What2See

FRANCESCO BALDUCCI

Politecnico di Milano

Documentazione di Progetto – Prova finale di Ingegneria del Software

francesco.balducci@mail.polimi.it

A.A. 2022/2023

INDICE

PROBLEMA *

L'idea nasce dall'assenza quasi totale di siti web che consentano, a turisti in visita a una città, di informarsi, valutare nel dettaglio e confrontare itinerari turistici (a tema culturale, panoramico, naturale, ecc...). Inoltre, in quanto personalmente appassionato nel conoscere nuove località, mi piacerebbe avere una piattaforma tramite la quale trovare persone interessate a farlo conoscere.

Vorrei dunque creare una piattaforma web per mettere in contatto i turisti in visita con le comunità locali che conoscono e promuovono il territorio locale. Intendo coinvolgere diversi attori tra cui: le guide, che propongono dei tour con tappe interessanti al loro interno; i turisti, che ricercano il loro tour ideale specificando i suoi vari aspetti (quali città, durata, tema, tags, ecc...); i moderatori, che verificano il corretto utilizzo della piattaforma (tour ingannevoli o truffe, ecc...).

Un turista, per utilizzare la piattaforma, deve essere autenticato. Questi può ricercare tra i tour disponibili e consultare ciascuno nel dettaglio (tappe con i relativi dettagli, durata, costo, recensioni, ecc.). I risultati della ricerca sono ordinati relativamente al numero di valutazioni e per quelli più segnati come percorsi dagli utenti. Egli ha, infatti, la possibilità di marcare i tour a cui ha preso parte, in modo da avere un rapido resoconto delle proprie esperienze in una pagina dedicata. Può inoltre recensire e segnalare i tour, e scambiare messaggi con le guide in caso di necessità di maggiori informazioni.

Una guida, previa registrazione, può creare nuovi tour, specificando tutte le informazioni che li caratterizzano, oppure modificare o eliminare quelli già creati. Può gestirne la visibilità: pubblico per comparire nei risultati di ricerca oppure condiviso solo con utenti specifici. È in grado di scambiare messaggi con gli utenti che hanno fatto richiesta di informazioni.

Un moderatore può invece rimuovere i tour segnalati in caso violino le norme della piattaforma.

ANALISI DEI REQUISITI *

Per risolvere la problematica presentata, ho individuato le seguenti macro-funzionalità da implementare:

1. GESTIONE UTENZA
    a. Registrazione (turisti e guide);
    b. Accesso (turisti, guide e moderatori);
2. CRUD TOUR
    a. Creazione e modifica (solo guida) di titolo, descrizione, visibilità (pubblico o specifici turisti), città, tema, tags, durata complessiva, costo complessivo, tappe (ciascuna con titolo, descrizione, costo, durata e dettagli sullo spostamento);
    b. Visualizzazione tour creati e possibilità di eliminazione (solo guida) su pagina personale;
    c. Visualizzazione tour condivisi con il proprio utente e completati (solo turista) su pagina personale;
    d. Ricerca tramite città, durata, tema, tags. I tour che rispettano tali filtri vengono mostrati come risultati in forma sintetica, ordinati per quelli con più recensioni e, secondariamente, per quelli più segnati come percorsi dai turisti. Nel caso della guida e dell'amministratore, inoltre, vengono rispettivamente mostrati per primi quelli di cui si è autori e quelli con più segnalazioni;
3. INTERAZIONE TOUR
    a. Marcatura come percorso (solo turista);
    b. Recensione (solo turista che ha segnato come percorso) con valutazione in stelle e breve descrizione;
    c. Segnalazione (solo turista) ai moderatori, con una breve descrizione, nel caso contenga informazioni errate o inadatte alla piattaforma;

4. INTERAZIONE UTENZA
    a. Scambio di messaggi (deve partire dal turista nel contesto di una pagina tour, poiché non esiste un modo per la guida di entrare a conoscenza di un turista) tra utenti che hanno intrapreso una conversazione;
5. MODERAZIONE
    a. Consultazione delle segnalazioni (solo moderatore) ricevute;
    b. Eliminazione di un tour segnalato o meno (solo moderatore);

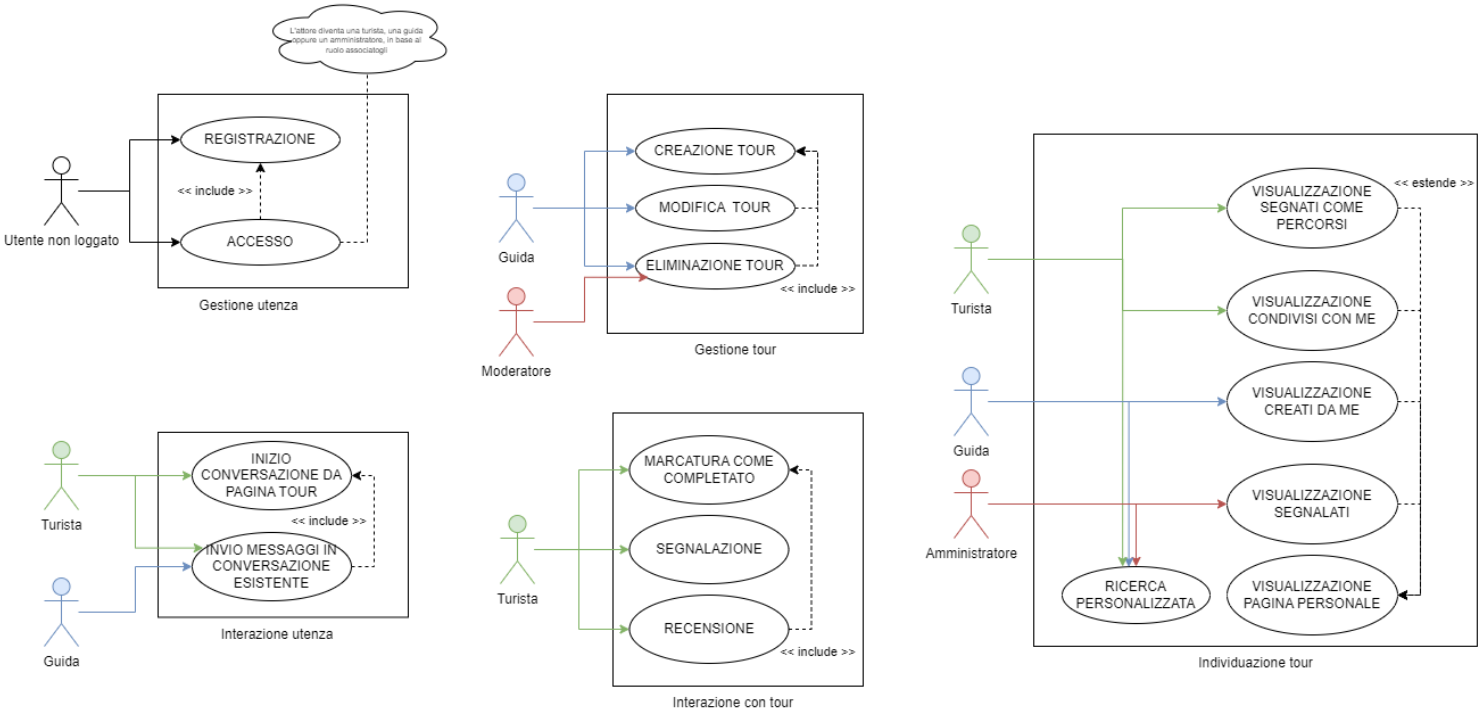
È emerso di conseguenza che le principali entità da creare avrebbero dovuto essere:

- Turisti, guide, amministratori (in generale utenti);
- Città;
- Tema;
- Tag;
- Tour e relative tappe;
- Recensione;
- Segnalazione;
- Conversazione e relativi messaggi;

PROGETTAZIONE *

UML - Use case diagram

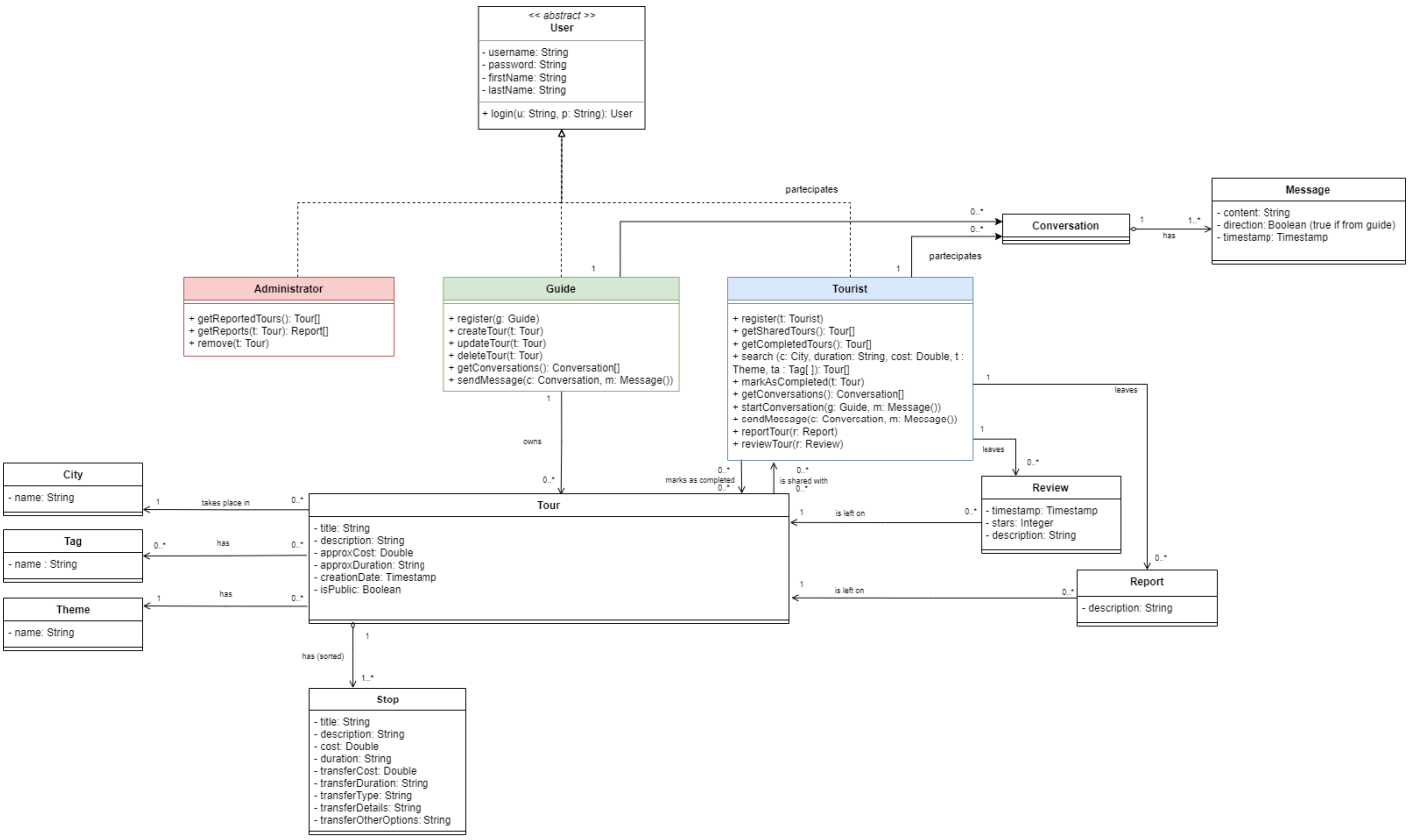
Per formalizzare più nel dettaglio le azioni intraprendibili dagli utenti, segue un diagramma con i casi d'uso principali:



Come indicato nella nuvoletta legata all'accesso, l'attore "utente non loggato" confluisce in uno dei tre ruoli specifici in seguito alla registrazione o all'accesso. Non è invece possibile utilizzare la piattaforma senza previo accesso.

UML - Class diagram

Le entità del dominio applicativo, con le loro proprietà, metodi e interdipendenze sono rappresentate nel seguente diagramma delle classi:



L'intento del diagramma è fornire un'idea generale delle entità e delle loro relazioni. Tale rappresentazione si evolverà successivamente e in linea con le necessità implementative di ciascuna side (back-end e front-end).

Sono state omesse per brevità le proprietà identificative, quelle che concretizzano le relazioni, getters, setters, costruttori, ecc.

La punta della freccia nelle relazioni indica unicamente la direzione verso la quale interpretare il verbo.

Nella relazione tra turista e segnalazione o recensione, il lato molti è da interpretare come condizionato alla distinzione di identificativo tour. Infatti, questi può segnalare, recensire e marcare come completati più tour, ma soltanto una volta per ciascuno.

Version control (Git)

Per organizzare al meglio il processo di sviluppo, ho scelto di utilizzare il software Git per il versioning control. Esso consente di raccogliere un gruppo di modifiche ai sorgenti in un "commit" (un'unità minima di aggiornamento). Dopo un certo numero di commit si esegue un "push" verso un servizio online che mantiene una copia della cronologia. In particolare, ho optato per GitHub, avendo già provato ad utilizzare Azure DevOps in passato.

Nei progetti più complessi e con più collaboratori, si preferisce strutturare ulteriormente le modifiche in "branch": ciascuna feature ne ha una associata e, una volta che questa raggiunge un certo grado di maturità del codice, si procede con una "pull request" verso la branch principale. La richiesta viene analizzata, commentata, modificata ed infine eventualmente approvata dagli sviluppatori con più esperienza.

Repository back-end - Repository front-end

Strategia di comunicazione tra le parti (API)

La comunicazione tra i due lati dell'applicazione è stata messa a punto attraverso le API ("application programming interface"). Si tratta di una metodologia estremamente utilizzata e che ha sostituito le infrastrutture monolitiche. Viene stabilito un contratto sull'interfacciamento che entrambe le parti devono rispettare, in modo che l'implementazione di ciascuna si possa evolvere fintanto che si rispetta tale accordo. Un esempio di contratto per una API di login può consistere in:

- URL (/api/user)
- Method (POST)
- Content-type (JSON)
- Query params (nessuno)
- Request body ({ 'username: 'user', 'password': '123' })
- Response body ({ 'id': 1, 'firstName': 'nome', ... })

SVILUPPO BACK-END

Panoramica

Per lo sviluppo back-end ho scelto di utilizzare il framework Spring Boot (versione 3) di Java (versione 17) e il DBMS MySQL. Il framework consente di sfruttare rapidamente diversi costrutti attualmente standard di fatto, sfruttando best-practice offerte da una vastissima comunità di sviluppatori open-source.

È estendibile con ulteriori plugin gestiti dalla piattaforma Maven, tra i quali ho scelto, ad esempio:

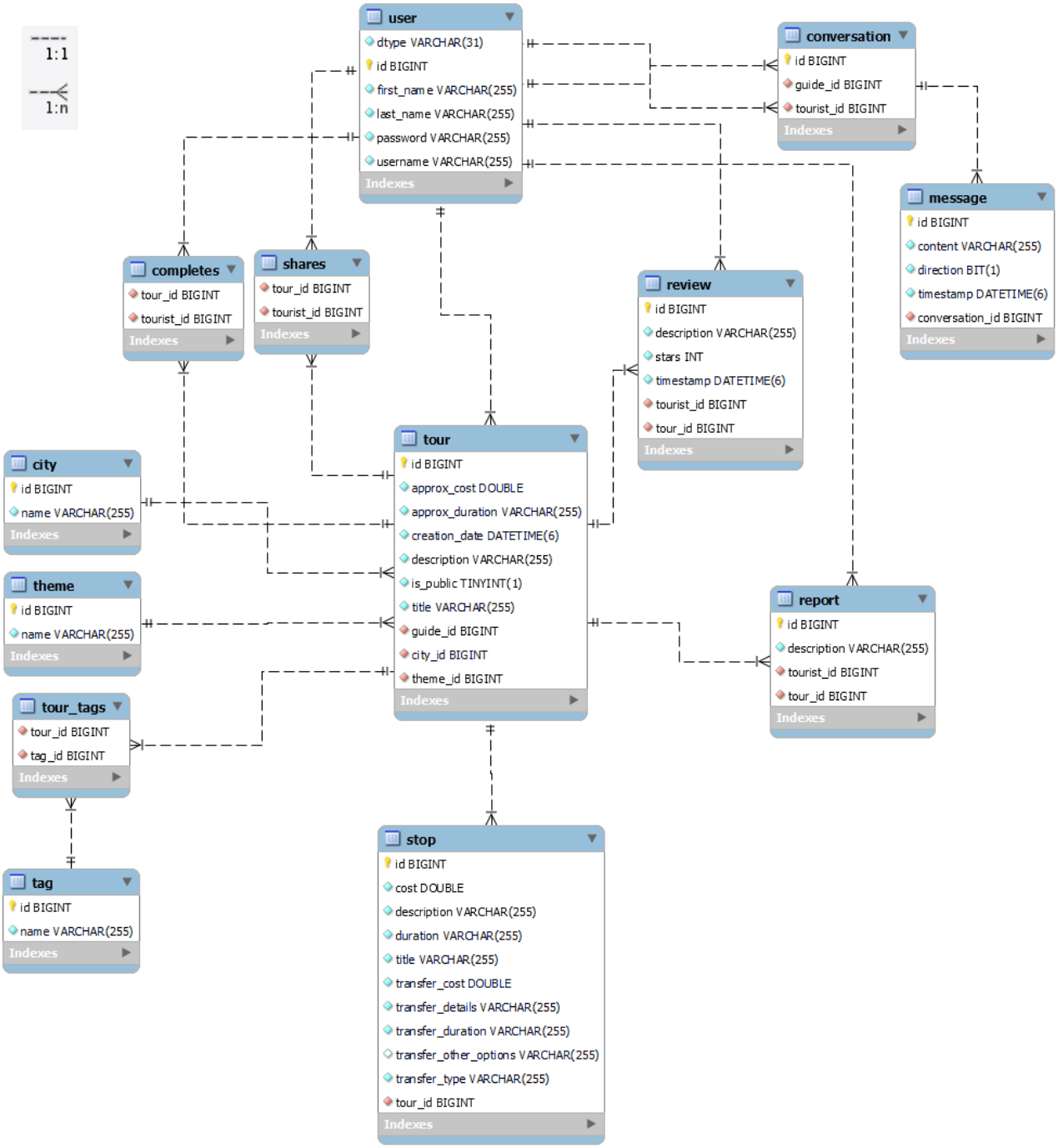
- Spring Boot Data JPA (per l'astrazione dal modello dei dati)
- Spring Boot Web (per il supporto alle API)
- JUnit Jupiter Engine (per JUnit 5)
- Lombok (per riddure la mole di codice boilerplate e ridondante)

Note sulla persistenza

Il principale vantaggio dell'utilizzo di Spring Data JPA è dato dall'astrazione che si ottiene dall'effettivo modello fisico: è possibile scrivere query in linguaggio JPQL, che vengono poi interpretate e mappate in istruzioni SQL e inviate al DBMS da Hibernate.

La libreria Hibernate si occupa di eseguire automaticamente le query DDL e DML per mantenere sempre aggiornate le entità con il sottostante database. Ad esempio, inserendo come proprietà di un'entità un'istanza di un'altra, Hibernate si occuperà di mappare ciò ad una colonna di collegamento uno-a-uno (o molti-a-molti, in base alla cardinalità che si trova dall'altra parte) tra le due tabelle e di garantire il vincolo di integrità. Nel momento in cui tale proprietà verrà valorizzata, anche la corrispondente query di UPDATE verrà eseguita in tempo reale.

Il diagramma entità-relazione risultante da tale procedura eseguita su tutte le entità e in linea con il class diagram progettato, è il seguente:



Si noti che, nel passaggio al modello logico, ora troviamo gli identificatori, i tipi concreti delle colonne di MySQL e le relazioni molti-a-molti trasformate in uno-a-molti con tabella di collegamento. Inoltre, la relazione di ereditarietà tra Utente e Turista, Guida o Amministratore è stata fatta confluire nella superclasse con l'aggiunta di una colonna che discrimina il tipo concreto. Questa scelta è opposta rispetto a quella di duplicare le colonne della superclasse in ciascuna tabella figlia, motivata dal fatto che queste condividono esattamente le medesime proprietà.

Suddivisione dei package

Ho strutturato la base di codice in sei livelli, dove ciascuno si basa sul sottostante per fornire funzionalità di più alto livello al soprastante:

1. Controller
2. DTO

→ View

3. Mapper
4. Service

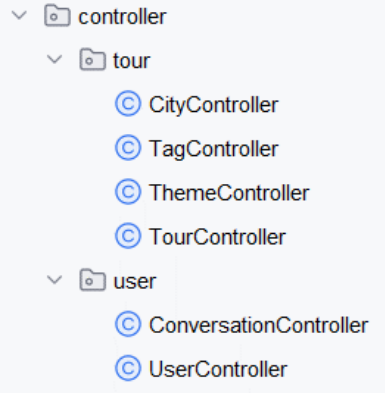
→ Controller

5. Entity
6. Repository

→ Model

Inoltre, all'interno di ciascun livello, ho creato un'ulteriore suddivise in base alla macro-funzionalità (tour o utente) di appartenenza.

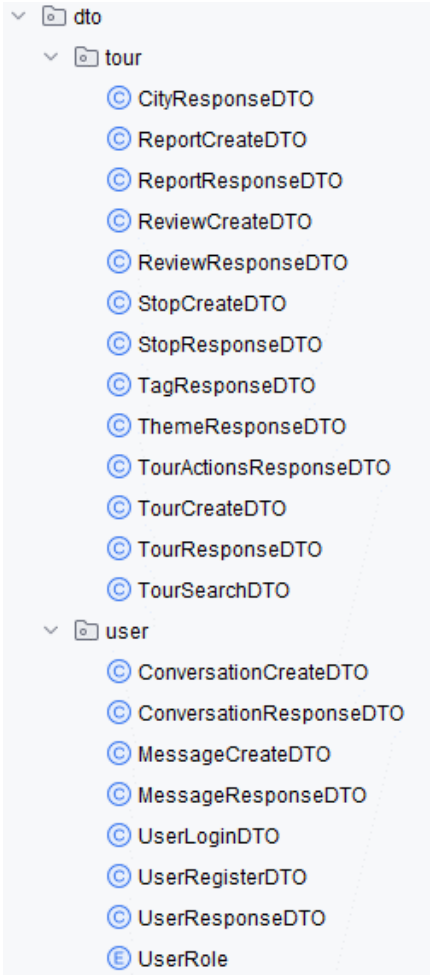
Controller



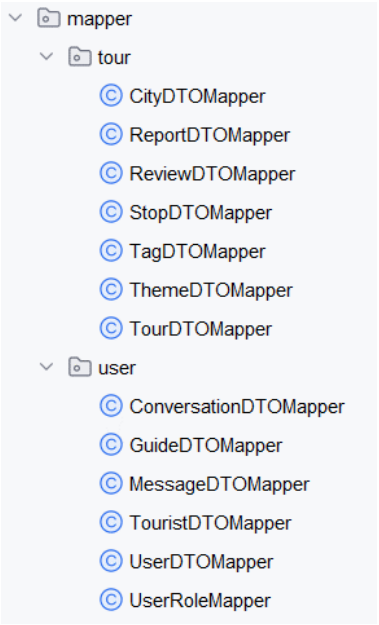
È il livello più vicino al front-end. Gestisce gli aspetti legati all'HTTP e si assicura che le richieste abbiano in entrata e uscita il formato corretto. La sua logica è limitata in modo da delegarla ai livelli sottostanti.

DTO

Contiene le classi che rappresentano il modello ricevuto ed inviato al front-end. Rende possibile l'indipendenza tecnologica tra i due lati dell'applicazione.

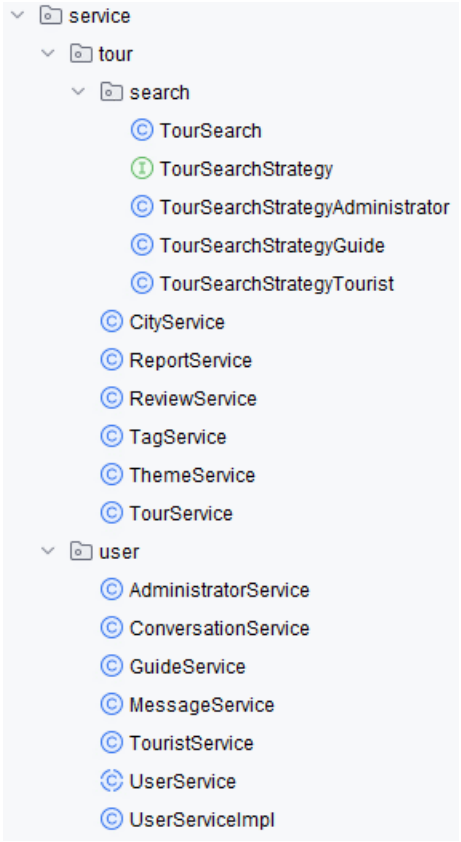


Mapper



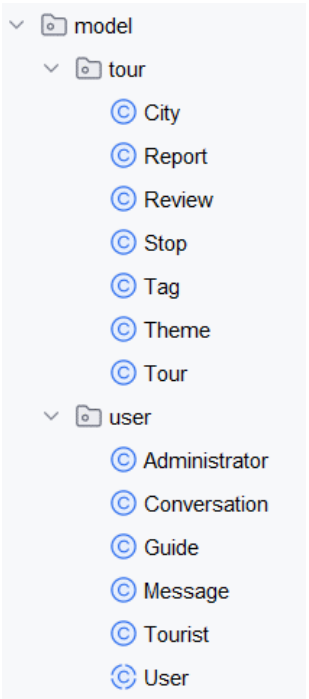
Consente di "tradurre" il modello dei dati conosciuto dalla View (DTO) con quello gestito dal Controller (Entity) e viceversa. Nella pratica si occupa di mappare tra loro due classi differenti ma compatibili.

Service



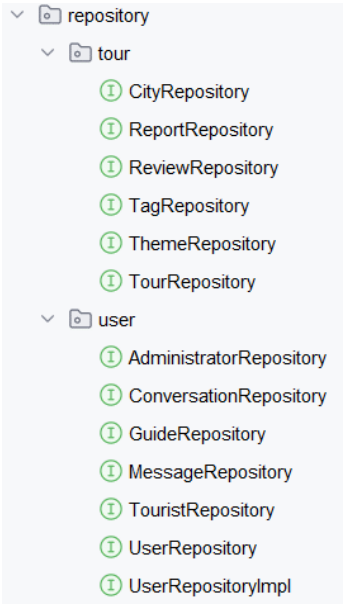
Si tratta di classi che concretizzano la business logic, ovvero quella del dominio applicativo.

Entity



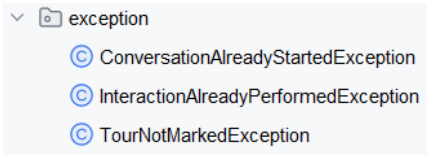
Contiene le classi che rappresentano il modello ricevuto ed inviato alle repository.

Repository



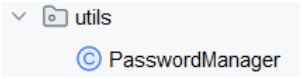
Consente di interfacciarsi con il database sottostante per salvare, recuperare, modificare ed eliminare le entità.

Exception



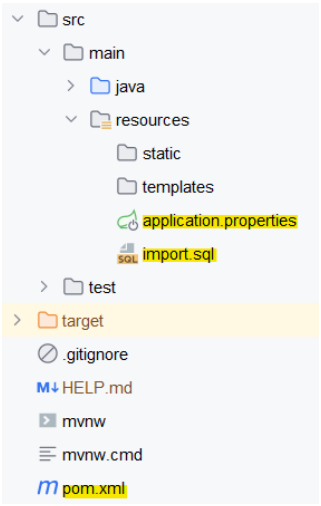
Contiene sottoclassi di RuntimeException che rappresentano situazioni eccezionali non esprimibili adeguatamente dalle eccezioni integrate in Java.

Utility



Fornisce metodi statici per operazioni comuni in modo da limitare la duplicazione di codice.

Altri file di configurazione



Al di fuori della cartella con il codice Java (e quindi dei package appena presentati) ho modificato altri file di configurazione necessari all'applicazione. Ho evidenziato quelli più rilevanti, che verranno esposti più avanti.

Illustrazione della classe principale per ciascun package

Per ciascuno dei package presentati nella sezione precedente, propongo una spiegazione dettagliata della classe più completa e, al tempo stesso, rilevante in termini di funzionalità. In caso di ripetizione di codice, per brevità, questo verrà descritto solo nella prima occorrenza.

Per ulteriori dettagli si può consultare il JavaDoc presente nel codice e/o la relativa documentazione HTML (aprendo il file `./javadoc/index.html`).

Controller (TourController)

Questo controller si occupa di gestire le funzionalità API relative a un tour o ad esso correlate.

Annotaitons

```
Controller for tour endpoints
2 usages  Francesco Balducci
@RequiredArgsConstructor
@RestController
@RequestMapping("/tour")
public class TourController {
```

`@RequiredArgsConstructor`

Annotation di Lombok che consente di generare i costruttori con le proprietà obbligatorie (`private final`) di questa classe. Questi verranno richiamati dal contesto di Spring Boot al momento dell'inizializzazione. Si troverà

questa e altre annotation simili in diverse parti dell'applicazione in quanto riduce il codice ridondante.

`@RestController`

Annotation di Spring Boot che abilita la classe alla gestione delle richieste HTTP tramite API.

`@RequestMapping("/tour")`

Imposta l'entry point per i metodi da mappare nella classe. Per poter essere selezionata dal web server per la gestione di una richiesta, quest'ultima dovrà contenere nella parte iniziare dell'URL il frammento `/tour`; quindi, si procederà a eseguire il matching con uno dei metodi disponibili.

Dipendenze

```
// dependencies autowired by spring boot

17 usages
private final TourService tourService;

10 usages
private final TourDTOMapper tourMapper;

1 usage
private final TagService tagService;

1 usage
private final ReviewService reviewService;

2 usages
private final ReviewDTOMapper reviewMapper;

1 usage
private final ReportService reportService;

3 usages
private final ReportDTOMapper reportMapper;

4 usages
private final UserService<User> userService;

5 usages
private final UserService<Tourist> touristService;

4 usages
private final UserService<Guide> guideService;

2 usages
private final UserService<Administrator> administratorService;

1 usage
private final UserDTOMapper userMapper;
```

Trattandosi del controller che gestisce la maggior parte delle funzionalità, è anche quello che dipende dalla maggior parte dei servizi e mapper. Attraverso queste dipendenze, che vengono iniettate da Spring Boot, le classi di questo layer comunicano con quelli sottostanti.

Esempio di GET mapping

```
Get a specific tour by its ID
Params: tourId – wanted tour
        userId – requesting user
Returns: wanted tour DTO
Throws: ResponseStatusException – HttpStatus.UNAUTHORIZED if user is not authorized to see the
        wanted tour
no usages  Francesco Balducci
@GetMapping("/{tourId}")
public ResponseEntity<TourResponseDTO> getById(@PathVariable Long tourId, @RequestHeader(value="Authentication") Long userId) {
    Tour t = tourService.findById(tourId);
    userService.findById(userId);
    if(!tourService.isVisible(t, userId)) { // ensure user (tourist, guide or administrator) can actually see the wanted tour
        throw new ResponseStatusException(HttpStatus.UNAUTHORIZED, "Non sei autorizzato a visualizzare questo tour");
    }
    return ResponseEntity.ok(tourMapper.convertResponse(t));
}
```

Quando arriva una richiesta ad un indirizzo quale `/tour/0`, questo metodo viene attivato come indicato nell'annotation.

I suoi parametri verranno valorizzati con i dati della richiesta. In particolare, `tourId` varrà 0 e `userId` varrà 1 (ipotizzando l'utente si sia precedentemente loggato con le credenziali del turista Mario Rossi, che ha ID 1).

In caso non si sia loggato, il front-end riceverà immediatamente una risposta di errore per la mancanza del parametro di autenticazione, ottenendo insomma che il metodo sia autenticato. In ogni caso, anche l'altro lato esegue dei controlli complementari per evitare a monte questa situazione.

Il metodo procederà a invocare il servizio relativo ai tour per recuperare quello richiesto tramite ID. Nel caso in cui non sia trovato, il servizio lancerà un'eccezione `NoSuchElementException` che, similarmente nel caso precedente, ritornerà al front-end una risposta di errore. Successivamente viene interrogato il servizio che gestisce gli utenti, nella sua variante non specifica per ruolo dato che il metodo è utilizzabile da più ruoli. Viene verificata l'esistenza dell'utente richiedente, per limitare manomissioni della richiesta dove può essere digitato un ID casuale. In caso non sia avvenuto nessuno degli errori descritti, siamo sostanzialmente sicuri che sia l'utente che il tour esistono. Si procede quindi a verificare che l'utente possa effettivamente "vedere" il tour richiesto (sempre per la situazione di modifiche manuali alle richieste). La logica di business del servizio definisce la visibilità di un tour

come segue: se esso è pubblico, tutti possono vederlo; altrimenti, se l'utente è un turista, deve essere tra quelli con il quale è stato condiviso, se si tratta di una guida deve esserne l'autore, se si tratta di un amministratore non ci sono limitazioni.

Infine, in caso non siano capitati errori, viene richiamato il mapper dei tour per modificare la struttura dell'entity ritornata dal servizio, in modo che si adegui al DTO atteso dal front-end.

DTO (TourResponseDTO)

Per l'invio di un oggetto tour al front-end, ho utilizzato una classe DTO (con sole proprietà) per modellizzare il tipo che esso si aspetta in base al contratto.

Annotations

```
Implementation of DTO pattern for returning tours in the outgoing body
See Also: com.what2see.model.tour.Tour
14 usages  Francesco Balducci
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class TourResponseDTO {
```

`@Data`

Annotation di Lombok che racchiude più funzionalità, tra cui la generazione dei getter e setter, del metodo toString(), equals(), hashCode(), ecc.

`@Builder`

Annotation di Lombok che predispone un builder per costruire in maniera compatta le istanze di questa classe, senza dover richiamare un setter per volta.

`@AllArgsConstructor, @NoArgsConstructor`

Annotations di Lombok che consentono di generare i costruttori con tutte le proprietà e quello vuoto. Richiesto dalla precedente annotation.

Esempio di proprietà

```
no usages
private @NotNull Long id;

no usages
private @NotNull UserResponseDTO author;     // guide

no usages
private @NotBlank String title;

no usages
private @NotBlank String description;

no usages
private @NotNull Boolean isPublic;
```

Ho riportato in questa classe le proprietà dell'entità Tour che ho ritenuto importanti da far conoscere al front-end per le funzionalità da implementarvi (in questo elenco non sono indicate tutte). In caso di sotto-entità, sono state a loro volta sostituite con il loro DTO. Le annotazioni come `@NotNull` aggiungono dei vincoli di validità per i valori assegnabili alla proprietà.

Mapper (TourDTOMapper)

Per trasformare le entità Tour nel DTO appena mostrato, ho creato una classe che si occupa di mappare i due tipi tra loro.

Annotation

```
Service that converts Tour entities from and to DTOs.
Is usually used in controller to communicate with client side.
2 usages   Francesco Balducci
@RequiredArgsConstructor
@Service
public class TourDTOMapper {
```

`@Service`

Annotation di Spring Boot che marca la classe tra quelle gestite dal container, in modo che venga istanziata una sola volta e iniettata in tutte le classi che dipendono da essa.

Dipendenze

Dovendo gestire anche i DTO ricevuti dal front-end e dato che questi, nel caso di sotto-entità già esistenti (es. la città dentro un tour), indicano soltanto il loro ID anziché l'oggetto completo, la classe dipende dai relativi servizi in modo da poterle recuperare per intero.

```
// dependencies autowired by spring boot

1 usage
private final UserService<Tourist> touristService;

1 usage
private final UserService<Guide> guideService;

1 usage
private final UserDTOMapper userMapper;

1 usage
private final CityService cityService;

1 usage
private final CityDTOMapper cityMapper;

1 usage
private final TagService tagService;

1 usage
private final TagDTOMapper tagMapper;

1 usage
private final ThemeDTOMapper themeMapper;

1 usage
private final ThemeService themeService;

2 usages
private final StopDTOMapper stopMapper;

1 usage
private final ReviewDTOMapper reviewMapper;
```

Esempio di metodo di mapping

```
Converts a Tour entity to a DTO that can be sent to client
Params: tour – entity to be converted
Returns: DTO that can be sent to client
Francesco Balducci
public TourResponseDTO convertResponse(Tour tour) {
    return TourResponseDTO.builder()
            .id(tour.getId())
            .author(userMapper.convertResponse(tour.getAuthor()))
            .title(tour.getTitle())
            .description(tour.getDescription())
            .isPublic(tour.isPublic())
            .city(cityMapper.convertResponse(tour.getCity()))
            .tags(tour.getTags().stream().map(tagMapper::convertResponse).collect(Collectors.toList()))
            .theme(themeMapper.convertResponse(tour.getTheme()))
            .approxCost(tour.getApproxCost())
            .approxDuration(tour.getApproxDuration())
            .creationDate(tour.getCreationDate())
            .stops(tour.getStops().stream().map(stopMapper::convertResponse).collect(Collectors.toList()))
            .reviews(tour.getReviews().stream().map(reviewMapper::convertResponse).collect(Collectors.toList()))
            .markedAsCompletedCount((long) tour.getMarkedTourists().size())
            .build();
}
```

Il metodo si occupa di trasformare il parametro entità Tour nel DTO di risposta da inviare al front-end. Sfruttando il builder di Lombok, costruisce l'istanza copiandovi i parametri necessari.

Service (TourService)

Questa classe racchiude tutta la principale logica di business legata ai tour.

Annotation

```
Service class that handles the business logic for Tour entities.
29 usages  Francesco Balducci
@Service
@Transactional
@RequiredArgsConstructor
public class TourService {
```

```
@Transactional
```

Annotation di Spring Boot che abilita l'utilizzo delle transazioni. In caso di un metodo che modifica più entità, fa in modo che ciò avvenga nel contesto di una transazione e di conseguenza che, in caso di problemi, si annulli in blocco l'operazione senza rischi quale uno stato di inconsistenza del database.

Dipendenze

```
// dependencies autowired by spring boot

5 usages
private final TourRepository tourRepository;

1 usage
private final CityService cityService;

1 usage
private final TagService tagService;

1 usage
private final ThemeService themeService;

2 usages
private final UserService<Administrator> administratorService;

1 usage
private final UserService<User> userService;
```

Questa classe è l'unica autorizzata ad avere accesso alla repository di un Tour, in modo da incapsulare le opzioni di accesso al database. Dipende anche da altri servizi poiché, ad esempio, potrebbe aver bisogno di recuperare le entità complete a partire dal loro ID.

Esempio di metodo

```
Marks the given Tour entity as completed by the given Tourist entity.
Params: t – tour to be marked as completed
        tt – requesting tourist
Throws: InteractionAlreadyPerformedException – if the tourist has already marked the tour as
        completed
5 usages  Francesco Balducci
public void markAsCompleted(Tour t, Tourist tt) throws InteractionAlreadyPerformedException {
    // check if the tourist has already marked the tour as completed
    List<Tourist> completes = t.getMarkedTourists();
    if(completes.contains(tt))
        throw new InteractionAlreadyPerformedException(t, tt);

    completes.add(tt);
}
```

Questo metodo, a partire da un tour e un turista, si occupa di marcare il primo come completato dal secondo.

La logica di business in questo caso consiste nell'impedire che tale azione venga fatta, dal turista, più volte sullo stesso tour. Pertanto, verifica se nell'elenco dei completamenti del tour ci sia l'istanza del turista, in qual caso lancia l'eccezione `InteractionAlreadyPerformedExceptinon`. In caso contrario, aggiunge il turista all'elenco. Hibernate rileverà la modifica all'entità e si occuperà di aggiornare il corrispondente record nel database.

Entity (Tour)

Questa classe rappresenta l'entità Tour nel modello, così com'è vista dalla repository e dai service.

Annotation

```
Entity that represents a tour in the database.
Francesco Balducci
@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Tour {
```

`@Entity`

Annotation di Spring Boot che marca la classe come entità persistente. Per questo motivo, Hibernate esegue le query DDL in modo che sia creata una tabella compatibile con istanze di questa classe, incluse le relazioni con altre entità.

Esempio di proprietà e relative annotations

```
no usages
@Column(nullable = false, columnDefinition = "tinyint(1) default 1")
private boolean isPublic;

no usages
@ManyToOne(cascade = { CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH })
@JoinColumn(name = "guideId", nullable = false)
private Guide author;

2 usages
@OneToMany(fetch = FetchType.EAGER, mappedBy = "tour", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Stop> stops;

no usages
@OneToMany(mappedBy = "tour", cascade = CascadeType.ALL)
private List<Report> reports;

no usages
@OneToMany(mappedBy = "tour", cascade = CascadeType.ALL)
private List<Review> reviews;

no usages
@ManyToOne(cascade = { CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH })
@JoinColumn(name = "cityId", nullable = false)
private City city;

no usages
@ManyToMany(cascade = { CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH })
@JoinTable(
        name = "tour_tags",
        joinColumns = {@JoinColumn(name = "tourId")},
        inverseJoinColumns = {@JoinColumn(name = "tagId")}
)
private List<Tag> tags;
```

Ad ogni proprietà corrisponde una colonna nel database. Inoltre, vi sono diversi tipi di annotation per personalizzare alcune caratteristiche relazionali:

```
@Column(nullable = false, columnDefinition = "tinyint(1) default 1")
```

Specifica che la colonna corrispondente deve essere Not Null e impone tale vincolo anche ai valori assegnabili all'oggetto. Inoltre, ho forzato la conversione del tipo di dato booleano in una colonna a singolo bit che di default è impostato a 1.

```
@OneToMany(fetch = FetchType.EAGER, mappedBy = "tour", cascade =
CascadeType.ALL, orphanRemoval = true)
private List<Stop> stops;
```

Specifica che la lista di entità va interpretata come collegamento uno-a-molti, dove questo è il lato uno. Esattamente come in una normale tabella relazionale, implica che questo non sia il lato che "amministra" la relazione (c'è il parametro `mappedBy`): deve essere l'altro lato a indicare una eventuale istanza di questa classe come proprietà per poter salvare effettivamente la relazione.

Il parametro `fetch` impostato a `EAGER` indica che le entità in relazione vadano recuperate a prescindere che siano effettivamente richieste dal codice. Questo perché di default Hibernate ottimizza il loro recupero al solo istante in cui sono utilizzate. Tuttavia, dato lo stretto legame concettuale e l'utilizzo quasi sempre contestuale del tour e delle sue fermate, ho ritenuto importante che siano recuperati in contemporanea di default.

```
@ManyToOne(cascade = { CascadeType.MERGE, CascadeType.PERSIST,
CascadeType.REFRESH })
@JoinColumn(name = "cityId", nullable = false)
private City city;
```

Specifica che l'entità va intesa come collegamento uno-a-molti, dove questo è il lato molti. Potendo amministrare la relazione, ho potuto specificare i vincoli di integrità referenziale, per i quali, in caso di modifiche al tour, queste vengano propagate anche alle fermate collegate. Ho anche personalizzato il nome della colonna con la chiave esterna e l'ho resa obbligatoria.

```
@ManyToMany(cascade = { CascadeType.MERGE, CascadeType.PERSIST,
CascadeType.REFRESH })
@JoinTable(
        name = "tour_tags",
        joinColumns = {@JoinColumn(name = "tourId")},
        inverseJoinColumns = {@JoinColumn(name = "tagId")}
)
private List<Tag> tags;
```

Specifica che la lista di entità va intesa come un collegamento molti-a-molti, dove questo è il lato che governa la relazione (dall'altro lato c'è il parametro `mappedBy`). Ho anche specificato il nome della tabella di collegamento e delle sue due colonne.

Repository (TourRepository)

Questa classe svolge la funzione di astrazione dal database e di interfacciamento con esso.

Annotation

```
Repository that provides CRUD operations for Tour entities.
17 usages  Francesco Balducci
@Repository
public interface TourRepository extends JpaRepository<Tour, Long> {
```

`@Repository`

Annotation di Spring Boot che abilita le operazioni CRUD sul database. Unito all' `extends JpaRepository<Tour, Long>` fornisce dei metodi di base (es. recupero di un'entità dal suo ID) e la possibilità di dichiararne di nuovi attraverso il linguaggio di query JPQL.

Non si tratta di una classe concreta: l'implementazione dei metodi dipende dalle specifiche del database sottostante e viene pertanto generata all'avvio del container.

Esempio di query JPQL

```
Searches for tours without considering their visibility status.
It is intended to only be used by administrators.
Params: city – city filter (optional)
        theme – theme filter (optional)
        approxDuration – maximum duration filter (optional)
Returns: list of tours that match the given criteria
```

```
1 usage  Francesco Balducci
@Query("SELECT t FROM Tour t LEFT JOIN t.tags tt WHERE " +
       "(:city IS NULL OR t.city = :city) AND " +
       "(:theme IS NULL OR t.theme = :theme) AND " +
       "(:approxDuration IS NULL OR t.approxDuration ⩽ :approxDuration)")
List<Tour> searchAll(City city, Theme theme, String approxDuration);
```

Questo metodo dichiara una query di ricerca per uno o più tour in base a dei filtri per città, tema e durata massima (applicati solo se non nulli). Come anticipato, Hibernate si occupa di tradurre ed eseguire la corrispondente query in linguaggio SQL, oltre che di istanziare le entità con i valori ricevuti e ritornarle al chiamante.

Exception (InteractionAlreadyPerformedExceptinon)

Questa eccezione personalizzata viene sollevata quando si cerca di eseguire un'azione che nel dominio applicativo è prevista come eseguibile solo una volta.

Esempio di costruttori

```
Exception thrown when a user tries to interact with a tour when that interaction has already been performed and isn't
expected to be repeatable. (e.g. tourist tries to mark / review / report a tour multiple times)
See Also: com.what2see.service.tour.ReviewService.create
```

```
25 usages  Francesco Balducci
public class InteractionAlreadyPerformedException extends RuntimeException {

    3 usages
    private Tour tour;

    3 usages
    private Tourist tourist;

    3 usages  Francesco Balducci
    public InteractionAlreadyPerformedException(Tour tour, Tourist tourist) {
        this.tour = tour;
        this.tourist = tourist;
    }

    no usages  Francesco Balducci
    public InteractionAlreadyPerformedException(String message) { super(message); }
```

La classe estende `RuntimeException` e tutti i costruttori di default. Dato che, allo stato attuale, viene sollevata esclusivamente come causa di un turista che vuole eseguire un'interazione su un tour, è presente un costruttore personalizzato in cui si può specificare i due oggetti coinvolti.

Utility (PasswordManager)

Questa classe con soli metodi statici consente di svolgere funzioni comuni sulle password dal punto di vista crittografico.

Metodo di hashing

```
Utility class that provides methods for hashing passwords.
4 usages  Francesco Balducci *
public class PasswordManager {

    Hashes a password using the SHA-256 algorithm.
    Params: password – password to be hashed
    Returns: hashed password
    3 usages  Francesco Balducci
    public static String hashPassword(String password) {
        // get an instance of the SHA-256 algorithm
        MessageDigest md;
        try {
            md = MessageDigest.getInstance( algorithm: "SHA-256");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
        // hash the password
        md.update(password.getBytes());
        byte[] digest = md.digest();
        // convert the hash to string
        return String.format("%064x", new java.math.BigInteger( signum: 1, digest));
    }
}
```

Il metodo `hashPassword` si occupa di prendere in ingresso una stringa contente una password in chiaro e di restituirla crittografata in hash SHA-256. Ho ritenuto indispensabile, per motivi di sicurezza, eseguire questa trasformazione prima di memorizzare un nuovo utente nel database.

Altri file di configurazione (application.properties, import.sql, pom.xml)

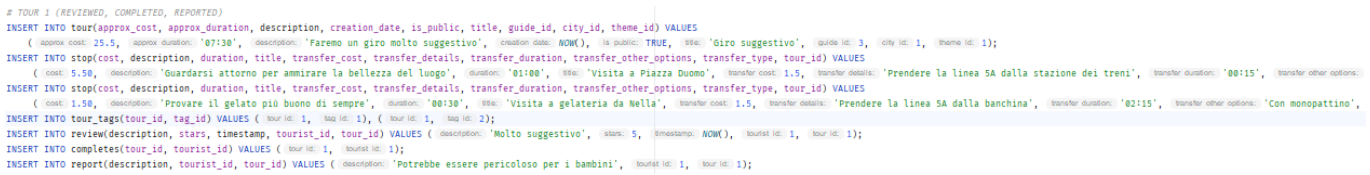
I file indicati non rientrano nel codice applicativo Java, ma sono indispensabili per configurare correttamente l'applicazione.

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/WhatToSee?useSSL=false&use
spring.datasource.username=root
spring.datasource.password=root
spring.sql.init.mode=always
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.properties.hibernate.format_sql=true
spring.mvc.view.prefix = /WEB-INF/views/
spring.mvc.view.suffix = .jsp
```

Questo file contiene una coppia chiave-valore per ogni riga con le configurazioni delle librerie utilizzate. Ad esempio, con `spring.jpa.properties.hibernate.dialect` specifico il linguaggio utilizzato dal DBMS e con `spring.datasource.url` l'indirizzo e la porta a cui è raggiungibile.

import.sql



```
# TOUR 1 (REVIEWED, COMPLETED, REPORTED)
INSERT INTO tour(approx_cost, approx_duration, description, creation_date, is_public, title, guide_id, city_id, theme_id) VALUES
    ( approx_cost: 25.5,  approx_duration: '07:30',  description: 'Faremo un giro molto suggestivo',  creation_date: NOW(),  is_public: TRUE,  title: 'Giro suggestivo',  guide_id: 3,  city_id: 1,  theme_id: 1);
INSERT INTO stop(cost, description, duration, title, transfer_cost, transfer_details, transfer_duration, transfer_other_options, transfer_type, tour_id) VALUES
    ( cost: 5.50,  description: 'Guardarsi attorno per ammirare la bellezza del luogo',  duration: '01:00',  title: 'Visita a Piazza Duomo',  transfer_cost: 1.5,  transfer_details: 'Prendere la linea 5A dalla stazione dei treni',  transfer_duration: '00:15',  transfer_other_options:
INSERT INTO stop(cost, description, duration, title, transfer_cost, transfer_details, transfer_duration, transfer_other_options, transfer_type, tour_id) VALUES
    ( cost: 1.50,  description: 'Provare il gelato più buono di sempre',  duration: '00:30',  title: 'Visita a gelateria da Nella',  transfer_cost: 1.5,  transfer_details: 'Prendere la linea 5A dalla banchina',  transfer_duration: '02:15',  transfer_other_options: 'Con monopattino',
INSERT INTO tour_tags(tour_id, tag_id) VALUES ( tour_id: 1,  tag_id: 1), ( tour_id: 1,  tag_id: 2);
INSERT INTO review(description, stars, timestamp, tourist_id, tour_id) VALUES ( description: 'Molto suggestivo',  stars: 5,  timestamp: NOW(),  tourist_id: 1,  tour_id: 1);
INSERT INTO completes(tour_id, tourist_id) VALUES ( tour_id: 1,  tourist_id: 1);
INSERT INTO report(description, tourist_id, tour_id) VALUES ( description: 'Potrebbe essere pericoloso per i bambini',  tourist_id: 1,  tour_id: 1);
```

Questo file SQL contiene le query DML personalizzate da eseguire all'avvio del container. Esse mi hanno consentito di avere, durante lo sviluppo e soprattutto nel testing, un database sempre popolato con dati realistici, eterogenei e prevedibili. L'elenco mostrato non è esaustivo.

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.what2see</groupId>
    <artifactId>WhatToSee-Apis</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>WhatToSee-Apis</name>
    <description>WhatToSee-Apis</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
```

Questo file XML contiene la dichiarazione del manifest dell'applicazione Maven e tutte le dipendenze da recuperare prima dell'avvio.

Design pattern

I design pattern sono delle strutture note di soluzioni a problemi ricorrenti nei processi di progettazione e sviluppo orientato agli oggetti. Puntano ad ottenere una base di codice flessibile ed estendibile, limitando in caso di espansioni le modifiche da apportare al codice già esiste.

Si suddividono in quattro categorie:

- Creazionali: meccanismi per istanziare oggetti in modo da rafforzare la flessibilità e il riutilizzo del codice;
- Strutturali: meccanismi di integrazione tra oggetti mantenendo una struttura flessibile;
- Comportamentali: modelli di comportamento per oggetti e meccanismi di scambio di informazione tra essi;

- Architetturali: modalità di organizzazione della logica applicativa per favorire l'evoluzione e l'indipendenza tecnologica;

Di seguito spiegherò i dettagli dei pattern che ho deciso di utilizzare e/o implementare. I primi tre sono parte del framework scelto.

Inversion of Control (architetturale)

Questa funzionalità è messa in atto da container di Spring Boot, che fornisce un'architettura di inversione del controllo: la responsabilità del reperimento delle dipendenze di una classe è delegata a un'entità esterna, che la amministra.

Dependency injection (comportamentale)

Come già presentato in più classi, ho sfruttato un meccanismo di iniezione delle dipendenze per evitare l'instaurazione di legami forti tra le due parti. Ciò consente, ad esempio, di modificare la dinamica in sede di test iniettando una versione "simulata" (o mock) di una dipendenza senza modificare né causare errori nella classe oggetto di test.

Singleton (creazionale)

Nel contesto di Spring Boot, un servizio è anche definito "bean". Solitamente, oltre ad essere iniettati ove richiesti, essi vengono istanziati una sola volta all'avvio dell'applicazione. Ciò trova origine nell'idea che i servizi non dovrebbero conservare uno stato, dovendo esporre funzioni pure. Ad esempio, la modifica accidentale a proprietà di stato potrebbe provocare effetti collaterali difficilmente rintracciabili e in molti punti dell'applicazione.

Per questa ragione, oltre che per performance, ho sfruttato il pattern Singleton per fare in modo che, in tutta l'applicazione, vi sia sempre e solo un'istanza di ciascun servizio.

MVC (architetturale)

Nella suddivisione dei package ho presentato la separazione dei sorgenti in modello (M), logica (C) e interfaccia (V). La motivazione dietro questa strutturazione è rappresentata dal vantaggio di isolare, grazie a questo pattern, le responsabilità e i punti di vista delle diverse aree applicative così da renderle tecnologicamente indipendenti e libere di evolversi in quasi completa autonomia.

Ad esempio, una classe dell'interfaccia come il TourController può unicamente comunicare, attraverso DTO e mapper, con il livello della logica, dove troviamo il TourService può fare stesso con la repository TourRepository per mezzo delle Entity.

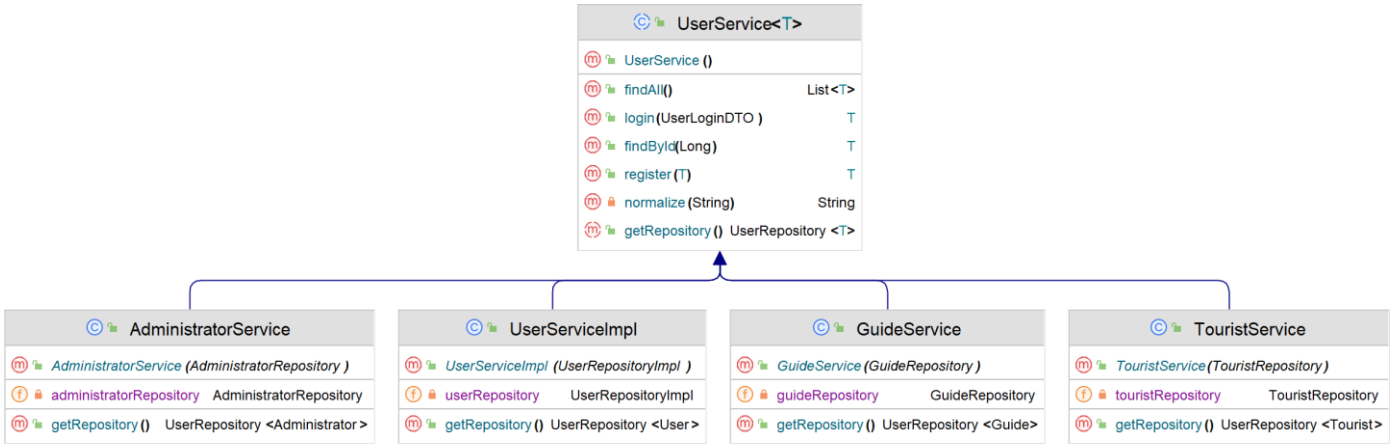
DTO (architetturale)

Nella sezione dedicata al package DTO ho elencato le classi con cui ho implementato questo pattern. Consiste nello svincolare il modello dei dati scambiato tra front-end e back-end da quello che ciascuno manipola internamente.

Il vantaggio è rappresentato dal fatto che, purché venga mantenuto l'accordo sul tipo di dato scambiato, il back-end si possa evolvere in maniera indipendente dal front-end senza che la modifica di uno obblighi a modificare anche l'altro. Inoltre, oltre a limitare la mole di dati scambiati, si circoscrivono al massimo le responsabilità di ciascuno, senza che siano esposte più informazioni di quelle strettamente necessarie all'assolvimento delle rispettive funzioni.

Factory method (creazionale)

Durante la progettazione, ho constatato che avrei necessitato di duplicare i medesimi metodi nei servizi legati alle differenti sottoclassi di `User`, con l'unica differenza dell'entità concreta gestita e della repository utilizzata. Di conseguenza, ho strutturato le classi nel seguente modo, per conformarmi al design pattern factory method:

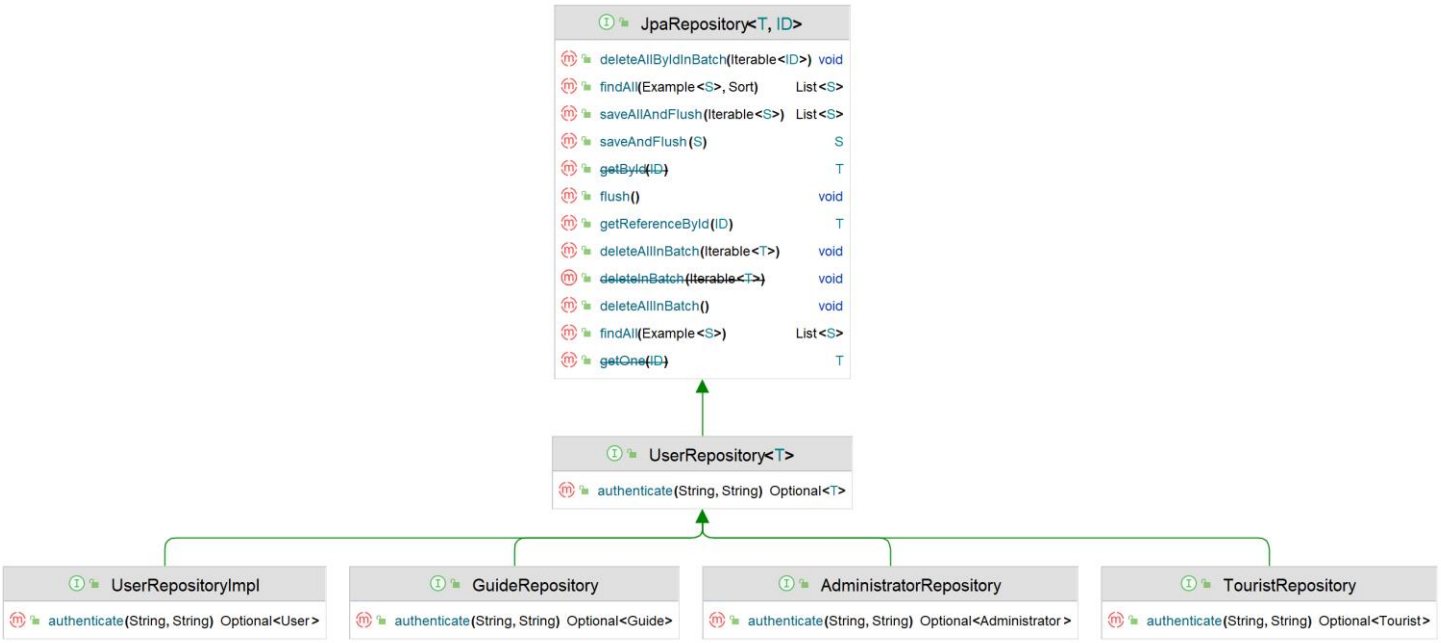


Ho creato in `UserService` un metodo astratto `getRepository()` che ritorna una generica repository parametrizzata per una sottoclassi di `User`. Ho quindi implementato i metodi comuni operando sul suddetto metodo. Successivamente ho creato una sottoclasse concreta di `UserService` per ciascuna sottoclasse di `User` e implementato il metodo `getRepository()` ritornando la repository specifica.

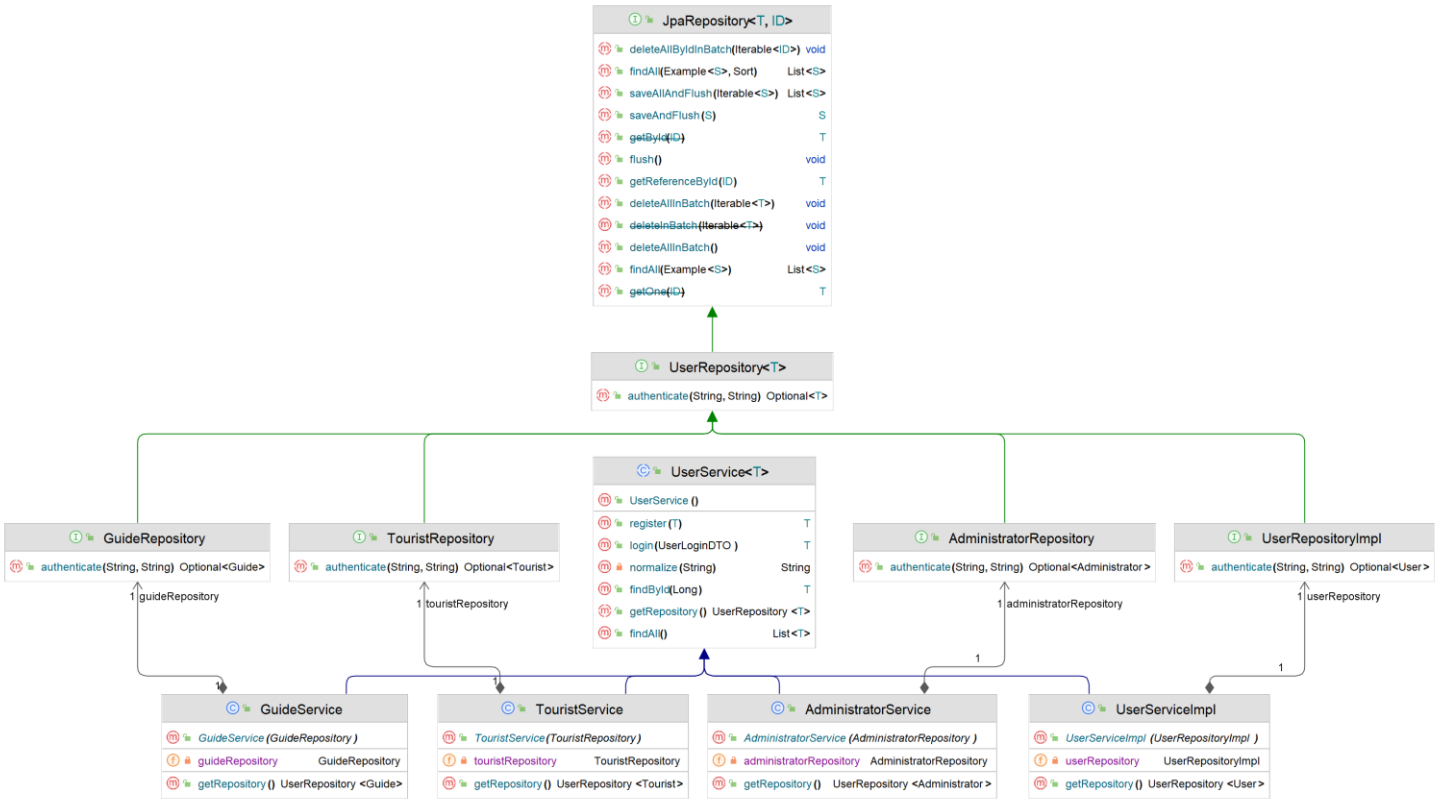
In letteratura abitualmente si individua la figura del "consumer", ovvero una classe che utilizza la struttura appena presentata dichiarando una variabile di tipo generico `UserService` e assegnandovi una sua sottoclasse concreta in base a una condizione. In questo caso, invece, il passaggio non è così evidente poiché, volendo sfruttare la dependency injection, è sufficiente per i servizi dichiarare una proprietà come, ad esempio, `private final UserService<Tourist> touristService;` per ricevere correttamente un'istanza di `TouristService`.

L'utilizzo del parametro `<T extends User>` consente di evitare di creare per errore servizi che gestiscano entità non corrette e, al tempo stesso, di godere della tipizzazione specifica per il sottotipo di `User` desiderato.

Ho approfittato di questa strutturazione del codice per generalizzare anche i tipi delle repository:



Infine, propongo un diagramma complessivo in cui vengono mostrate le relazioni tra i due interventi appena descritti:



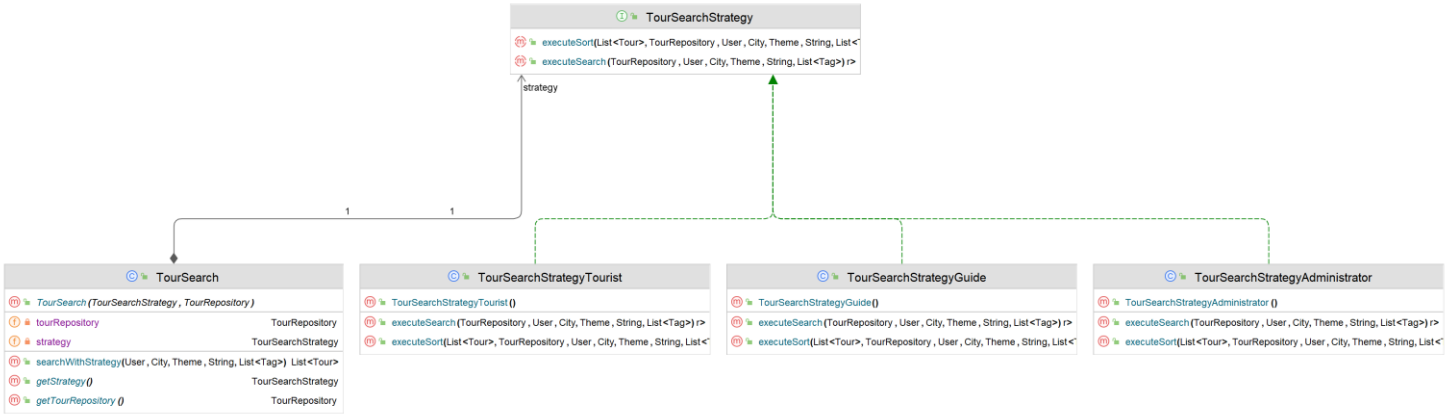
La motivazione per cui è stata creata anche un service e una repository per `User` nasce dalla necessità di eseguire il log in di un utente indipendentemente dal suo ruolo. Anche in questa situazione non ci sono problemi dato che, nonostante l'entità ritornata sarà di tipo `User` ed essendo questo tipo astratto,

siamo sicuri che si tratterà necessariamente di un'istanza di una delle sue sottoclassi, ottenendo anche il beneficio di ridurre drasticamente la duplicazione di codice la cui unica differenziazione consisterebbe nel tipo gestito.

Strategy (comportamentale)

Tra i requisiti funzionali vi è la necessità di ricercare tra i tour secondo più modalità, dipendenti dal tipo di utente. Per questo motivo ho deciso di implementare il design pattern Strategy, che consente di definire una classe che ha il compito di eseguire un'azione secondo una modalità definita a runtime. Ho quindi creato un package chiamato `search` all'interno di `service`, in modo da delimitare esplicitamente l'utilizzo del pattern.

Ecco il diagramma delle classi da cui esso è caratterizzato:



Un'istanza di `TourSearch` viene creata all'interno del metodo di ricerca in `TourService` e le viene passata una strategia (successivamente modificabile) e la repository `TourRepository` (non potrebbe recuperarla autonomamente trattandosi di un'istanziazione "manuale"). In particolare, la strategia è scelta dal servizio attraverso uno `switch` sul tipo di utente richiedente.

Quindi, viene chiamato il metodo `searchWithStrategy` di `TourSearch`. Tale metodo, che riceve tutti i parametri di ricerca incluso l'utente richiedente, non fa nient'altro che richiamare il metodo `executeSearch` e poi `executeSort` sulla strategia in quel momento impostata, senza avere informazione su quale essa concretamente sia. Questo è possibile perché ciascuna strategia estende l'interfaccia `TourSearchStrategy`, che dichiara tali metodi astratti. All'interno di

`TourSearchStrategyGuide`, ad esempio, si provvede a invocare il metodo `searchPublicOrCreatedBy` di `TourRepository` e ad ordinare i risultati mostrando, prioritariamente, per primi quelli creati dalla guida richiedente, poi per quelli più recensiti e infine quelli più segnati come percorsi. Invece, in `TourSearchStrategyAdministrator` si utilizza il metodo `searchAll` e l'ordinamento primario è su quelli più segnalati.

Il vantaggio principale di questo pattern consiste nella possibilità di estendere illimitatamente le strategie (a patto che queste siano racchiudibili in una interfaccia comune), dovendo soltanto creare una nuova sottoclasse di `TourSearchStrategy` e una nuova casistica di scelta della strategia nel servizio.

Testing

Durante lo sviluppo, gli errori di compilazione sono relativamente semplici da rintracciare, dato che vengono evidenziati dall'IDE nel luogo esatto dove sono originati. Tuttavia, con l'evolversi della base di codice (specialmente in un ambiente collaborativo), è facile perdere di vista una configurazione erronea di un metodo che, nonostante non impedisca la compilazione, dia luogo ad un risultato errato nel dominio del problema. Per questo motivo è essenziale prevedere un piano di testing efficace e conforme alle proprie necessità.

In questo caso, ho scelto di concentrarmi sulle classi del package `service`, in quanto considero essenziale rispettare la business logic ed essendo queste le classi dove si concentra la maggior parte del codice extra-liberia. Ho tralasciato la necessità di mockare le repository da cui questi dipendono poiché ho sostanzialmente utilizzato le API previste dalla libreria, che è assunta funzionante. Per rendere prevedibile il contenuto del database su cui confrontare l'esito del test ho utilizzato il file `import.sql` già presentato.

Illustrazione della classe di test principale (TourServiceTest)

Per i medesimi motivi per cui ho scelto, nella sezione di spiegazione dettagliata delle classi, di trattare le classi relative all'entità `Tour`, propongo un'illustrazione generale della classe di test per `TourService`.

EntityMock

Avendo bisogno di confrontare il valore di ritorno del servizio testato a fronte di un valore atteso, ho creato una classe annotata con `@Component` (analogo a `@Service`) chiamata `EntityMock`. Essa contiene un metodo get*Entity* e getAll*Entities*, per ciascuna entity, che la/e recupera dalla repository reale.

Annotations

```
Test class for TourService.
2 usages  Francesco Balducci
@Transactional
@SpringBootTest
@ExtendWith(MockitoExtension.class)
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
class TourServiceTest {
```

`@SpringBootTest`

Annotation che implica l'avvio del container di Spring Boot, necessario per assicurarsi della corretta configurazione del framework e per fornire tutte le dipendenze.

`@ExtendWith(MockitoExtension.class)`

Annotation che richiede, come plugin di testing, la libreria `Mockito` per simulare il comportamento di eventuali dipendenze mockate.

Dipendenze

```
// dependencies autowired by spring boot
46 usages
private final EntityMock mock;
27 usages
private final TourService tourService;
```

Generalmente, le uniche dipendenze richieste sono la classe di utilità per recuperare le entità e il servizio oggetto di test.

Esempio di test positivo

```
Tests TourService.markAsCompleted(Tour, Tourist) in the successful case.
Ensures that the tour is marked as completed by the expected tourist.
no usages  Francesco Balducci
@Test
void markAsCompleted() {
    // setup
    Tourist subject = mock.getTourist();
    // find a tour that is not already marked by the subject (otherwise the test would fail)
    List<Long> expectedMarkedIds = subject.getMarkedTours().stream().map(Tour::getId).toList();
    Tour expectedNotMarked = mock.getAllTours().stream().filter(t → !expectedMarkedIds.contains(t.getId())).findAny().orElseThrow();
    // under test
    tourService.markAsCompleted(expectedNotMarked, subject);
    // assertion
    assertTrue(expectedNotMarked.getMarkedTourists().stream().map(User::getId).anyMatch(t → t.equals(subject.getId())));
}
```

Questo metodo si occupa di testare l'esito positivo, in uno scenario realistico, di una chiamata al metodo `markAsCompleted` del servizio `TourService`.

Viene recuperato un turista dall'`EntityMock`, quindi, a partire dall'elenco dei tour da egli segnati come completati, si ricava uno di quelli non marcati. Il motivo per cui è necessaria questa ricerca è rappresentato dal fatto che, in caso si scegliesse arbitrariamente un tour da marcare, si potrebbe finire per testare un caso differente, ovvero quello della marcatura di un tour già segnato. Infatti, per quanto questa situazione sia gestibile dal metodo, non è tuttavia l'oggetto del test attuale.

Si procede quindi con la chiamata al metodo oggetto di test e, infine, si fanno delle asserzioni. In particolare, ci aspettiamo che all'interno dell'elenco dei turisti che hanno segnato il tour in questione come completato, vi sia il soggetto iniziale. In caso positivo, il test ha successo.

Esempio di test negativo

Tests `TourService.markAsCompleted(Tour, Tourist)` in the failing case because the tourist has already marked the tour as completed.
Ensures that a `InteractionAlreadyPerformedException` is thrown.

no usages Francesco Balducci

```
@Test
void noMultipleMarkAsCompleted() {
    // setup
    // find a tour that is already marked and one of those tourists (otherwise the test would fail)
    Tour expectedMarked = mock.getAllTours().stream().filter(t ->  t.getMarkedTourists().size() > 0).findAny().orElseThrow();
    Tourist subject = expectedMarked.getMarkedTourists().stream().findAny().orElseThrow();
    // under test and assertion
    assertThrows(InteractionAlreadyPerformedException.class, () -> tourService.markAsCompleted(expectedMarked, subject));
}
```

Nel particolare caso del metodo `markAsCompleted`, avendo creato un'eccezione personalizzata che viene sollevata in caso un turista provi a marcare più volte un tour come completato, ho ritenuto importante testare separatamente anche questa situazione.

Analogamente a prima, recupero un tour da `EntityMock` ed estraggo un turista dall'elenco di quelli che lo hanno segnato come completato, quindi provo ad eseguire nuovamente l'operazione.

In questo caso l'asserzione consiste nel sollevamento dell'eccezione che, se verificata, fa completare con successo il test.

Rapporto sulla coverage

In base alle informazioni fornite dall'IDE utilizzato, dopo aver eseguito l'intera collezione di test creata, si ottengono i seguenti risultati di coverage del codice:

Element	Class, %	Method, %	Line, %
com	65% (75/115)	42% (207/492)	37% (314/841)
what2see	65% (75/115)	42% (207/492)	37% (314/841)
controller	85% (6/7)	18% (6/32)	4% (6/137)
dto	17% (7/41)	12% (25/196)	12% (25/196)
exception	100% (3/3)	12% (3/25)	26% (8/30)
mapper	100% (13/13)	37% (13/35)	10% (18/175)
model	86% (25/29)	70% (96/136)	71% (98/138)
repository	100% (0/0)	100% (0/0)	100% (0/0)
service	100% (18/18)	96% (60/62)	98% (150/152)
utils	100% (1/1)	100% (1/1)	71% (5/7)
ServletInitializer	0% (0/1)	0% (0/1)	0% (0/1)
WhatToSeeApisApplication	100% (2/2)	75% (3/4)	80% (4/5)

Le uniche due righe di codice non incluse si riferiscono alle due proprietà nella classe `TourSearch`, della cui copertura però ne ho la certezza come conseguenza della copertura della funzionalità di ricerca nel suo complesso.

Per ulteriori dettagli sulla coverage si può consultare la documentazione HTML (aprendo il file `./testing/index.html`).

SVILUPPO FRONT-END

Panoramica

Per lo sviluppo front-end ho scelto di utilizzare il framework Angular (versione 15), che si basa sulla piattaforma Node (versione 18.12) e sul linguaggio di programmazione TypeScript (versione 4.8). Esso consente di creare piattaforme web altamente strutturate e adatte a progetti medio-grandi.

Differentemente dai siti web tradizionali (con più documenti), in questo caso si parla di SPA (single page application). In questo caso, il client scarica un solo documento HTML, CSS e JS che progressivamente cambia aspetto e URL, creando la sensazione di navigare un normale sito caratterizzato da ridottissimi ritardi nel passaggio da un'area dell'applicazione ad un'altra.

Un altro vantaggio consiste nella riutilizzabilità del codice per mezzo dei componenti, che possono essere posizionati in più parti dell'applicazione senza doverli duplicare.

Il linguaggio di programmazione TypeScript è un'estensione di JavaScript che vi potenzia l'apparato di tipizzazione. Nativamente, infatti, quest'ultimo è un linguaggio debolmente tipizzato e ciò lo rende poco adatto alle applicazioni più complesse e robuste. Con TypeScript, si ricevono errori di compilazione dall'IDE in caso, ad esempio, si provi ad assegnare un tipo non conforme a quello dichiarato.

È importante sottolineare che TypeScript non può essere interpretato dai browser e per questo viene "traspilato" in JavaScript al momento della build.

Come ulteriori librerie ho utilizzato:

- Tailwind CSS (per evitare di dover dichiarare classi CSS elementari ed avere quasi tutti gli stili a colpo d'occhio direttamente nel template);

- Angular Material (per i componenti di base come il modale, la snackbar e la sidebar);

Suddivisione delle cartelle

Con il fine di mantenere in ordine i file in relazione alla loro tipologia, li ho raggruppati secondo la seguente suddivisione:

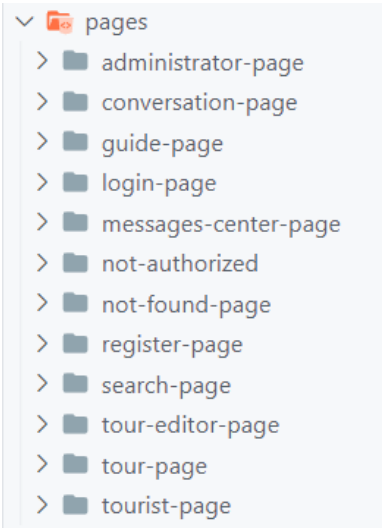
1. Pages
2. Components

} View + Controller

3. DTOs
4. Services

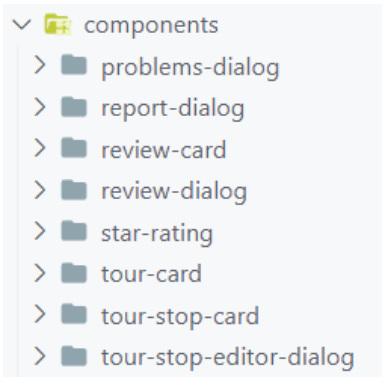
} Model

Pages



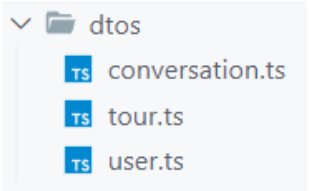
Contiene i components che in particolare sono oggetto di routing (privi di selettore), per distinguerli da quelli che sono invece "blocchi di costruzione". Il loro ruolo è quindi mettere insieme tutti i pezzi necessari per formare una pagina che sia correttamente visualizzabile dall'utente.

Components



Sono blocchi di base, dotati di selettore, che vengono richiamati in pages o altri componenti. Devono gestire un'adeguata interfaccia di comunicazione con l'esterno e anche una visualizzazione che si adatti bene ai diversi contesti d'utilizzo.

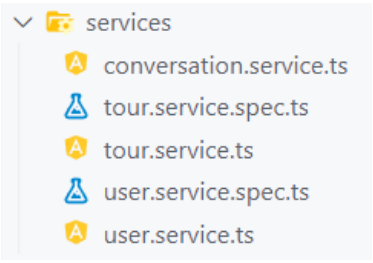
DTOs



Sono l'omologo TypeScript del package già presentato per il back-end. A differenza di quest'ultimo, però, non troviamo anche un package per gli entity dato che questi

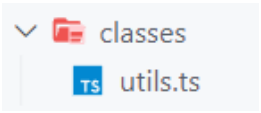
file contengono già i tipi più completi che si ha necessità di gestire da questa side. Un'altra differenza con Java consiste nella pratica che ho adottato, abbastanza comune, di racchiudere più tipi per ciascun file anziché uno solo.

Services



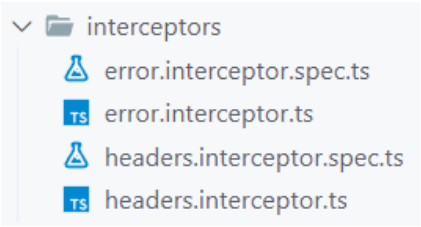
Anche in questo caso troviamo un'analogia con il back-end, dato che queste classi possono essere intese come l'unione di controller (metodi che comunicano con il back-end) e service (metodi della logica di business privi di stato).

Classes



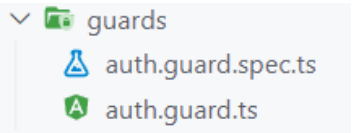
Si tratta di classi di utilità con metodi statici abbastanza generici da non costituire un service specifico.

Interceptors



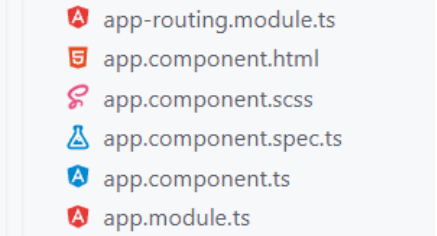
Sono delle classi gestite dal framework che vengono eseguite ad ogni richiesta HTTP entrante o uscente, per modificare la loro forma o comportamento.

Guards



Queste classi vengono invocate dal framework prima di navigare alle rotte a cui sono associate, in modo che eseguano una logica personalizzata che decreti se si possa effettivamente raggiungere tale rotta o meno.

Default package



Rappresentano il punto di accesso dell'applicazione nonché la struttura portante di ogni pagina.

Illustrazione della classe principale per ciascuna cartella

Per ciascuna cartella presentata nella sezione precedente, propongo di seguito una spiegazione dettagliata del file più rilevante in termini di funzionalità. In caso di ripetizione di codice, per brevità, questo verrà descritto solo nella prima occorrenza.

Per ulteriori dettagli si può consultare il JSDoc presente nel codice e/o la relativa documentazione HTML (aprendo il file `./coverage/index.html`).

Page (TourPageComponent)

Questa pagina si occupa di mostrare un tour, nei suoi vari aspetti e funzionalità, all'utente.

Esempio di template

```
<h3 class="!my-2">Tappe ({{ tour.stops.length }}):</h3>
<div class="flex flex-col gap-y-2">
    <app-tour-stop-card *ngFor="let stop of tour.stops" [data]="stop"></app-tour-stop-card>
</div>

<h3 class="!my-2">Dettagli complessivi:</h3>
<ul class="list-disc list-inside ml-2">
    <li>Città: {{ tour.city.name }}</li>
    <li>Durata approssimativa: {{ tour.approxDuration }}</li>
    <li>Costo approssimativo: {{ tour.approxCost | currency }}</li>
    <li>Segnato come completato da {{ tour.markedAsCompletedCount }} turisti</li>
    <li>Tema: {{ tour.theme.name }}</li>
    <li>
        <div class="inline-flex items-center">
            <span>Tags:</span>
            <mat-chip-listbox class="ml-2">
                <mat-chip *ngFor="let tag of tour.tags">{{ tag.name }}</mat-chip>
            </mat-chip-listbox>
        </div>
    </li>
</ul>
```

In questa porzione del template, vengono interpolate le informazioni del tour mostrando quelle elementari (durata, città, ecc.) direttamente in un elenco, mentre per le fermate si riutilizza il componente card che incapsula la logica e l'aspetto di una fermata.

Dipendenze

```
constructor(
  private router: Router,
  private route: ActivatedRoute,
  private tourService: TourService,
  private conversationService: ConversationService,
  private dialogService: MatDialog,
  private notify: MatSnackBar,
) {}
```

Ho dichiarato le dipendenze necessarie per permettere al componente di implementare le sue funzionalità, tra cui: navigare in altre pagine, recuperare i parametri della rotta (per l'ID del tour richiesto), interrogare il back-end per le informazioni sul tour e le conversazioni delle guida, mostrare modali e la snackbar.

Esempio di metodo

```
/**
 * Tourist requested to mark the tour as completed
 */
markCompleted(): void {
  this.tourService.markAsCompleted(tourId: this.tour!.id).subscribe(next: () => {
    // tour marked as completed, notify the user and refresh the available actions
    this.notify.open(message: 'Segnato come completato con successo!', action: undefined, config: { panelClass: 'success-snackbar' });
    this.afterAction()
  })
}
```

Quando nel template capita l'evento del click sul bottone per segnare un tour come percorso, questo metodo viene richiamato. Esso invoca il metodo `markAsCompleted` di `TourService` passandogli l'ID del tour caricato. L'informazione sull'utente richiedente verrà inserita automaticamente dall'interceptor. Con il `subscribe`, si dichiara la funzione che verrà eseguita in maniera asincrona al ricevimento della risposta positiva del back-end, ovvero la comparsa di una snackbar per notificare l'utente dell'avvenuta azione. Viene inoltre richiamata la funzione che comporta l'aggiornamento delle azioni successivamente intraprendibili dall'utente.

Component (TourCardComponent)

Questo componente riutilizzabile si occupa di mostrare sinteticamente un tour, ad esempio come risultato di ricerca.

Template

```
<mat-card>
    <mat-card-header>
        <mat-card-title class="!inline">{{ data.title }}</mat-card-title>
        <mat-card-subtitle class="pl-2 !inline">
            {{ "di " + data.author.firstName + " " + data.author.lastName }}
        </mat-card-subtitle>
    </mat-card-header>
    <mat-card-content>
        <h3 class="!mb-2">
            {{ data.city.name + " &#x25cf; " + data.approxDuration + " &#x25cf; " +
            + data.stops.length + " tappe &#x25cf; " + (data.approxCost | currency) +
            + " &#x25cf; " + data.theme.name }}
        </h3>
        <h3>
            {{ data.reviews.length }} recensioni &#x25cf; percorso da
            {{ data.markedAsCompletedCount }} turisti
        </h3>
    </mat-card-content>
</mat-card>
```

Ho utilizzato il componente card di Angular Material come struttura di base, interpolando le informazioni che ho ritenuto essenziali da mostrare riguardo un tour.

Metodo

```
export class TourCardComponent {

  @Input() data!: Tour

}
```

In questo caso, la logica è estremamente limitata, dovendo soltanto ricevere, come binding in ingresso dal componente richiedente, l'oggetto tour da mostrare.

DTO (Tour)

Questo file contiene le dichiarazioni del tipo `Tour` (dal punto di vista di questa side, corrispondente al DTO del back-end) e di tutte le entità ad esso associate.

Nel contesto di Angular, si tratta quasi sempre di "interfacce" anziché di classi, ovvero di tipi che impongono ad una variabile la presenza di proprietà di un certo tipo al suo interno. Nonostante ci sia la possibilità di dichiararvi metodi, si preferisce spostare questa logica all'interno di un componente o di un servizio, dato che in questi casi è più facile eseguire il debug essendo eseguiti nel contesto gestito da Angular.

Esempio di tipizzazione

```
export interface Tour {
    id: number
    author: User
    title: string
    description: string
    isPublic: boolean
    city: City
    tags: Tag[]
    theme: Theme
    approxCost: number
    approxDuration: string
    creationDate: Date
    stops: Stop[]
    reviews: Review[]
    markedAsCompletedCount: number
}
```

Questa interfaccia definisce le proprietà di un `Tour` per come ricevuto dal back-end. Ogni campo indica come chiave il nome della proprietà e di seguito il suo tipo. Troviamo anche proprietà che sono a loro volta oggetti di cui esso è composto.

Service (TourService)

Questo servizio si occupa di interfacciarsi, tramite API, con le funzionalità back-end relative o correlate a un tour.

Annotation

```
/**
 * Service that handles the tour features.
 */
@Injectable(options: {
  providedIn: 'root'
})
export class TourService {
```

`@Injectable` Annotation di Angular che contrassegna la classe come istanziata in automatico dal framework e iniettabile in tutte le classi che dovessero indicarla come dipendenza nel costruttore.

Dipendenza

`constructor(private http: HttpClient) { }` Avendo bisogno di eseguire richieste HTTP, ho indicato come dipendenza il corrispondente servizio messo a disposizione dal framework.

Esempio di metodo

```
/**
 * Mark the given tour as completed.
 * Only available for tourists.
 * @param tourId the id of the tour to mark as completed
 * @returns completes when the action is done
 */
public markAsCompleted(tourId: number): Observable<void> {
  return this.http.post<void>(url: environment.apiUrl + '/tour/' + tourId + '/completed', body: null)
}
```

A partire dall'ID del tour, viene eseguita una richiesta HTTP di tipo POST all'URL indicato nella variabile d'ambiente `apiUrl`, che punta al server in esecuzione locale con il back-end. Non è previsto nessun body né nella richiesta né nella risposta. L'informazione sull'utente richiedente verrà inserita automaticamente dall'interceptor.

Class (Utils)

Questa classe si occupa di fornire funzioni statiche di utilità comune che ho ritenuto sufficientemente generiche da non inserire in un servizio ad-hoc.

Esempio di metodo

```
/**
 * Finds invalid controls in a form and returns their names.
 * @param form form to be checked
 * @returns names of invalid controls
 */
public static findInvalidControls(form: FormGroup): string[] {
    const invalid = [];
    const controls = form.controls;
    for (const name in controls) {
        if (controls[name].invalid) {
            invalid.push( ... items: name);
        }
    }
    return invalid;
}
```

Questo metodo prende in ingresso un FormGroup, ovvero la rappresentazione lato codice di un form dinamico, e itera su tutti i controlli di primo livello che esso contiene, per ritornare i nomi di quelli che contengono errori di validazione. Ho utilizzato questo metodo principalmente per motivi di debug.

Interceptor (HeadersInterceptor)

Questa classe è annotata con `@Injectable` (come un servizio) ed estende `HttpInterceptor`. Viene individuata ed invocata da Angular per eseguire della logica all'esecuzione di ogni nuova richiesta HTTP, con l'obiettivo di aggiungervi headers rilevanti ai fini dell'autenticazione.

Dipendenza

```
constructor(private userService: UserService) { }
```

Avendo bisogno di recuperare informazioni sull'eventuale utente loggato in sessione, questo servizio dipende da `UserService`.

Metodo

```
intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
  const loggedUser = this.userService.getSession();
  if(loggedUser =/= null) {
    request = request.clone(update: {
      setHeaders: {
        Authentication: loggedUser.id.toString()
      }
    });
  }
  return next.handle(req: request);
}
```

Questo è il metodo che bisogna implementare dall'interfaccia. Ha come parametri la richiesta che sta partendo, un oggetto che consente di farla proseguire e si aspetta come ritorno un oggetto analogo al primo parametro,

con eventuali modifiche. Viene recuperato l'utente dalla sessione e, se esiste (è stato eseguito il login), viene aggiunto un header personalizzato alla richiesta denominato `Authentication` a cui corrisponderà l'ID del suddetto utente. Tale header è richiesto dal back-end come requisito per accedere alle API autenticate.

Guard (AuthGuard)

Similarmente al caso precedente, anche questa classe è annotata con `@Injectable` ed estende `CanActivate`. Viene individuata ed invocata da Angular per eseguire della logica prima di (eventualmente) raggiungere una rotta che ha indicato questa classe come "barriera" di ingresso.

Esempio di metodo

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
  const loggedUserRole = this.userService.getSessionRole();
  if(loggedUserRole !== null) { // is logged?
    const routeRole: UserRole[] = route.data['role'];
    if(!routeRole || (routeRole.indexOf(searchElement: loggedUserRole) !== -1)) { // if role is defined for role, does it match?
      return true;
    } else {
      this.router.navigate(commands: ['/not-authorized']);
    }
  } else {
    this.router.navigate(commands: ['/login']);
  }
  return false;
}
```

Il metodo da implementare dall'interfaccia ha come parametri uno snapshot della rotta da attivare e dello stato del router (non utilizzato) e si aspetta un booleano che obbliga a proseguire o meno con l'attivazione della rotta.

Come nel precedente caso, si recupera da `UserService` l'eventuale utente in sessione. Se questo non esiste, l'utente è re-instradato verso la rotta di login, dato che tutte le funzionalità richiedono l'autenticazione.

Se invece è loggato, si confronta il suo ruolo con quello opzionalmente indicato come requisito a livello di rotta (ad esempio, la pagina di modifica di un tour è riservata alle sole guide) e, se il suo vi è indicato, è autorizzato a proseguire, altrimenti verrà re-instradato verso una rotta predisposta per comunicare la mancata autorizzazione.

Default package (AppRoutingModule)

Questo modulo è richiesto dal modulo di root `AppModule` e ha la specifica funzione di gestire le rotte disponibili nell'applicazione.

Annotation

```
/**
 * Module that contains all the routes of the application.
 */
@NgModule(obj: {
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

`@NgModule`

Essendo un modulo, ha bisogno di questa annotation di Angular, nel cui oggetto di configurazione si specifica che esso ha come input le rotte da configurare e come ouput il modulo di routing configurato.

Esempio di rotta

```
{ path: 'tour', children: [
  { path: 'create', component: TourEditorPageComponent, canActivate: [AuthGuard], data: { title: 'Crea tour', role: [UserRole.GUIDE] }},
  { path: ':id/edit', component: TourEditorPageComponent, canActivate: [AuthGuard], data: { title: 'Modifica tour', role: [UserRole.GUIDE] } },
  { path: ':id', component: TourPageComponent, canActivate: [AuthGuard], data: { title: 'Visualizza tour' }},
]},
```

Questa rotta viene attivata da tutti gli URL che iniziano con `/tour` e prosegue con il matching con una delle rotte figlie. Ad esempio, `/tour/1` attiva la pagina `TourPageComponent` già descritta in precedenza, a patto che l'`AuthGuard` lo consenta secondo le regole previste circa l'utente loggato e il suo ruolo (deve essere una guida). È anche presente un titolo che viene autonomamente recuperato dall'`AppComponent` per mostrarlo sulla navbar.

Design pattern

Come già introdotto nella sezione dedicata al back-end, i design pattern rappresentano una soluzione efficace a problemi ricorrenti. Anche nel caso del front-end, ho deciso di utilizzare e/o implementarne alcuni. I primi tre sono integrati nel framework scelto.

Dependency injection (comportamentale)

Nelle precedenti sezioni ho mostrato porzioni di codice con le quali i componenti e i servizi hanno richiesto l'iniezione di servizi. È infatti messo a disposizione dal framework un meccanismo che, al momento dell'attivazione

di un componente, gli fornisce tutte le dipendenze richieste. La particolarità consiste nella possibilità di sovrascrivere la modalità di default, dichiarando token iniettabili ad-hoc, tra cui ad esempio una semplice stringa. Sono quindi ammesse funzionalità avanzate come la possibilità di fornire un medesimo servizio in moduli differenti dando alla dipendenza forme concrete differenti in ciascuno.

Observer (comportamentale)

La documentazione ufficiale consiglia fortemente l'adozione della libreria RxJS. Essa fornisce un'alternativa molto più scalabile e flessibile alle classiche `Promise` di JavaScript.

Il pattern risolve la necessità di avere una fonte asincrona di informazione, alla quale una o più parti dell'applicazione sono interessate. Per questo motivo, sincronicamente esse si sottoscrivono all'evento indicando la funzione che vogliono venga successivamente chiamata con i dettagli dell'evento.

Ecco che, ad esempio, il metodo di `HttpClient` ritorna un `Observable<T>` dove `T` è il tipo atteso nel body della risposta. Su tale oggetto, il componente invoca il metodo `subscribe()` che concretamente fa partire la richiesta. Come parametro esso riceve una arrow function (dichiarata sul momento) e che verrà richiamata alla ricezione del risultato.

Singleton (creazionale)

Nel caso generale, un servizio iniettabile da Angular attraverso la dependency injection viene anche istanziato all'avvio dell'applicazione in modo da mantenere una singola istanza comune a tutti gli utilizzatori. Come nel caso del back-end, condizione importante per l'applicabilità di questo pattern è l'assenza di uno stato, che altrimenti impedirebbe di avere delle funzioni pure, la cui uscita quindi dovrebbe dipendere unicamente dagli ingressi.

La presenza di questo pattern consente di aumentare la condivisione delle già limitate risorse a disposizione del browser utente.

MVC (architetturale)

Ho ritenuto fondamentale suddividere il codice in modo da distinguere il più possibile le componenti di View, Controller e Model.

Possiamo individuare nei componenti la parte di View, dato che contengono il markup e lo stile che verrà direttamente mostrato all'utente. Esso è anche formato da un file TypeScript che ricopre il ruolo di Controller dato che contiene la logica di gestione dei form e le validazioni. In caso ci fossero state funzioni più complesse e indipendenti dallo stato del componente, avrei valutato la possibilità di esternalizzarle in un servizio apposito.

I servizi e i file con le dichiarazioni dei tipi svolgono il ruolo di Model dato che consentono l'accesso ai DTO forniti dal back-end.

SVILUPPI FUTURI

Alcune funzionalità interessanti che si potrebbero sviluppare nel futuro per ampliare ulteriormente le funzionalità potrebbero essere:

- Rendere il sito installabile come app ibrida;
- Implementare un tema personalizzato ad Angular Material;
- Integrare un servizio di mappe per consentire la navigazione passo-passo fra le tappe;
- Consentire il caricamento di immagini per individuare meglio le tappe;
- Aggiungere la generazione di un QR code per condividere il tour su altre piattaforme;
- Aggiungere possibilità di recensire la guida;
- Possibilità di creare un tour visualizzabile a pagamento e/o con guida in loco;
- Autenticazione con JWT e SSO sfruttando la libreria Spring Security;
- Utilizzo di database non relazionali per la memorizzazione delle conversazioni;
- Aggiunta di API pubbliche senza autenticazione, in modo da fornire funzionalità anche ad utenti anonimi;
- Aggiunta di uno stato bozza alla creazione di un tour;
- Utilizzo dei Web Socket per la ricezione dei messaggi;
- Condivisione delle tappe tra i tour;
- Creazione di una pagina vetrina per guida e turista;

CONCLUSIONI

Nel complesso, l'applicazione risulta reattiva e piacevole all'utilizzo, anche da device di natura differente (smartphone, laptop, desktop, ecc.). Ho cercato di privilegiare l'uniformità dell'esperienza utente e la sua utilizzabilità, evitando di mostrare un'interfaccia utente complessa e macchinosa. Le funzionalità disponibili al momento sono essenziali ma già sufficienti per iniziare a offrire il servizio ad una platea ristretta di utenti finali.

Ripercorrendo i passi compiuti, la progettazione mi ha aiutato a formalizzare il problema in modo puntuale e metodico. Durante lo sviluppo, ho cercato di implementare ogni feature limitando al massimo la dipendenza tecnologica tra front-end e back-end, effettuando verifiche sui dati da entrambi i lati. Inoltre, ho cercato di approfondire e applicare le best-practice consigliate dalla letteratura, dalla documentazione e dalle community online. Infine, attraverso i test, ho predisposto una soglia minima garantita di affidabilità del codice in esecuzione.

Non mancano idee per gli sviluppi futuri, al cui supporto sono stati messi in atto i design pattern presentati. Le tecnologie estremamente attuali utilizzate rendono la base di codice (e le relative competenze) particolarmente spendibili nel marcato attuale.

Grazie a questo progetto, ho potuto mettere in gioco tutte le competenze di progettazione e programmazione apprese nel corso di questi tre anni di ingegneria. Mi ritengo soddisfatto del risultato ottenuto e mi piacerebbe ricevere riscontro riguardo le occasioni di miglioramento.

Francesco Balducci