Source Code

MyMoviePlan:

MyMoviePlanApplication:

```
package com.MyMoviePlan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MyMoviePlanApplication {
   public static void main(String[] args) {
      SpringApplication.run(MyMoviePlanApplication.class, args);
   }
}
```

ServletInitializer:

```
package com.MyMoviePlan;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
public class ServletInitializer extends SpringBootServletInitializer {
	@Override
	protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
		return application.sources(MyMoviePlanApplication.class);
	}
}
```

Config: InitialData:

```
package com.MyMoviePlan.config;
import com.MyMoviePlan.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;
@Component
public class InitialData implements CommandLineRunner {
   @Autowired
   private UserService service;
   @Autowired
   private PasswordEncoder passwordEncoder;
   @Override
   public void run(String... args) throws Exception {
   }
}
```

Controller:
AuditoriumController:

```
package com.MyMoviePlan.controller;
import com.MyMoviePlan.entity.*;
import com.MyMoviePlan.exception.AuditoriumNotFoundException;
import com.MyMoviePlan.exception.BookingNotFoundException;
import com.MyMoviePlan.exception.MovieShowNotFoundException;
import com.MyMoviePlan.exception.ShowNotFoundException;
import com.MyMoviePlan.model.TicketDetails;
import com.MyMoviePlan.model.UserRole;
import com.MyMoviePlan.repository.*;
import com.MyMoviePlan.service.UserService;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import java.util.stream.Collectors;
@CrossOrigin
@RestController
@RequestMapping("/auditorium")
@AllArgsConstructor
public class AuditoriumController {
  private final ShowRepository show;
  private final UserService service;
  private final BookingRepository booking;
  private final MovieRepository movie;
  private final MovieShowsRepository movieShow;
  private final AuditoriumRepository auditorium;
  @GetMapping({"/", "all"})
  public List<AuditoriumEntity> findAllAuditoriums() {
    return this.auditorium.findAll();
  }
  @GetMapping("{auditorium_id}")
  @PreAuthorize("hasAuthority('WRITE')")
  public AuditoriumEntity findAuditoriumById(@PathVariable final int auditorium_id) {
    return this.auditorium.findById(auditorium_id)
        .orElseThrow(() ->
            new AuditoriumNotFoundException("Auditorium with id: " + auditorium_id + "
not found."));
  }
  @PostMapping("add")
  @PreAuthorize("hasAuthority('WRITE')")
  public AuditoriumEntity saveAuditorium(@RequestBody final AuditoriumEntity
auditorium) {
    return this.auditorium.save(auditorium);
  }
  @PutMapping("update")
  @PreAuthorize("hasAuthority('UPDATE')")
  public AuditoriumEntity updateAuditorium(@RequestBody final AuditoriumEntity
auditorium) {
    return this.auditorium.save(auditorium);
  }
  @DeleteMapping("delete/{auditorium_id}")
  @PreAuthorize("hasAuthority('DELETE')")
  public void deleteAuditorium(@PathVariable final int auditorium_id) {
    this.auditorium.deleteById(auditorium_id);
  }
  /*
   *   ============================ Show Controller ==========================
   */
  @GetMapping("{auditorium_id}/show/{show_id}")
  public ShowEntity findShowById(@PathVariable final int auditorium_id,
                                 @PathVariable final int show_id) {
    return this.findAuditoriumById(auditorium_id).getShows()
        .stream()
        .filter(show -> show.getId() == show_id)
        .findFirst()
        .orElseThrow(() ->
                new ShowNotFoundException("Show with Id: " + show_id + " not found"));
  }

  @GetMapping("{auditorium_id}/show/all")
  public List<ShowEntity> findAllShows(@PathVariable final int auditorium_id) {
    return this.findAuditoriumById(auditorium_id).getShows();
  }

  @PostMapping("{auditorium_id}/show/add")
  @PreAuthorize("hasAuthority('WRITE')")
  public ShowEntity saveShow(@PathVariable final int auditorium_id,
                             @RequestBody final ShowEntity show) {
    final AuditoriumEntity auditorium = this.findAuditoriumById(auditorium_id);
    show.setAuditorium(auditorium);
    return this.show.save(show);
  }

  @PutMapping("{auditorium_id}/show/update")
  @PreAuthorize("hasAuthority('UPDATE')")
  public ShowEntity updateShow(@PathVariable final int auditorium_id,
                       @RequestBody final ShowEntity show) {
    final AuditoriumEntity auditorium = this.findAuditoriumById(auditorium_id);
    show.setAuditorium(auditorium);
    return this.show.save(show);
  }

  @DeleteMapping("{auditorium_id}/show/delete/{show_id}")
  @PreAuthorize("hasAuthority('DELETE')")
  public void deleteShow(@PathVariable final int auditorium_id,
                     @PathVariable final int show_id) {
    final ShowEntity show = this.findShowById(auditorium_id, show_id);
    this.show.deleteById(show.getId());
  }

  /*
   *   ============================ Movie Show Controller
=========================
   */

  @GetMapping("movie/{movieId}")
  public List<AuditoriumEntity> findAuditoriumsByMovieId(@PathVariable final int
movieId) {
    return this.findAllAuditoriums().stream()
        .filter(halls -> halls.getShows()
                .stream()
                .anyMatch(show -> show.getMovieShows()
                        .stream()
                        .anyMatch(m_show -> m_show.getMovieId() == movieId)))
        .collect(Collectors.toList());
  }

  @GetMapping("{auditorium_id}/movie/{movieId}")
  public List<ShowEntity> findShowsByMovieId(@PathVariable final int auditorium_id,
@PathVariable final int movieId) {
    return this.findAllShows(auditorium_id).stream()
        .filter(show -> show.getMovieShows()
                .stream()
                .anyMatch(m_show -> m_show.getMovieId() == movieId))
        .collect(Collectors.toList());
  }

  @GetMapping("{auditorium_id}/show/{show_id}/movie-show/all")
  public List<MovieShowsEntity> findAllMovieShows(@PathVariable final int auditorium_id,
                                  @PathVariable final int show_id) {
```

```
        return this.findShowById(auditorium_id, show_id)
                .getMovieShows();
    }

    @GetMapping("{auditorium_id}/show/{show_id}/movie-show/{movie_show_id}")
    public MovieShowsEntity findMovieShowById(@PathVariable final int auditorium_id,
                                              @PathVariable final int show_id,
                                              @PathVariable final int movie_show_id) {
        return this.findShowById(auditorium_id, show_id)
                .getMovieShows()
                .stream()
                .filter(movie_show -> movie_show.getId() == movie_show_id)
                .findFirst()
                .orElseThrow(
                        () -> new MovieShowNotFoundException("Movie Show with id: "
                                + movie_show_id + " not found"));
    }

    @PostMapping("{auditorium_id}/show/{show_id}/movie-show/add")
    @PreAuthorize("hasAuthority('WRITE')")
    public MovieShowsEntity saveMovieShow(@PathVariable final int auditorium_id,
                                          @PathVariable final int show_id,
                                          @RequestBody final MovieShowsEntity movieShow) {
        final ShowEntity show = this.findShowById(auditorium_id, show_id);
        final int movieId = movieShow.getMovieId();
        movieShow.setShow(show);
        movieShow.setMovieId(this.movie.findById(movieId).get().getId());
        return this.movieShow.save(movieShow);
    }

    @PutMapping("{auditorium_id}/show/{show_id}/movie-show/update")
    @PreAuthorize("hasAuthority('UPDATE')")
    public MovieShowsEntity updateMovieShow(@PathVariable final int auditorium_id,
                                            @PathVariable final int show_id,
                                            @RequestBody final MovieShowsEntity movieShow) {
        final ShowEntity show = this.findShowById(auditorium_id, show_id);
        movieShow.setShow(show);
        return this.movieShow.save(movieShow);
    }

    @DeleteMapping("{auditorium_id}/show/{show_id}/movie-
show/delete/{movie_show_id}")
    @PreAuthorize("hasAuthority('DELETE')")
    public void deleteMovieShow(@PathVariable final int auditorium_id,
                        @PathVariable final int show_id,
                        @PathVariable final int movie_show_id) {
        final MovieShowsEntity movieShow = this.findMovieShowById(auditorium_id, show_id,
movie_show_id);
        this.movieShow.deleteById(movieShow.getMovieId());
    }

    /*
     *   ============================= Booking Controller
=========================
     */

    @GetMapping("{auditorium_id}/show/{show_id}/movie-
show/{movie_show_id}/booking/{booking_id}")
    @PreAuthorize("hasAuthority('READ')")
    public BookingEntity findBookingById(@PathVariable final int auditorium_id,
                               @PathVariable final int show_id,
                               @PathVariable final int movie_show_id,
                               @PathVariable final int booking_id) {
        final MovieShowsEntity movieShow = this.findMovieShowById(auditorium_id, show_id,
movie_show_id);
        return movieShow.getBookings()
                .stream().filter(booking -> booking.getId() == booking_id)
                .findFirst()
                .orElseThrow(() -> new BookingNotFoundException("Booking with id: "
                        + booking_id + " not found."));
    }

    @GetMapping("{auditorium_id}/show/{show_id}/movie-
show/{movie_show_id}/booking/all")
    @PreAuthorize("hasAuthority('WRITE')")
    public List<BookingEntity> allBookings(@PathVariable final int auditorium_id,
                                @PathVariable final int show_id,
                                @PathVariable final int movie_show_id) {
        final UserEntity user = this.service.getLoggedInUser();
        if (user.getUserRole().equals(UserRole.ROLE_ADMIN) ||
user.getUserRole().equals(UserRole.ROLE_SUPER_ADMIN))
            return this.findMovieShowById(auditorium_id, show_id,
movie_show_id).getBookings();
        else
            return this.findMovieShowById(auditorium_id, show_id,
movie_show_id).getBookings()
                    .stream().filter(booking -> booking.getUserId().equals(user.getId()))
                    .collect(Collectors.toList());
    }

    @PostMapping("{auditorium_id}/show/{show_id}/movie-
show/{movie_show_id}/booking/add")
//    @PreAuthorize("hasAuthority('WRITE')")
    public BookingEntity saveBooking(@PathVariable final int auditorium_id,
                                     @PathVariable final int show_id,
                                     @PathVariable final int movie_show_id,
                                     @RequestBody final BookingEntity booking) {
        final MovieShowsEntity moveShow = this.findMovieShowById(auditorium_id, show_id,
movie_show_id);
        booking.setUserId(this.service.getLoggedInUser().getId());
//        booking.setUserId(this.service.findByMobile("8099531318").get().getId());
        booking.setMovieShow(moveShow);
        booking.setBookingDetails(new BookingDetailsEntity(auditorium_id, show_id,
movie_show_id, moveShow.getMovieId()));
        return this.booking.save(booking);
    }

    @PutMapping("{auditorium_id}/show/{show_id}/movie-
show/{movie_show_id}/booking/update")
    @PreAuthorize("hasAuthority('UPDATE')")
    public BookingEntity updateBooking(@PathVariable final int auditorium_id,
                                       @PathVariable final int show_id,
                                       @PathVariable final int movie_show_id,
                                       @RequestBody final BookingEntity booking) {
        final MovieShowsEntity moveShow = this.findMovieShowById(auditorium_id, show_id,
movie_show_id);
        booking.setMovieShow(moveShow);
        return this.booking.save(booking);
    }

    @DeleteMapping("{auditorium_id}/show/{show_id}/movie-
show/{movie_show_id}/booking/delete/{booking_id}")
    @PreAuthorize("hasAuthority('READ')")
    public void deleteBookingById(@PathVariable final int auditorium_id,
                                  @PathVariable final int show_id,
                                  @PathVariable final int movie_show_id,
                                  @PathVariable final int booking_id) {
        final BookingEntity booking = this.findBookingById(auditorium_id, show_id,
movie_show_id, booking_id);
        this.booking.deleteById(booking.getId());
    }
```

```
   @GetMapping("ticket-details/{booking_id}")
   @PreAuthorize("hasAuthority('READ')")
   public TicketDetails getMovieDetails(@PathVariable final int booking_id) {

       final PaymentEntity payment = this.booking.findById(booking_id).get().getPayment();

       final MovieShowsEntity movieShow = this.movieShow.findAll().stream().filter(m_show -
> m_show.getBookings()
               .stream().anyMatch(booking -> booking.getId() == booking_id)).findFirst().get();

       final MovieEntity movie = this.movie.findById(movieShow.getMovieId()).get();

       final ShowEntity showEntity = show.findAll().stream()
               .filter(show -> show.getMovieShows()
                       .stream().anyMatch(m_show -> m_show.getId() ==
movieShow.getId())).findFirst().get();

       final AuditoriumEntity auditorium = this.auditorium.findAll().stream().filter(hall ->
hall.getShows()
               .stream().anyMatch(show -> show.getId() == showEntity.getId())).findFirst().get();

       return new TicketDetails(auditorium.getName(), showEntity.getName(),
showEntity.getStartTime(), payment.getAmount(), movie.getName(), movie.getImage(),
movie.getBgImage());
   }
}
```

BookingController:

```
package com.MyMoviePlan.controller;

import com.MyMoviePlan.entity.BookingDetailsEntity;
import com.MyMoviePlan.entity.BookingEntity;
import com.MyMoviePlan.entity.UserEntity;
import com.MyMoviePlan.exception.BookingNotFoundException;
import com.MyMoviePlan.model.UserRole;
import com.MyMoviePlan.repository.BookingRepository;
import com.MyMoviePlan.service.UserService;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@CrossOrigin
@RestController
```

```
@RequestMapping("/booking")
@AllArgsConstructor
public class BookingController {

    private final BookingRepository repository;
    private final UserService service;

    @GetMapping("{id}")
    @PreAuthorize("hasAuthority('READ')")
    public BookingEntity findById(@PathVariable final int id) {
        return repository.findById(id)
                .orElseThrow(() -> new BookingNotFoundException("Booking with id: " + id + " not
found."));
    }

    @GetMapping("all")
    @PreAuthorize("hasAuthority('READ')")
    public List<BookingEntity> allBookings() {
        final UserEntity user = service.getLoggedInUser();
        if (user.getUserRole().equals(UserRole.ROLE_ADMIN) ||
user.getUserRole().equals(UserRole.ROLE_SUPER_ADMIN))
            return repository.findAll();
        else return repository.findAllByUserIdOrderByBookedOnAsc(user.getId());
    }

    @GetMapping("{username}/all")
    @PreAuthorize("hasAuthority('READ')")
    public List<BookingEntity> findAllByUserId(@PathVariable String username) {
        if (!(username.contains("-") && username.length() > 10))
            username = service.getUser(username).getId();
        return repository.findAllByUserIdOrderByBookedOnAsc(username);
    }

    @DeleteMapping("delete/{id}")
    @PreAuthorize("hasAuthority('DELETE')")
    public void deleteBooking(@PathVariable final int id) {
        repository.deleteById(id);
    }

    @GetMapping("{id}/details")
    @PreAuthorize("hasAuthority('READ')")
    public BookingDetailsEntity findByDetailsId(@PathVariable final int id) {
        return this.findById(id).getBookingDetails();
    }
```

```
}
MovieController:
package com.MyMoviePlan.controller;

import com.MyMoviePlan.entity.MovieEntity;
import com.MyMoviePlan.exception.MovieNotFoundException;
import com.MyMoviePlan.repository.MovieRepository;
import com.MyMoviePlan.repository.MovieShowsRepository;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.*;

@CrossOrigin
@RestController
@RequestMapping("/movie")
@AllArgsConstructor
public class MovieController {

  private final MovieRepository movieRepository;
  private final MovieShowsRepository movieShowsRepository;

  @GetMapping({"/", "all"})
  public List<MovieEntity> findAll() {
    return movieRepository.findAll();
  }

  @GetMapping("{movie_id}")
  public MovieEntity findById(@PathVariable final int movie_id) {
    return movieRepository.findById(movie_id)
        .orElseThrow(() -> new MovieNotFoundException("Movie with movie_id: " +
movie_id + " not found."));
  }

  @GetMapping("up-coming")
  public List<MovieEntity> upComing(@RequestParam(value = "records", required = false)
Optional<String> records) {
    List<MovieEntity> movies;
    List<MovieEntity> allMovies;
    if (records.isPresent()) {
      movies = new ArrayList<>();
      allMovies = this.findAll();
      movieShowsRepository.findFewUpComing(Integer.parseInt(records.get()))
```

```
                .forEach(m_show -> movies.add(allMovies.stream()
                        .filter(movie -> (movie.getId() == m_show.getMovieId() &&
movie.getRelease().getTime() > new Date().getTime()))
                        .findFirst().orElse(null)));
        } else {
            movies = new ArrayList<>();
            allMovies = this.findAll();
            movieShowsRepository.findAllUpComing()
                .forEach(m_show -> movies.add(allMovies.stream()
                        .filter(movie -> movie.getId() == m_show.getMovieId() &&
movie.getRelease().getTime() > new Date().getTime())
                        .findFirst().orElse(null)));
        }
//        return (movies.size() > 0 && !movies.contains(null)) ? movies : new ArrayList<>();
        movies.removeAll(Collections.singletonList(null));
        return movies;
    }

    @GetMapping("now-playing")
    public List<MovieEntity> nowPlaying(@RequestParam(value = "records", required = false)
Optional<String> records) {
        List<MovieEntity> movies;
        List<MovieEntity> allMovies;
        if (records.isPresent()) {
            movies = new ArrayList<>();
            allMovies = this.findAll();
            movieShowsRepository.findFewNowPlaying(Integer.parseInt(records.get()))
                .forEach(m_show -> movies.add(allMovies.stream()
                        .filter(movie -> movie.getId() == m_show.getMovieId())
                        .findFirst().orElse(null)));
        } else {
            movies = new ArrayList<>();
            allMovies = this.findAll();
            movieShowsRepository.findAllNowPlaying()
                .forEach(m_show -> movies.add(allMovies.stream()
                        .filter(movie -> movie.getId() == m_show.getMovieId())
                        .findFirst().orElse(null)));
        }
        movies.removeAll(Collections.singletonList(null));
        return movies;
    }

    @GetMapping("now-playing-up-coming")
    public List<MovieEntity> nowPlayingAndUpComing() {
        final List<MovieEntity> movies = new ArrayList<>();
        final List<MovieEntity> allMovies = this.findAll();
        movieShowsRepository.findAllNowPlayingAndUpComing()
                .forEach(m_show -> movies.add(allMovies.stream()
                        .filter(movie -> movie.getId() == m_show.getMovieId())
                        .findFirst().orElse(null)));
        movies.removeAll(Collections.singletonList(null));
        return movies;
    }

    @GetMapping("not-playing")
    public List<MovieEntity> notPlaying() {
        final List<MovieEntity> movies = new ArrayList<>();
        final List<MovieEntity> allMovies = this.findAll();
        movieShowsRepository.findAllNotPlaying()
                .forEach(m_show -> movies.add(allMovies.stream()
                        .filter(movie -> movie.getId() == m_show.getMovieId())
                        .findFirst().orElse(null)));
        movies.removeAll(Collections.singletonList(null));
        return movies;
    }

    @PostMapping("add")
    @PreAuthorize("hasAuthority('WRITE')")
    public MovieEntity saveMovie(@RequestBody final MovieEntity movie) {
        return movieRepository.save(movie);
    }

    @PutMapping("update")
    @PreAuthorize("hasAuthority('UPDATE')")
    public MovieEntity updateMovie(@RequestBody final MovieEntity movie) {
        return movieRepository.save(movie);
    }

    @DeleteMapping("delete/{movie_id}")
    @PreAuthorize("hasAuthority('DELETE')")
    public void deleteMovie(@PathVariable final int movie_id) {
        movieRepository.deleteById(movie_id);
    }
}
```

MovieShowController:

```
package com.MyMoviePlan.controller;

import com.MyMoviePlan.entity.BookingEntity;
import com.MyMoviePlan.entity.MovieShowsEntity;
import com.MyMoviePlan.exception.MovieShowNotFoundException;
import com.MyMoviePlan.model.BookedSeats;
import com.MyMoviePlan.repository.MovieShowsRepository;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@CrossOrigin
@RestController
@RequestMapping("/movie-show")
@AllArgsConstructor
public class MovieShowController {

  private final MovieShowsRepository repository;

  @PostMapping("add")
  @PreAuthorize("hasAuthority('WRITE')")
  public MovieShowsEntity save(@RequestBody MovieShowsEntity movieShow) {
    return repository.save(movieShow);
  }

  @GetMapping("up-coming")
  @PreAuthorize("hasAuthority('READ')")
  public List<MovieShowsEntity> upComing(@RequestParam(value = "records", required =
false) Optional<String> records) {
    if (records.isPresent())
      return repository.findFewUpComing(Integer.parseInt(records.get()));
    return repository.findAllUpComing();
  }

  @GetMapping("now-playing")
  public List<MovieShowsEntity> nowPlaying(@RequestParam(value = "records", required =
false) Optional<String> records) {
    if (records.isPresent())
      return repository.findFewNowPlaying(Integer.parseInt(records.get()));
    return repository.findAllNowPlaying();
  }
```

```
    @GetMapping("now-playing-up-coming")
    public List<MovieShowsEntity> nowPlayingAndUpComing() {
        return repository.findAllNowPlayingAndUpComing();
    }

    @GetMapping("not-playing")
    @PreAuthorize("hasAuthority('WRITE')")
    public List<MovieShowsEntity> notPlaying() {
        return repository.findAllNotPlaying();
    }

    @GetMapping("all")
    public List<MovieShowsEntity> findAllMovieShows() {
        return repository.findAll();
    }

    @GetMapping("{movie_show_id}")
    public MovieShowsEntity findMovieShowById(@PathVariable final int movie_show_id) {
        return repository.findById(movie_show_id)
                .orElseThrow(
                        () -> new MovieShowNotFoundException("Movie Show with id: " +
movie_show_id + " not found")
                );
    }

    @DeleteMapping("delete/{movie_show_id}")
    @PreAuthorize("hasAuthority('DELETE')")
    public void deleteMovieShow(@PathVariable final int movie_show_id) {
        repository.deleteById(movie_show_id);
    }


    /*
     *   ============================ Booking Controller
=========================
     */

    @GetMapping("{movie_show_id}/booked-seats/{on}")
    @PreAuthorize("hasAuthority('READ')")
    public BookedSeats bookedSeats(@PathVariable final int movie_show_id, @PathVariable
final String on) {
        final List<BookingEntity> bookings =
this.findMovieShowById(movie_show_id).getBookings()
                .stream().filter(m_show -> m_show.getDateOfBooking().toString().equals(on))
                .collect(Collectors.toList());

        int count = 0;
        List<String> seats = new ArrayList<>();
        for (BookingEntity booking : bookings) {
            count += booking.getTotalSeats();
            seats.addAll(booking.getSeatNumbers());
        }
        return new BookedSeats(count, seats);
    }
}
```

ShowController:

```
package com.MyMoviePlan.controller;

import com.MyMoviePlan.entity.BookingEntity;
import com.MyMoviePlan.entity.MovieShowsEntity;
import com.MyMoviePlan.entity.ShowEntity;
import com.MyMoviePlan.exception.BookingNotFoundException;
import com.MyMoviePlan.exception.MovieShowNotFoundException;
import com.MyMoviePlan.exception.ShowNotFoundException;
import com.MyMoviePlan.repository.BookingRepository;
import com.MyMoviePlan.repository.MovieRepository;
import com.MyMoviePlan.repository.MovieShowsRepository;
import com.MyMoviePlan.repository.ShowRepository;
import com.MyMoviePlan.service.UserService;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@CrossOrigin
@RestController
@RequestMapping("/show")
@AllArgsConstructor
public class ShowController {

    private final ShowRepository show;
    private final MovieShowsRepository movieShow;
    private final MovieRepository movie;
    private final UserService service;
    private final BookingRepository booking;

    @GetMapping("{show_id}")
    public ShowEntity findShowById(@PathVariable final int show_id) {
        return this.show.findById(show_id)
                .orElseThrow(() -> new ShowNotFoundException("Show with Id: " + show_id + " not
found"));
    }

    @GetMapping({"/", "all"})
    public List<ShowEntity> findAllShows() {
        return this.show.findAll();
    }

    @DeleteMapping("delete/{show_id}")
    @PreAuthorize("hasAuthority('DELETE')")
    public void deleteShow(@PathVariable final int show_id) {
        this.show.deleteById(show_id);
    }


    /*
     *   ============================== Movie Show Controller
===========================
     */

    @GetMapping("{show_id}/movie-show/all")
    public List<MovieShowsEntity> findAllMovieShows(@PathVariable final int show_id) {
        return this.findShowById(show_id)
                .getMovieShows();
    }

    @GetMapping("{show_id}/movie-show/{movie_show_id}")
    public MovieShowsEntity findMovieShowById(@PathVariable final int show_id,
                                              @PathVariable final int movie_show_id) {
        return this.findShowById(show_id)
                .getMovieShows()
                .stream()
                .filter(movie_show -> movie_show.getId() == movie_show_id)
                .findFirst()
                .orElseThrow(
                        () -> new MovieShowNotFoundException("Movie Show with id: "
                                + movie_show_id + " not found"));
    }

    @PostMapping("{show_id}/movie-show/add")
    @PreAuthorize("hasAuthority('WRITE')")
```

```
  public MovieShowsEntity saveMovieShow(@PathVariable final int show_id,
                                        @RequestBody final MovieShowsEntity movieShow) {
    final ShowEntity show = this.findShowById(show_id);
    final int movieId = movieShow.getMovieId();
    movieShow.setShow(show);
    movieShow.setMovieId(this.movie.findById(movieId).get().getId());
    return this.movieShow.save(movieShow);
  }

  @PutMapping("{show_id}/movie-show/update")
  @PreAuthorize("hasAuthority('UPDATE')")
  public MovieShowsEntity updateMovieShow(@PathVariable final int show_id,
                                          @RequestBody final MovieShowsEntity movieShow) {
    final ShowEntity show = this.findShowById(show_id);
    movieShow.setShow(show);
    return this.movieShow.save(movieShow);
  }

  @DeleteMapping("{show_id}/movie-show/delete/{movie_show_id}")
  @PreAuthorize("hasAuthority('UPDATE')")
  public void deleteMovieShow(@PathVariable final int show_id,
                              @PathVariable final int movie_show_id) {
    final MovieShowsEntity movieShow = this.findMovieShowById(show_id,
movie_show_id);
    this.movieShow.deleteById(movieShow.getMovieId());
  }

  /*
   *   ============================= Booking Controller
=========================
   */

  @GetMapping("{show_id}/movie-show/{movie_show_id}/booking/{booking_id}")
  @PreAuthorize("hasAuthority('READ')")
  public BookingEntity findBookingById(@PathVariable final int show_id,
                                       @PathVariable final int movie_show_id,
                                       @PathVariable final int booking_id) {
    final MovieShowsEntity movieShow = this.findMovieShowById(show_id,
movie_show_id);
    return movieShow.getBookings()
        .stream().filter(booking -> booking.getId() == booking_id)
        .findFirst()
        .orElseThrow(() -> new BookingNotFoundException("Booking with id: "
            + booking_id + " not found."));
  }

  @GetMapping("{show_id}/movie-show/{movie_show_id}/booking/all")
  @PreAuthorize("hasAuthority('READ')")
  public List<BookingEntity> allBookings(@PathVariable final int show_id,
                                         @PathVariable final int movie_show_id) {
    return this.findMovieShowById(show_id, movie_show_id).getBookings();
  }

  @PostMapping("{show_id}/movie-show/{movie_show_id}/booking/add")
  @PreAuthorize("hasAuthority('WRITE')")
  public BookingEntity saveBooking(@PathVariable final int show_id,
                                   @PathVariable final int movie_show_id,
                                   @RequestBody final BookingEntity booking) {
    final MovieShowsEntity moveShow = this.findMovieShowById(show_id,
movie_show_id);
//        booking.setUserId(this.service.getLoggedInUser().getId());
    booking.setUserId(this.service.findByMobile("8318152817").get().getId());
    booking.setMovieShow(moveShow);
    return this.booking.save(booking);
  }

  @PutMapping("{show_id}/movie-show/{movie_show_id}/booking/update")
  @PreAuthorize("hasAuthority('UPDATE')")
  public BookingEntity updateBooking(@PathVariable final int show_id,
                                     @PathVariable final int movie_show_id,
                                     @RequestBody final BookingEntity booking) {
    final MovieShowsEntity moveShow = this.findMovieShowById(show_id,
movie_show_id);
    booking.setMovieShow(moveShow);
    return this.booking.save(booking);
  }

  @DeleteMapping("{show_id}/movie-
show/{movie_show_id}/booking/delete/{booking_id}")
  @PreAuthorize("hasAuthority('READ')")
  public void deleteBookingById(@PathVariable final int show_id,
                                @PathVariable final int movie_show_id,
                                @PathVariable final int booking_id) {
    final BookingEntity booking = this.findBookingById(show_id, movie_show_id,
booking_id);
    this.booking.deleteById(booking.getId());
  }
}
```

UserController:

```
package com.MyMoviePlan.controller;

import com.MyMoviePlan.entity.UserEntity;
import com.MyMoviePlan.model.Credentials;
import com.MyMoviePlan.model.HttpResponse;
import com.MyMoviePlan.model.Token;
import com.MyMoviePlan.service.UserService;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.HttpServletRequest;
import java.util.List;

@CrossOrigin
@RestController
@RequestMapping("/user")
@AllArgsConstructor
public class UserController {

   private final UserService service;
   private final HttpServletRequest request;

   @GetMapping("/")
   public String index() {
      return "Welcome " + service.getUserName();
   }

   @PostMapping("authenticate")
   public Token authenticate(@RequestBody final Credentials credentials) {

      return service.authenticate(credentials);
   }

   @GetMapping("check/{username}")
   public Token checkUniqueness(@PathVariable final String username) {
      return service.checkUniqueness(username);
   }

   @GetMapping("get-user")
   @PreAuthorize("hasAuthority('READ')")
   public UserEntity user() {
      return service.getLoggedInUser()
```

```
            .setPassword(null);
    }

    @GetMapping("all")
    @PreAuthorize("hasAuthority('WRITE')")
    public List<UserEntity> allUsers() {
        return service.findAll();
    }

    @PutMapping("update/{username}")
    @PreAuthorize("hasAuthority('READ')")
    public UserEntity updateUser(@RequestBody final UserEntity userEntity,
                                 @PathVariable final String username) {

        return service.update(userEntity, username);
    }

    @PostMapping("sign-up")
    public HttpResponse signUp(@RequestBody final UserEntity userEntity) {

        return service.register(userEntity);
    }

    @PutMapping("change-password")
    @PreAuthorize("hasAuthority('READ')")
    public HttpResponse changePassword(@RequestBody final Credentials credentials) {

        return service.changePassword(credentials);
    }

    @PutMapping("forgot-password")
    public HttpResponse forgotPassword(@RequestBody final Credentials credentials) {
        return service.forgotPassword(credentials);
    }

    @DeleteMapping("delete/{username}")
    @PreAuthorize("hasAuthority('DELETE')")
    public HttpResponse delete(@PathVariable final String username) {
        return service.deleteById(username);
    }
}
```

Entity:
ActorEntity:

```
package com.MyMoviePlan.entity;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.*;

import javax.persistence.*;
import java.io.Serializable;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "actors")
public class ActorEntity implements Serializable {

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

   @Column(name = "is_cast")
   private String isCast;

   private String name;

   private String role;

   @Column(length = Integer.MAX_VALUE, columnDefinition="TEXT")
   private String image;

          @JsonIgnore
   @ToString.Exclude
   @EqualsAndHashCode.Exclude
   @ManyToOne(targetEntity = MovieEntity.class)
   private MovieEntity movie;

   public ActorEntity(String name, String role, String image) {
      this.name = name;
      this.role = role;
      this.image = image;
   }
}
```

AuditoriumEntity:

```
package com.MyMoviePlan.entity;

import lombok.*;

import javax.persistence.*;
import java.io.Serializable;
import java.util.List;

//@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class,
//        property = "id", scope = ShowEntity.class)
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
@Table(name = "auditoriums")
public class AuditoriumEntity implements Serializable {

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

    private String name;

    @Column(length = Integer.MAX_VALUE, columnDefinition="TEXT")
    private String image;

    private String email;

    @Column(name = "customer_care_no")
    private String customerCareNo;

    private String address;

    @Column(name = "seat_capacity")
    private int seatCapacity;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @ElementCollection
    @CollectionTable(name = "auditorium_facilities", joinColumns = @JoinColumn(name =
"auditorium_id"))
    @Column(name = "facility")
    private List<String> facilities;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @ElementCollection
    @CollectionTable(name = "auditorium_safeties", joinColumns = @JoinColumn(name =
"auditorium_id"))
    @Column(name = "safety")
    private List<String> safeties;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @JoinColumn(name = "auditorium_id", referencedColumnName = "id")
    @OneToMany(targetEntity = ShowEntity.class, cascade = CascadeType.ALL)
//    @JoinTable(name = "auditorium_shows",
//            joinColumns = @JoinColumn(name = "auditorium_id", unique = false),
//            inverseJoinColumns = @JoinColumn(name = "show_id", unique = false))
    private List<ShowEntity> shows;

    public AuditoriumEntity(String name, String image, String email, String customerCareNo, String
address,
                            int seatCapacity, List<String> facilities, List<String> safeties, List<ShowEntity> shows)
{
        this.name = name;
        this.image = image;
        this.email = email;
        this.customerCareNo = customerCareNo;
        this.address = address;
        this.seatCapacity = seatCapacity;
        this.facilities = facilities;
        this.safeties = safeties;
        this.shows = shows;
    }

    public AuditoriumEntity setId(int id) {
        this.id = id;
        return this;
    }

    public AuditoriumEntity setName(String name) {
        this.name = name;
        return this;
    }

    public AuditoriumEntity setImage(String image) {
        this.image = image;
        return this;
    }

    public AuditoriumEntity setEmail(String email) {
        this.email = email;
        return this;
    }
    public AuditoriumEntity setCustomerCare(String customerCareNo) {
        this.customerCareNo = customerCareNo;
        return this;
    }

    public AuditoriumEntity setAddress(String address) {
        this.address = address;
        return this;
    }

    public AuditoriumEntity setSeatCapacity(int seatCapacity) {
        this.seatCapacity = seatCapacity;
        return this;
    }

    public AuditoriumEntity setFacilities(List<String> facilities) {
        this.facilities = facilities;
        return this;
    }

    public AuditoriumEntity setSafeties(List<String> safeties) {
        this.safeties = safeties;
        return this;
    }

    public AuditoriumEntity setShows(List<ShowEntity> shows) {
        this.shows = shows;
        return this;
    }
}
```

BookingDetailsEntity:

```
package com.MyMoviePlan.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.io.Serializable;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "booking_details")
```

```
public class BookingDetailsEntity implements Serializable {

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

    @Column(name = "auditorium_id")
    private int auditoriumId;

    @Column(name = "show_id")
    private int showId;

    @Column(name = "movie_show_id")
    private int movieShowId;

    @Column(name = "movie_id")
    private int movieId;

    public BookingDetailsEntity(int auditoriumId, int showId, int movieShowId, int movieId) {
        this.auditoriumId = auditoriumId;
        this.showId = showId;
        this.movieShowId = movieShowId;
        this.movieId = movieId;
    }
}
```
BookingEntity:
```
package com.MyMoviePlan.entity;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.*;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.List;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "bookings")
public class BookingEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private double amount;

    @Column(name = "total_seats")
    private int totalSeats;

    @Column(name = "booked_on")
    @Temporal(TemporalType.DATE)
    private Date bookedOn;

    @Column(name = "date_of_booking")
    @Temporal(TemporalType.DATE)
    private Date dateOfBooking;

    @Column(name = "user_id")
    private String userId;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @ElementCollection
    @CollectionTable(name = "booked_seats", joinColumns = @JoinColumn(name = "booking_id"))
    @Column(name = "seat_numbers")
    private List<String> seatNumbers;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @OneToOne(targetEntity = PaymentEntity.class, cascade = CascadeType.ALL)
    @JoinColumn(name = "payment_id")
    private PaymentEntity payment;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @OneToOne(targetEntity = BookingDetailsEntity.class, cascade = CascadeType.ALL)
    @JoinColumn(name = "booking_details_id")
    private BookingDetailsEntity bookingDetails;

    @JsonIgnore
    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @ManyToOne(targetEntity = MovieShowsEntity.class)
    private MovieShowsEntity movieShow;

    public BookingEntity(double amount, int totalSeats, Date bookedOn, Date dateOfBooking,
List<String> seatNumbers,
                    PaymentEntity payment, String userId, MovieShowsEntity movieShow) {
        this.amount = amount;
        this.totalSeats = totalSeats;
        this.bookedOn = bookedOn;
        this.dateOfBooking = dateOfBooking;
        this.seatNumbers = seatNumbers;
        this.payment = payment;
        this.userId = userId;
        this.movieShow = movieShow;
    }

    public BookingEntity setMovieShow(MovieShowsEntity movieShow) {
        this.movieShow = movieShow;
        return this;
    }

    public BookingEntity setId(int id) {
        this.id = id;
        return this;
    }

    public BookingEntity setAmount(double amount) {
        this.amount = amount;
        return this;
    }

    public BookingEntity setTotalSeats(int totalSeats) {
        this.totalSeats = totalSeats;
        return this;
    }

    public BookingEntity setStatus(Date bookedOn) {
        this.bookedOn = bookedOn;
        return this;
    }

    public BookingEntity setDateOfBooking(Date dateOfBooking) {
        this.dateOfBooking = dateOfBooking;
        return this;
    }

    public BookingEntity setSeatNumbers(List<String> seatNumbers) {
        this.seatNumbers = seatNumbers;
        return this;
    }

    public BookingEntity setPayment(PaymentEntity payment) {
        this.payment = payment;
        return this;
    }

    public BookingEntity setUserId(String userId) {
        this.userId = userId;
        return this;
    }
}
```
MovieEntity:
```
package com.MyMoviePlan.entity;

import lombok.*;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.List;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "movies")
public class MovieEntity implements Serializable {

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

    private String name;

    @Column(length = Integer.MAX_VALUE, columnDefinition = "TEXT")
    private String image;

    @Column(name = "bg_image", length = Integer.MAX_VALUE, columnDefinition="TEXT")
    private String bgImage;

    @Column(length = 9000)
    private String story;

    private String year;

    private String duration;

    private String caption;

    @Column(name = "added_on")
    @Temporal(TemporalType.DATE)
    private Date addedOn;

    @Temporal(TemporalType.DATE)
    private Date release;
private String language;

@ToString.Exclude
@EqualsAndHashCode.Exclude
@ElementCollection
@CollectionTable(name = "movie_genres", joinColumns = @JoinColumn(name = "movie_id"))
@Column(name = "genre")
private List<String> genres;

@ToString.Exclude
@EqualsAndHashCode.Exclude
@OneToMany(targetEntity = ActorEntity.class, cascade = CascadeType.ALL)
@JoinColumn(name = "movie_id", referencedColumnName = "id")
private List<ActorEntity> casts;

@ToString.Exclude
@EqualsAndHashCode.Exclude
@OneToMany(targetEntity = ActorEntity.class, cascade = CascadeType.ALL)
@JoinColumn(name = "movie_id", referencedColumnName = "id")
private List<ActorEntity> crews;

public MovieEntity(String name, String image, String bgImage, String story, String year,
            String duration, String caption, Date addedOn, Date release, String language,
            List<String> genres, List<ActorEntity> casts, List<ActorEntity> crews) {
  this.name = name;
  this.image = image;
  this.bgImage = bgImage;
  this.story = story;
  this.year = year;
  this.duration = duration;
  this.caption = caption;
  this.addedOn = addedOn;
  this.release = release;
  this.language = language;
  this.genres = genres;
  this.casts = casts;
  this.crews = crews;
}

public MovieEntity setId(int id) {
  this.id = id;
  return this;
}

public MovieEntity setName(String name) {
  this.name = name;
  return this;
```

```
}

public MovieEntity setImage(String image) {
   this.image = image;
   return this;
}

public MovieEntity setBgImage(String bgImage) {
   this.bgImage = bgImage;
   return this;
}

public MovieEntity setStory(String story) {
   this.story = story;
   return this;
}

public MovieEntity setYear(String year) {
   this.year = year;
   return this;
}

public MovieEntity setDuration(String duration) {
   this.duration = duration;
   return this;
}

public MovieEntity setCaption(String caption) {
   this.caption = caption;
   return this;
}

public MovieEntity setAddedOn(Date addedOn) {
   this.addedOn = addedOn;
   return this;
}

public MovieEntity setRelease(Date release) {
   this.release = release;
   return this;
}

public MovieEntity setLanguages(String language) {
   this.language = language;
   return this;
}

public MovieEntity setGenres(List<String> genres) {
        this.genres = genres;
        return this;
    }

    public MovieEntity setCasts(List<ActorEntity> casts) {
        this.casts = casts;
        return this;
    }

    public MovieEntity setCrews(List<ActorEntity> crews) {
        this.crews = crews;
        return this;
    }
}
```

MovieShowsEntity:

```
package com.MyMoviePlan.entity;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.*;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.List;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "movie_shows")
public class MovieShowsEntity implements Serializable {

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

    @Temporal(TemporalType.DATE)
    @Column(name = "show_start")
    private Date start;

    @Temporal(TemporalType.DATE)
    @Column(name = "show_end")
    private Date end;

    @Column(name = "movie_id")
    private int movieId;
    @JsonIgnore
    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @ManyToOne(targetEntity = ShowEntity.class)
    private ShowEntity show;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @JoinColumn(name = "movie_show_id", referencedColumnName = "id")
    @OneToMany(targetEntity = BookingEntity.class, cascade = CascadeType.ALL)
//    @JoinTable(name = "movie_show_bookings",
//            joinColumns = @JoinColumn(name = "movie_show_id", unique = false),
//            inverseJoinColumns = @JoinColumn(name = "booking_id", unique = false))
    private List<BookingEntity> bookings;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @OneToOne(targetEntity = PriceEntity.class, cascade = CascadeType.ALL)
    @JoinColumn(name = "price_id")
    private PriceEntity price;

    public MovieShowsEntity(int id, Date start, Date end, List<BookingEntity> bookings, int movieId) {
        this.id  = id;
        this.start = start;
        this.end = end;
        this.bookings = bookings;
        this.movieId = movieId;
    }

    public MovieShowsEntity setId(int id) {
        this.id = id;
        return this;
    }

    public MovieShowsEntity setStart(Date start) {
        this.start = start;
        return this;
    }

    public MovieShowsEntity setEnd(Date end) {
        this.end = end;
        return this;
    }

    public MovieShowsEntity setShow(ShowEntity show) {
        this.show = show;
        return this;
    }
    public MovieShowsEntity setMovieId(int movieId) {
        this.movieId = movieId;
        return this;
    }
}
```

PaymentEntity:

```
package com.MyMoviePlan.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "payments")
public class PaymentEntity implements Serializable {

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

    private double amount;

    @Column(name = "payment_date")
    @Temporal(TemporalType.DATE)
    private Date paymentDate;

    @Column(name = "card_number", length = 20)
    private String cardNumber;

    @Column(name = "card_expiry_month", length = 5)
    private String cardExpiryMonth;

    @Column(name = "card_expiry_year", length = 5)
    private String cardExpiryYear;

    @Column(name = "card_cvv", length = 5)
    private String cardCVV;
    public PaymentEntity(double amount, Date paymentDate, String cardNumber, String
cardExpiryMonth,
                         String cardExpiryYear, String cardCVV) {
        this.amount = amount;
        this.paymentDate = paymentDate;
        this.cardNumber = cardNumber;
        this.cardExpiryMonth = cardExpiryMonth;
        this.cardExpiryYear = cardExpiryYear;
        this.cardCVV = cardCVV;
    }

    public PaymentEntity setId(int id) {
        this.id = id;
        return this;
    }

    public PaymentEntity setAmount(double amount) {
        this.amount = amount;
        return this;
    }

    public PaymentEntity setPaymentDate(Date paymentDate) {
        this.paymentDate = paymentDate;
        return this;
    }

    public PaymentEntity setCardNumber(String cardNumber) {
        this.cardNumber = cardNumber;
        return this;
    }

    public PaymentEntity setCardExpiryMonth(String cardExpiryMonth) {
        this.cardExpiryMonth = cardExpiryMonth;
        return this;
    }

    public PaymentEntity setCardExpiryYear(String cardExpiryYear) {
        this.cardExpiryYear = cardExpiryYear;
        return this;
    }

    public PaymentEntity setCardCVV(String cardCVV) {
        this.cardCVV = cardCVV;
        return this;
    }
}
```
PriceEntity:
```
package com.MyMoviePlan.entity;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.io.Serializable;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "prices")
public class PriceEntity implements Serializable {

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

   private double general;

   private double silver;

   private double gold;

   public PriceEntity(double general, double silver, double gold) {
      this.general = general;
      this.silver = silver;
      this.gold = gold;
   }
}
```
ShowEntity:
```
package com.MyMoviePlan.entity;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.*;

import javax.persistence.*;
import java.io.Serializable;
import java.util.List;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
```

```
@Table(name = "shows")
public class ShowEntity implements Serializable {

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

   private String name;

   @Column(name = "start_time")
   private String startTime;

   @JsonIgnore
   @ToString.Exclude
   @EqualsAndHashCode.Exclude
   @ManyToOne(targetEntity = AuditoriumEntity.class)
   private AuditoriumEntity auditorium;

   //    @JsonManagedReference
   @ToString.Exclude
   @EqualsAndHashCode.Exclude
   @OneToMany(targetEntity = MovieShowsEntity.class, cascade = CascadeType.ALL)
   @JoinColumn(name = "show_id", referencedColumnName = "id")
   private List<MovieShowsEntity> movieShows;

   public ShowEntity(String name, String startTime, List<MovieShowsEntity> movieShows) {
      this.name = name;
      this.startTime = startTime;
      this.movieShows = movieShows;
   }

   public ShowEntity setId(int id) {
      this.id = id;
      return this;
   }

   public ShowEntity setName(String name) {
      this.name = name;
      return this;
   }

   public ShowEntity setStartTime(String startTime) {
      this.startTime = startTime;
      return this;
   }

   public ShowEntity setAuditorium(AuditoriumEntity auditorium) {
      this.auditorium = auditorium;
```

```
        return this;
    }

    public ShowEntity setMovieShows(List<MovieShowsEntity> movieShows) {
        this.movieShows = movieShows;
        return this;
    }
}
```
UserEntity:
```
package com.MyMoviePlan.entity;

import com.MyMoviePlan.model.UserRole;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import java.io.Serializable;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@Table(name = "users")
public class UserEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY, generator = "uuid2")
    @GenericGenerator(name = "uuid2", strategy = "uuid2")
    private String id;

    @Column(length = 50)
    private String name;

    @Column(nullable = false, length = 50, unique = true)
    private String email;

    @Column(nullable = false, length = 10, unique = true)
    private String mobile;

    @Column(length = 60)
    private String gender;

    private String password;
    private Boolean terms;

    @Column(name = "is_account_non_expired")
    private Boolean isAccountNonExpired;

    @Column(name = "is_account_non_locked")
    private Boolean isAccountNonLocked;

    @Column(name = "is_credentials_non_expired")
    private Boolean isCredentialsNonExpired;

    @Column(name = "is_enabled")
    private Boolean isEnabled;

    @Column(name = "user_role", length = 20)
    @Enumerated(EnumType.STRING)
    private UserRole userRole;

    public UserEntity(String name, String email, String mobile, String gender, String password, Boolean
terms,
                      Boolean isAccountNonExpired, Boolean isAccountNonLocked,
                      Boolean isCredentialsNonExpired, Boolean isEnabled, UserRole userRole) {
        this.name = name;
        this.email = email;
        this.mobile = mobile;
        this.gender = gender;
        this.password = password;
        this.terms = terms;
        this.isAccountNonExpired = isAccountNonExpired;
        this.isAccountNonLocked = isAccountNonLocked;
        this.isCredentialsNonExpired = isCredentialsNonExpired;
        this.isEnabled = isEnabled;
        this.userRole = userRole;
    }

    public UserEntity setId(String id) {
        this.id = id;
        return this;
    }

    public UserEntity setName(String name) {
        this.name = name;
        return this;
    }

    public UserEntity setEmail(String email) {
        this.email = email;
        return this;
```

```
}

public UserEntity setMobile(String mobile) {
   this.mobile = mobile;
   return this;
}

public UserEntity setGender(String gender) {
   this.gender = gender;
   return this;
}

public UserEntity setPassword(String password) {
   this.password = password;
   return this;
}

public UserEntity setActive(Boolean active) {
   terms = active;
   return this;
}

public UserEntity setAccountNonExpired(Boolean accountNonExpired) {
   isAccountNonExpired = accountNonExpired;
   return this;
}

public UserEntity setAccountNonLocked(Boolean accountNonLocked) {
   isAccountNonLocked = accountNonLocked;
   return this;
}

public UserEntity setCredentialsNonExpired(Boolean credentialsNonExpired) {
   isCredentialsNonExpired = credentialsNonExpired;
   return this;
}

public UserEntity setEnabled(Boolean enabled) {
   isEnabled = enabled;
   return this;
}

public UserEntity setUserRole(UserRole userRole) {
   this.userRole = userRole;
   return this;
}

public UserEntity setTerms(Boolean terms) {
        this.terms = terms;
        return this;
    }
}
```

Filter:
JWTFilter:

```
package com.MyMoviePlan.filter;

import com.MyMoviePlan.model.HttpResponse;
import com.MyMoviePlan.security.ApplicationUserDetailsService;
import com.MyMoviePlan.util.JWTUtil;
import io.jsonwebtoken.JwtException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component()
public class JWTFilter extends OncePerRequestFilter {

    @Autowired
    private JWTUtil jwtUtil;

    @Autowired
    private ApplicationUserDetailsService userDetailsService;

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

        try {
            String authorization = request.getHeader("Authorization");
            String token = null;
            String userName = null;

            if (authorization != null && authorization.startsWith("Bearer ")) {
```

```
            token = authorization.substring(7);
            userName = jwtUtil.getUsernameFromToken(token);
        }

        if (userName != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails
                    = userDetailsService.loadUserByUsername(userName);

            if (jwtUtil.validateToken(token, userDetails)) {
                UsernamePasswordAuthenticationToken authenticationToken
                        = new UsernamePasswordAuthenticationToken(userDetails,
                        null, userDetails.getAuthorities());

                authenticationToken.setDetails(
                        new WebAuthenticationDetailsSource().buildDetails(request)
                );
                SecurityContextHolder.getContext().setAuthentication(authenticationToken);
            }
        }
        filterChain.doFilter(request, response);
    } catch (JwtException exception) {
        setErrorResponse(HttpStatus.NOT_ACCEPTABLE, response, exception);
    } catch (Exception exception) {
        setErrorResponse(HttpStatus.INTERNAL_SERVER_ERROR, response, exception);
    }
  }

  private void setErrorResponse(HttpStatus status, HttpServletResponse response, Exception
exception) {
    response.setStatus(status.value());
    response.setContentType("application/json");
    final HttpResponse httpResponse =
        new HttpResponse(status.value(),
            HttpStatus.valueOf(status.value()).getReasonPhrase(),
            exception.getMessage());
    try {
        final String json = httpResponse.covertToJson();
        response.getWriter().write(json);
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Service:
BookingDetailsRepository:

```
package com.MyMoviePlan.service;
```

```
import com.MyMoviePlan.entity.BookingDetailsEntity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookingDetailsRepository extends JpaRepository<BookingDetailsEntity, Integer> {
}
```

UserService:

```
package com.MyMoviePlan.service;

import com.MyMoviePlan.entity.UserEntity;
import com.MyMoviePlan.exception.UnAuthorizedException;
import com.MyMoviePlan.exception.UserNotFoundException;
import com.MyMoviePlan.model.Credentials;
import com.MyMoviePlan.model.HttpResponse;
import com.MyMoviePlan.model.Token;
import com.MyMoviePlan.repository.UserRepository;
import com.MyMoviePlan.util.JWTUtil;
import lombok.AllArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.servlet.http.HttpServletRequest;
import java.util.List;
import java.util.Optional;
import java.util.regex.Pattern;

import static com.MyMoviePlan.model.UserRole.*;

@Service
@Transactional
@AllArgsConstructor
public class UserService {

  private final JWTUtil jwtUtil;
  private final UserRepository repository;
  private final HttpServletRequest request;
  private final PasswordEncoder passwordEncoder;
  private final AuthenticationManager authenticationManager;

  public Optional<UserEntity> findByEmail(final String email) {
    return repository.findByEmail(email);
    }

    public Optional<UserEntity> findByMobile(final String mobile) {
        return repository.findByMobile(mobile);
    }

    public Optional<UserEntity> findById(final String id) {
        return repository.findById(id);
    }

    public UserEntity findByUserName(final String username) {

        final UserEntity user = this.getUser(username);
        if (authorizeUser(user, getUserName()))
            return user;
        else
            throw new UnAuthorizedException("You are not authorized access this account");
    }

    public List<UserEntity> findAll() {
        return repository.findAll();
    }

    public UserEntity save(final UserEntity user) {
        return repository.save(user);
    }

    public UserEntity update(final UserEntity userEntity, final String username) {
        final UserEntity user = this.getUser(username);

        if (!authorizeUser(user, getUserName()))
            throw new UnAuthorizedException("You are not authorized access this account");

        userEntity.setName(isNullOrEmpty(userEntity.getName()) ? user.getName() :
userEntity.getName())
                .setEmail(isNullOrEmpty(userEntity.getEmail()) ? user.getEmail() : userEntity.getEmail())
                .setMobile(isNullOrEmpty(userEntity.getMobile()) ? user.getMobile() :
userEntity.getMobile())
                .setTerms(user.getTerms())
                .setPassword(user.getPassword())
                .setUserRole(user.getUserRole())
                .setAccountNonExpired(user.getIsAccountNonExpired())
                .setAccountNonLocked(user.getIsAccountNonLocked())
                .setCredentialsNonExpired(user.getIsCredentialsNonExpired())
                .setEnabled(user.getIsEnabled());

        return repository.save(userEntity);
    }
```

```
  public HttpResponse register(final UserEntity user) {

    if (user.getEmail().contains("super"))
      user.setUserRole(ROLE_SUPER_ADMIN);
    else if (user.getEmail().contains("admin"))
      user.setUserRole(ROLE_ADMIN);
    else
      user.setUserRole(ROLE_USER);

    user.setAccountNonExpired(true)
        .setAccountNonLocked(true)
        .setCredentialsNonExpired(true)
        .setEnabled(true)
        .setPassword(this.passwordEncoder.encode(user.getPassword()));
    try {
      this.save(user);
    } catch (Exception e) {
      throw new RuntimeException("SQL Unique key constrains volition");
    }
    return new HttpResponse(HttpStatus.OK.value(), HttpStatus.OK.getReasonPhrase(), "Your
account is created");
  }

  public HttpResponse deleteById(final String username) {
    final UserEntity user = this.getUser(username);
    repository.deleteById(user.getId());
    return new HttpResponse(HttpStatus.OK.value(), HttpStatus.OK.getReasonPhrase(), "Your
account is deleted");
  }

  public HttpResponse forgotPassword(final Credentials credentials) {
    final UserEntity user = this.getUser(credentials.getUsername());

    user.setPassword(this.passwordEncoder.encode(credentials.getPassword()));
    this.save(user);
    return new HttpResponse(HttpStatus.OK.value(), HttpStatus.OK.getReasonPhrase(), "Your
password reset successfully");
  }

  public HttpResponse changePassword(final Credentials credentials) {

    final UserEntity user = this.getUser(credentials.getUsername());

    if (!this.authorizeUser(user, this.getUserName()))
      throw new UnAuthorizedException("You are not authorized access this account");

    user.setPassword(this.passwordEncoder.encode(credentials.getPassword()));
```

```
        this.save(user);
        return new HttpResponse(HttpStatus.OK.value(), HttpStatus.OK.getReasonPhrase(), "Your
password changed successfully");
    }

    public Token authenticate(final Credentials credentials) {
        try {
            authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            credentials.getUsername(),
                            credentials.getPassword()
                    )
            );
        } catch (BadCredentialsException e) {
            throw new BadCredentialsException("Invalid username or password");
        }

        final String token = jwtUtil.generateToken(credentials.getUsername());
        return new Token(token);
    }

    public Token checkUniqueness(final String username) {
        final String regex = "^[A-Za-z0-9+_.-]+@(.+)$";
        Optional<UserEntity> user = null;
        if (Pattern.matches(regex, username))
            user = findByEmail(username);
        else if (username.contains("-") && username.length() > 10)
            user = findById(username);
        else
            user = findByMobile(username);
        return user.isPresent() ? new Token(username) : new Token(null);
    }

    public UserEntity getUser(final String idOrEmailOrMobile) {
        final String regex = "^[A-Za-z0-9+_.-]+@(.+)$";
        Optional<UserEntity> user = null;
        if (Pattern.matches(regex, idOrEmailOrMobile))
            user = findByEmail(idOrEmailOrMobile);
        else if (idOrEmailOrMobile.contains("-") && idOrEmailOrMobile.length() > 10)
            user = findById(idOrEmailOrMobile);
        else
            user = findByMobile(idOrEmailOrMobile);
        return user
                .orElseThrow(() -> new UserNotFoundException("User: '" + idOrEmailOrMobile + "' not
found"));
    }

    public String getUserName() {
        final String token = request.getHeader("Authorization");
        return jwtUtil.getUserName(token);
    }

    public UserEntity getLoggedInUser() {
        return getUser(getUserName());
    }

    private boolean authorizeUser(final UserEntity user, final String username) {
        if (user.getEmail().equals(username) || user.getId().equals(username) ||
user.getMobile().equals(username))
            return true;
        else
            return user.getUserRole().equals(ROLE_ADMIN) ||
user.getUserRole().equals(ROLE_SUPER_ADMIN);

    }

    private boolean isNullOrEmpty(String value) {
        return value == null || value.equals(" ") || value.equals(null) ? true : false;
    }
}
```

Util:
BeanSupplier:

```
package com.MyMoviePlan.util;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class BeanSupplier {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(10);
    }

//    @Bean
//    public FilterRegistrationBean corsFilter() {
//        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
//        CorsConfiguration config = new CorsConfiguration();
//        config.setAllowCredentials(Boolean.TRUE);
//        config.addAllowedOrigin(CorsConfiguration.ALL);
//        config.addAllowedHeader(CorsConfiguration.ALL);
//        config.addAllowedMethod(CorsConfiguration.ALL);
//       source.registerCorsConfiguration("/**", config);
//       FilterRegistrationBean bean = new FilterRegistrationBean();
//       bean.setFilter(new CorsFilter());
//       bean.setOrder(0);
//       return bean;
//    }
}
```

JWTUtil:

```
package com.MyMoviePlan.util;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

import java.io.Serializable;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Component
public class JWTUtil implements Serializable {

    public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;
    private static final long serialVersionUID = 234234523523L;

    @Value("${jwt.secret}")
    private String secretKey;

    //retrieve username from jwt token
    public String getUsernameFromToken(final String token) {
        return getClaimFromToken(token, Claims::getSubject);
    }

    //retrieve expiration date from jwt token
    private Date getExpirationDateFromToken(final String token) {
        return getClaimFromToken(token, Claims::getExpiration);
    }

    private <T> T getClaimFromToken(final String token, final Function<Claims, T> claimsResolver) {
        final Claims claims = getAllClaimsFromToken(token);
        return claimsResolver.apply(claims);
    }
    //for retrieving any information from token we will need the secret key
    private Claims getAllClaimsFromToken(final String token) {
        Claims claims = null;
        try {
            claims = Jwts.parser()
                    .setSigningKey(secretKey)
                    .parseClaimsJws(token)
                    .getBody();
        } catch (JwtException exception) {
            throw new JwtException("Invalid Token");
        }
        return claims;
    }

    //check if the token has expired
    private Boolean isTokenExpired(final String token) {
        final Date expiration = getExpirationDateFromToken(token);
        return expiration.before(new Date());
    }

    //generate token for user
//    public String generateToken(UserDetails userDetails) {
//        Map<String, Object> claims = new HashMap<>();
//        return doGenerateToken(claims, userDetails.getUsername());
//    }

    public String generateToken(final String username) {
        Map<String, Object> claims = new HashMap<>();
        return doGenerateToken(claims, username);
    }

    //while creating the token -
    //1. Define  claims of the token, like Issuer, Expiration, Subject, and the ID
    //2. Sign the JWT using the HS512 algorithm and secret key.
    private String doGenerateToken(final Map<String, Object> claims, final String username) {
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(username)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY * 1000))
                .signWith(SignatureAlgorithm.HS512, secretKey)
                .compact();
    }

    //validate token
    public Boolean validateToken(final String token, final UserDetails userDetails) {
        final String username = getUsernameFromToken(token);
        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }

    public String getUserName(final String header) {
        return getUsernameFromToken(header.substring(7));
    }
}
```