Autowired
    private UserService userService; // Inietta il servizio UserService

    public void createDb() {
        try (CSVParser parser =
CSVFormat.DEFAULT.withDelimiter().parse(new
InputStreamReader(usersCsv.getInputStream())) {
            for (CSVRecord record : parser) {
                insertUser(record.get(0), record.get(1)); // Inserisce un
utente nel database con username e password
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void insertUser(String username, String password) {
        User user = new User(); // Crea un nuovo utente
        user.setUsername(username); // Imposta l'username dell'utente
        user.setPassword(password); // Imposta la password dell'utente
        userService.save(user); // Salva l'utente nel database
    }

    @Override
    public void run(ApplicationArguments args) {
        createDb(); // Crea il database
    }
}

```

## Riepilogo dei passaggi

Questa guida illustra i passaggi necessari per creare un backend utilizzando Java, Spring Boot, Spring Data JPA e Postgres. I passaggi includono:

1. Creazione del progetto Spring Boot con le dipendenze necessarie con Spring Initializr.
2. Configurazione di un database HSQLDB o Postgres nel file `application.properties`.
3. Creazione di un'entità, un repository, un servizio e un controller per gestire le operazioni sulle entità.
4. Configurazione dell'applicazione Spring Boot con le classi di configurazione per l'audit delle entità, la localizzazione delle risorse, la sicurezza dell'applicazione, ModelMapper e la cache di Spring.
5. Creazione di una classe `DbGenerator` per popolare il database con i dati di esempio.