



# Università degli Studi di Salerno

Corso di Ingegneria del Software  
Classe 1 Resto 0  
Corso di Laurea in Informatica  
A.A. 2022/23

## Quiad Object Design Document

Versione 6.0  
30/12/2022



**Partecipanti al progetto e scriventi**

Nome	Matricola
Di Pasquale Valerio	0512110638
Troisi Vito	0512109807

---

**Revision History**

Data	Versione	Descrizione	Autore
30/11/2022	1.0	Prima stesura ODD e Indice	D.P.V. T.V.
02/12/2022	2.0	Sezione introduttiva e vincoli in OCL	D.P.V. T.V.
08/12/2022	2.1	Scheletro sezione Interfaccia delle classi da integrare con modelli ristrutturati	T.V.
09/12/2022	3.0	Trade-offs, organizzazione in directory	T.V.
10/12/2022	4.0	Primi contratti classi solution domain, caching server side, sviluppati CD-OD 1, CD-OD 2, SD-OD 1	D.P.V. T.V.
15/12/2022	4.1	Fine contratti	D.P.V.
15/12/2022	4.2	Note sul mapping	T.V.
16/12/2022	5.0	Aggiunta diagrammi realizzati; note sui design pattern	T.V.
23/12/2022	5.1	Aggiunta nomenclatura standard per operazioni	D.P.V.
29/12/2022	5.2	Client class diagram: invertito verso dipendenza tra service e entity	T.V.
29/12/2022	5.3	Aggiunto SD-OD 2	D.P.V. T.V.
30/12/2022	6.0	Refactoring CD-OD: rimozione classe UtenteBase non necessaria	T.V.

## Indice

1. Introduzione	
1.1 Overview.....	p.4
1.2 Trade-offs di Object Design.....	p.4
1.2 Linee guida per la documentazione delle interfacce.....	p.4
1.4 Definizioni, acronimi, abbreviazioni.....	p.5
1.5 Riferimenti.....	p.5
2. Organizzazione delle directories.....	p.6
3. Interfacce delle classi e modelli.....	p.7
4. Note sui Design Pattern utilizzati.....	p.25
5. Note sul mapping dei modelli sul codice.....	p.25

## Overview

Nel presente documento, sarà esplicitata la progettazione di dettaglio del sistema Quiad, per la gestione del proprio albero genealogico.

Lo Object Design Document vuole presentare i trade-offs di progettazione e, per quanto concerne la *slice* del sistema che sarà implementata, la specifica delle interfacce dei moduli e l'organizzazione delle classi dei medesimi. Seguono la presente sezione introduttiva, alcune linee guida utili per la comprensione delle scelte di design e di specifica delle interfacce suddette.

## Trade-offs di Object Design

In aggiunta ai trade-offs evidenziati in fase di progettazione architetturale se ne evidenziano alcuni relativi alla progettazione di dettaglio dello stralcio del sistema presentato nei paragrafi seguenti.

- **Spazio occupato VS Aggiornamento continuo**

Per le medesime ragioni illustrate in fase di progettazione di sistema e viste le scelte illustrate nel RMD sezione 2.1, sarà prevista una cache nella quale conservare i documenti recentemente acceduti. Ciò consente di evitare l'uso della rete per accedere al database e poi al file system per recuperare documenti i quali, si osserva, non sono generalmente soggetti a cambiamenti.

## Linee guida per la documentazione di modelli e interfacce

Le interfacce delle classi dei moduli che saranno soggetti ad implementazione, saranno specificate tenendo conto delle seguenti prassi:

- I nomi di attributi, metodi e parametri saranno esplicitati in camel-case.
- I nomi dei metodi getter/setter legati ad un attributo seguiranno la forma "get...()" e "set...()", imponendovi il nome dell'attributo.

- Precondizioni, postcondizioni ed invarianti di classe saranno esplicitati mediante OCL (Object Constraint Language).
- Le classi Route, Service, Controller e Model utilizzate per implementare i servizi dei sottosistemi saranno precedute dal nome della classe del dominio applicativo (od eventualmente da un nome significativo per un insieme di classi) con le quali interagiscono.
- I nomi delle operazioni seguono lo standard "entity:operation" (e.g. "node:create").

### **Definizioni, acronimi, abbreviazioni**

Una lista alfabetizzata di definizioni ed acronimi utili per la lettura della presente:

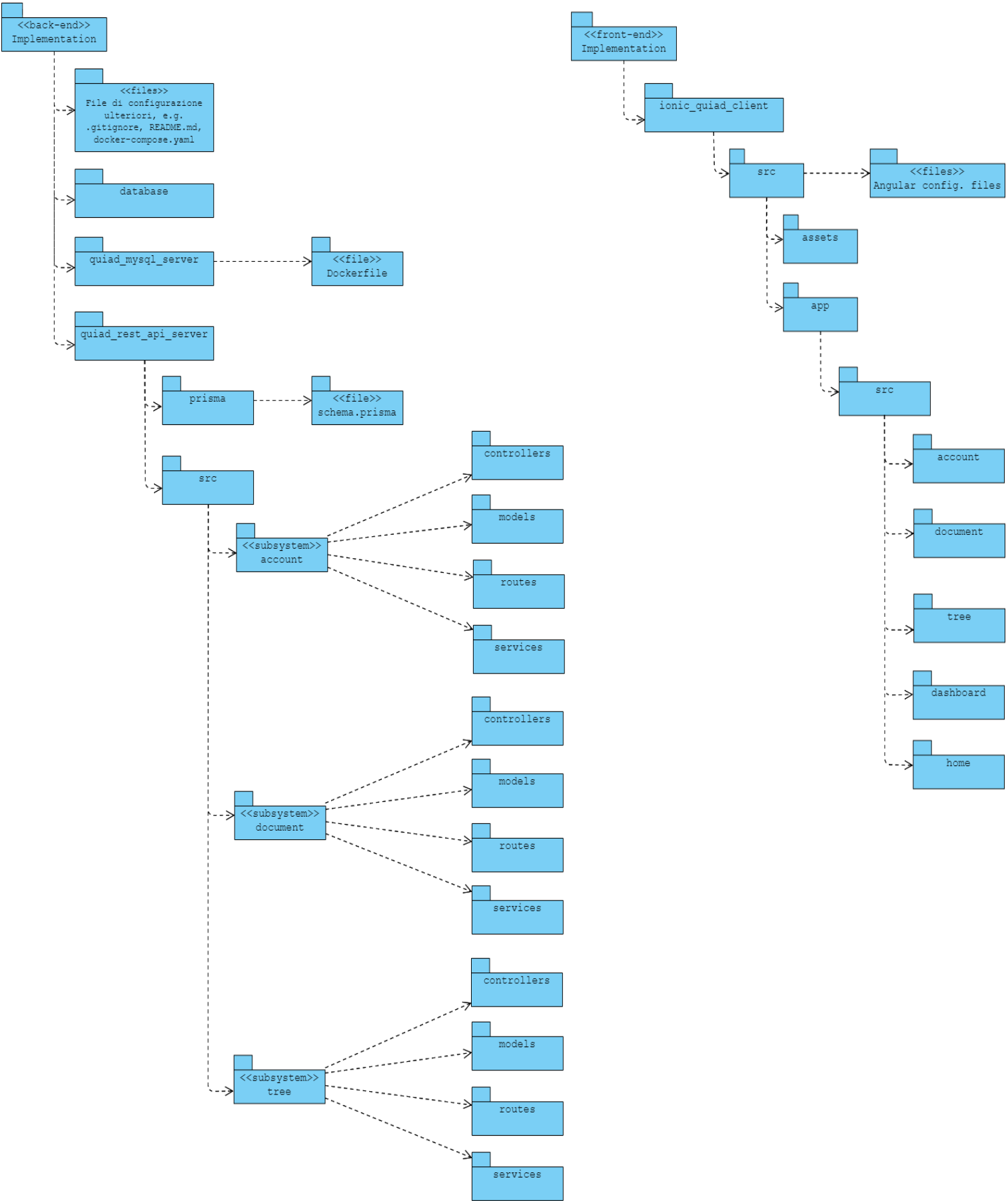
- CD-OD X: Class Diagram [relativo a] Object Design con identificativo X.
- SD-OD X: Sequence Diagram [relativo a] Object Design con identificativo X.
- RAD: Requirements Analysis Document.
- RMD: Rationale Management Document, i.e. il documento in cui è riportato il razionale dietro alcune scelte fatte in fase di progettazione.
- OCL: Object Constraint Language, il quale sarà impiegato per esplicitare i contratti di ogni metodo, e per ciascuna classe.
- ODD: Object Design Document, il presente documento.
- SDD: System Design Document, i.e. il documento di progettazione architetturale del sistema.

### **Riferimenti**

Al fine di garantire una migliore comprensione dello ODD, si invita ad osservare il Requirements Analysis Document, in particolar modo le sezioni 3.5.3 e 3.5.4, rispettivamente legate alla modellazione ad oggetti ed alla modellazione dinamica.

Si invita inoltre a consultare il SDD per analizzare le scelte legate alla progettazione di sistema, quale la gestione dei dati persistenti, esplicitata alla sezione 2.4.

Organizzazione delle directory

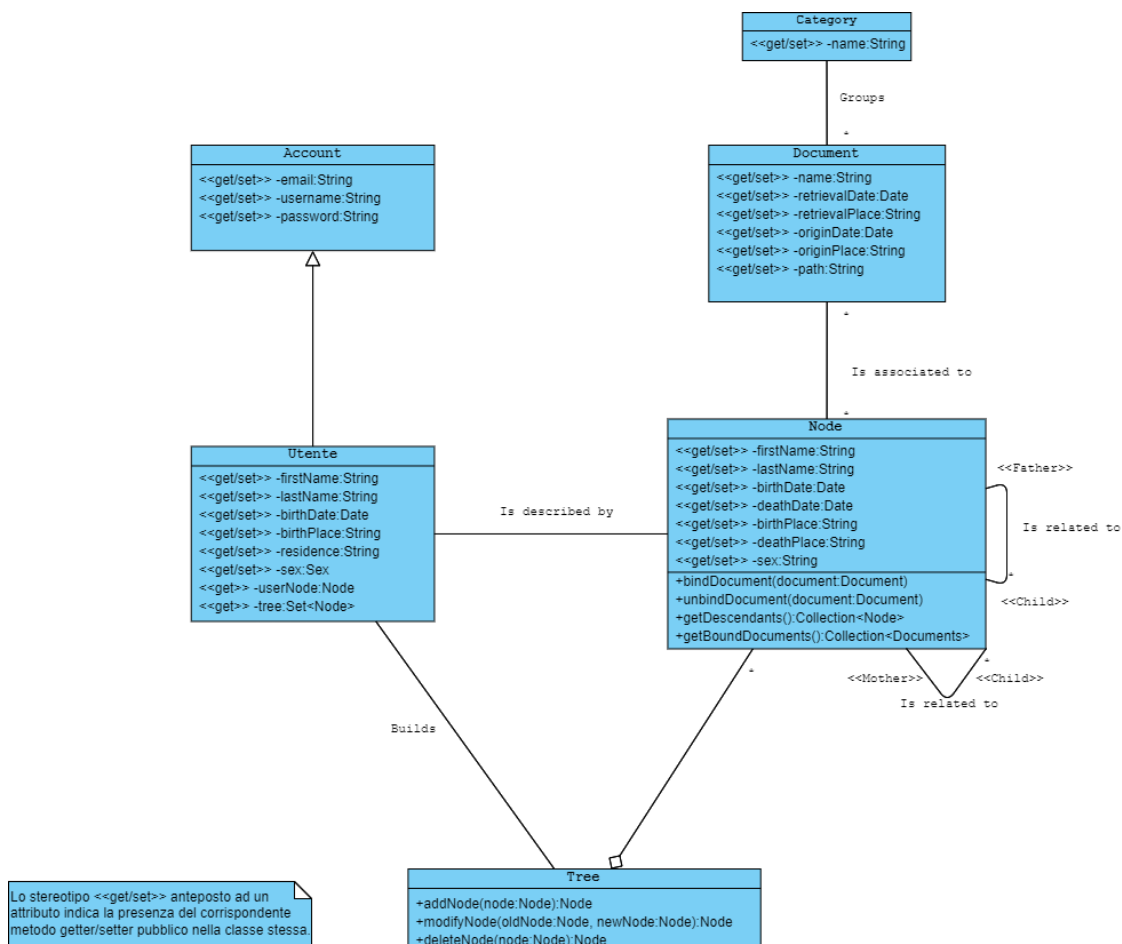


Per quanto concerne le linee guida generali dell'organizzazione delle classi del progetto in directory, si osservi che ciascun sottosistema (riferirsi al SDD, sezione 2.2) sarà mappato su una precisa directory. Nel dettaglio, la struttura si presenta come mostrato in precedenza. In tale package diagram, ogni package è da intendersi come una directory, ed ogni arco orientato come la relazione "contiene".

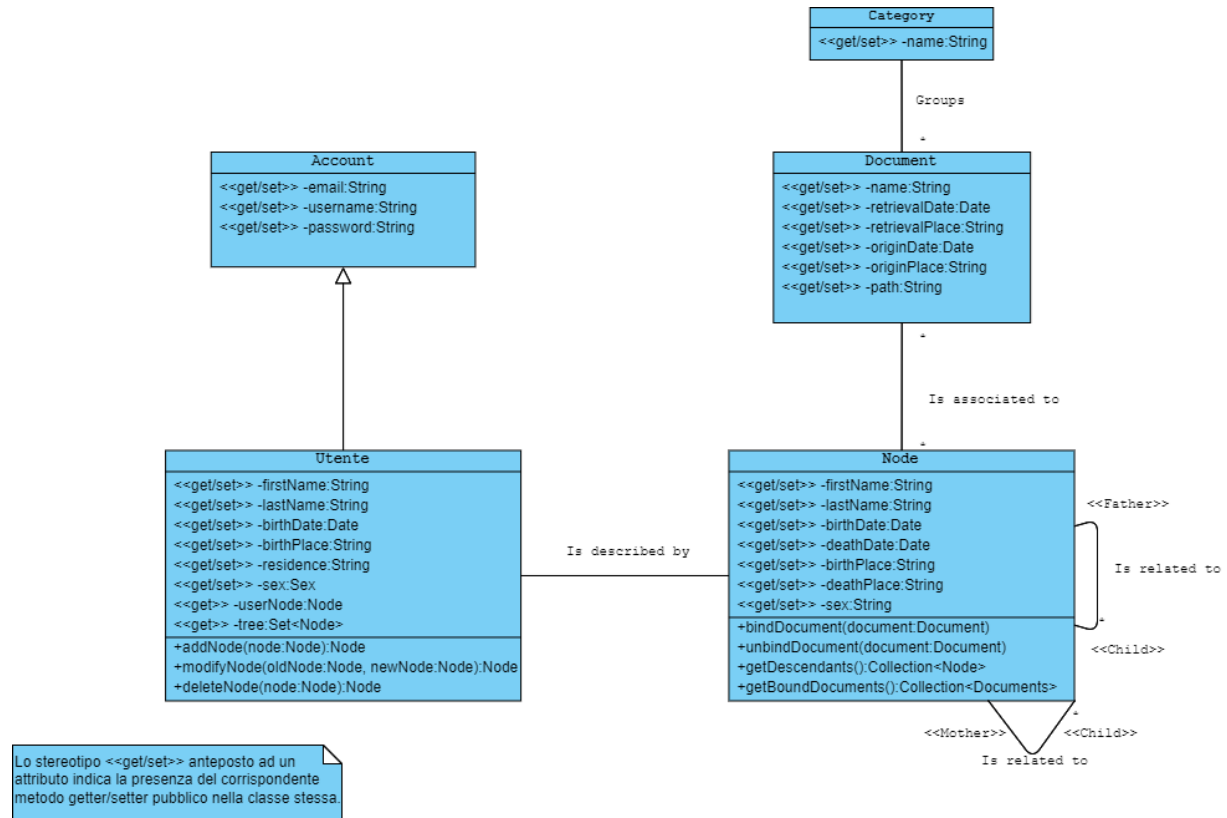
## Interfacce delle classi e modelli

Il Class Diagram presentato nel RAD (paragrafo 3.5.3) richiede una ristrutturazione, al fine di evitare futili ridondanze. Un utente base (o curatore) può infatti essere ben rappresentato dal proprio nodo, con ogni informazione inserita dall'utente stesso in fase di registrazione. Conseguentemente, è stata definita una nuova astrazione, Account, che ristruttura lo schema come presentato di seguito. Al medesimo sono state aggiunte le informazioni circa visibilità e tipi dei membri delle classi.

Si osservi che, per implementazioni future, la classe Curator estenderà Utente e la classe Supervisor estenderà Account, così da mantenere la consistenza (e la stessa semantica) rispetto al Class Diagram descritto in fase di Object Modeling.



Inoltre, si osservi che l'albero e l'utente sono coinvolti in un'associazione 1-1. E' possibile accorpare l'albero nell'utente, che si occuperà di gestire la collezione di nodi.



Seguono i contratti di classi e metodi proposti, specificati mediante OCL.



**Classe Utente:**

```
context Utente::addNode(n) pre:
    n != null  $\wedge$ 
    n.getFather() != n  $\wedge$ 
    n.getMother() != n  $\wedge$ 
    tree->include(n.getFather())  $\wedge$ 
    tree->include(n.getMother())
```

```
context Utente::addNode(n) post:
    tree->include(n)  $\wedge$ 
    tree->n.getFather().getDescendants()->asSet()->include(n)  $\wedge$ 
    tree->n.getMother().getDescendants()->asSet()->include(n)
```

```
context Utente::modifyNode(m,n) pre:
    n != null  $\wedge$ 
    m != null  $\wedge$ 
    tree->include(m)  $\wedge$ 
    m.getFather() = n.getFather()  $\wedge$ 
    m.getMother() = n.getMother()
```

```
context Utente::modifyNode(m,n) post:
    !tree->include(m)  $\wedge$ 
    tree->include(n)
```

```
context Utente::deleteNode(n) pre:
    n != null  $\wedge$ 
    tree->include(n)  $\wedge$ 
    n != getUserNode()
```

```
context Utente::deleteNode(n) post:
    !tree->include(n)  $\wedge$ 
    forAll(s|n.getDescendants()) !tree->include(s)
```

**Classe Node:**

```
context Node::bindDocument(d) pre:
    d != null ∧
    !self.getBoundDocuments()->asSet()->include(d)
```

```
context Node::bindDocument(d) post:
    self.getBoundDocuments()->asSet()->include(d)
```

```
context Node::unbindDocument(d) pre:
    d != null ∧
    self.getBoundDocuments()->asSet()->include(d)
```

```
context Node::unbindDocument(d) post:
    !self.getBoundDocuments()->asSet()->include(d)
```

---

Il modello dei dati può poi essere mappato su un DB relazionale, in accordo con quanto esplicitato nel SDD, sezione 2.4.

**Mapping classi-relazioni**

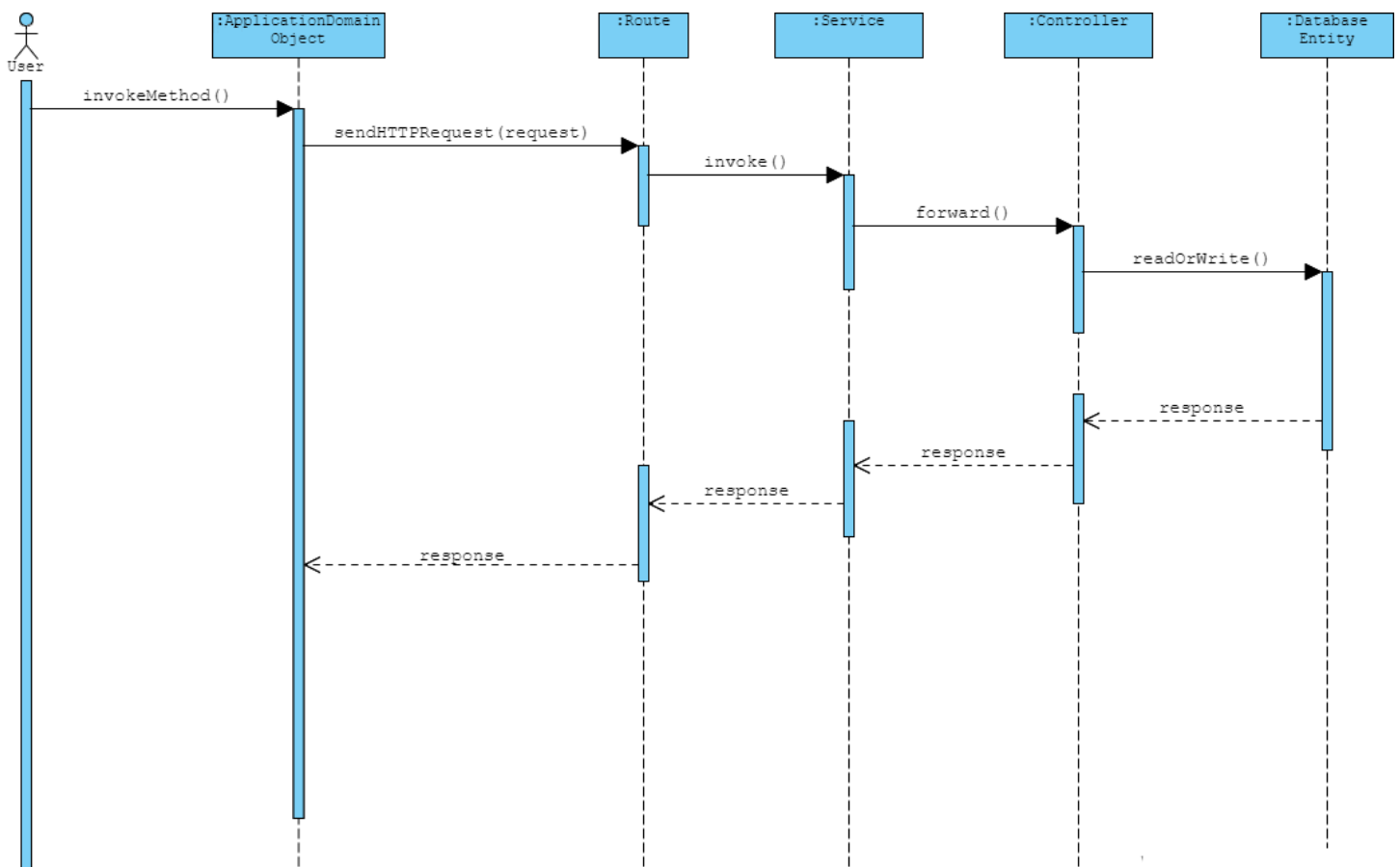
- Role(**RoleID**, RoleName)
- Operation(**OperationID**, OperationName)
- Capability(Role.**RoleID**, Operation.**OperationID**)
- Category(**CategoryID**, CategoryName, Description)
- Account(**AccountID**, Email, Username, Password)
- Node(**NodeID**, FirstName, LastName, BirthDate, DeathDate, BirthPlace, DeathPlace, Sex, Node.FatherID, Node.MotherID)
- User(**UserID**, Residence, AccountID, Node.NodeID)
- Document(**DocumentID**, DocumentPath, RetrievalDate, RetrievalPlace, OriginDate, OriginPlace, DocumentName)
- NodeDocuments(Node.**NodeID**, Document.**DocumentID**)

N.B. Il grassetto indica una chiave primaria ed il corsivo un vincolo di integrità referenziale (i.e. chiave esterna) alla chiave primaria di una data relazione ulteriore.

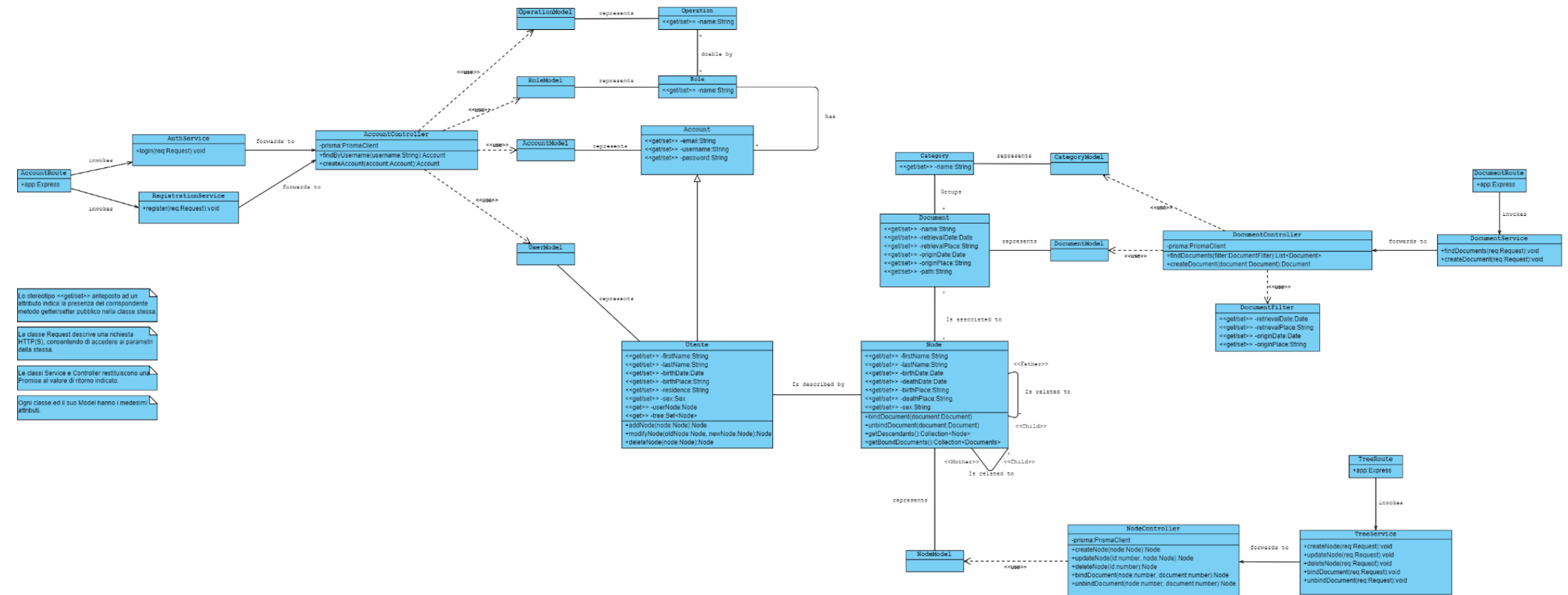
Il modello ad oggetti può dunque essere integrato con le classi del dominio delle soluzioni, nel quale, in particolare:

- Gli oggetti di tipo *Route* hanno il ruolo di associare i metodi dei servizi ad un *endpoint* della REST API.
- Gli oggetti *Service* traducono le richieste ricevute dai *client* in metodi legati agli oggetti del dominio applicativo. Il collateral effect è la restituzione di una risposta HTTP al client.
- I metodi delle classi di tipo *Controller*, ricevono i parametri tradotti dal *Service* e sulla base di questi effettuano operazioni sul database per mezzo di oggetti dell'ORM Prisma. Conseguentemente, il collateral effect sarà l'interazione (lettura o scrittura) con il database.

Una semplice descrizione delle interazioni tra gli oggetti sopraelencati è la seguente: (**SD-OD 1**)



Class diagram del solution domain (back-end): (CD-OD 1)



Lo stereotipo <<getset>> anteposto ad un attributo indica la presenza del corrispondente metodo getter/setter pubblico nella classe stessa.

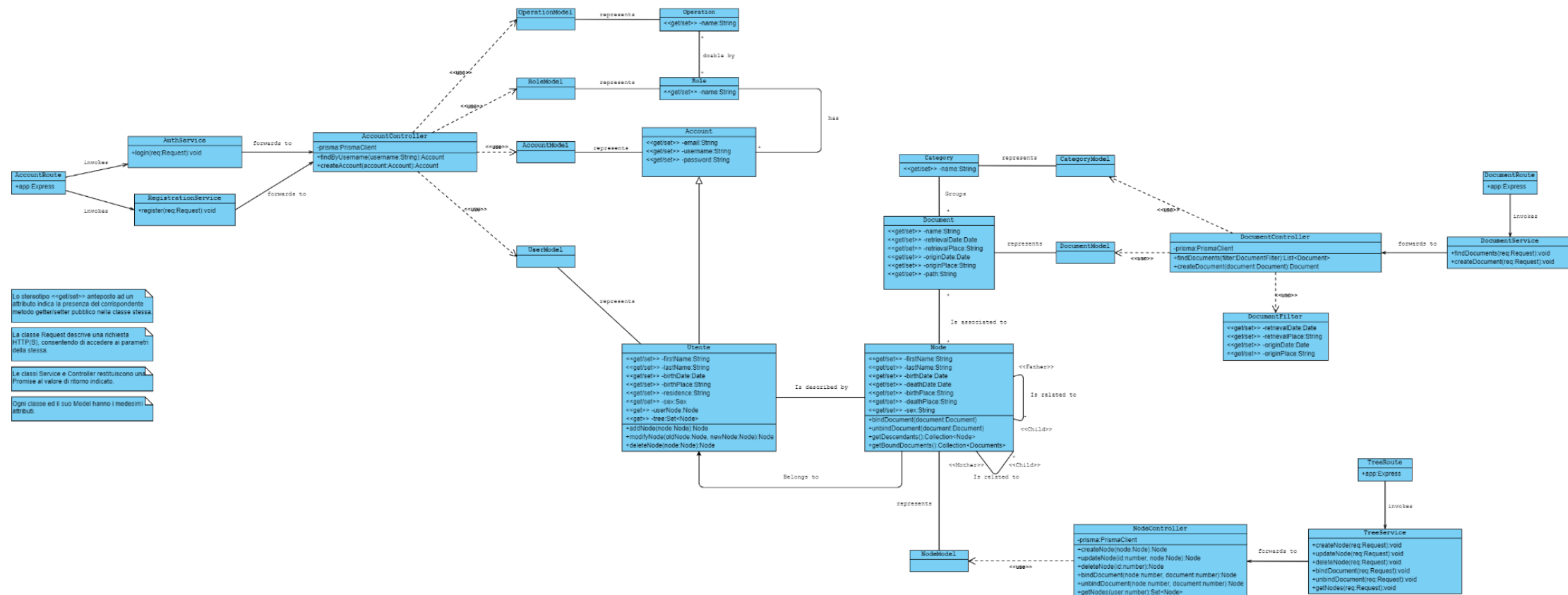
La classe Request descrive una richiesta HTTP(S), consentendo di accedere ai parametri della stessa.

Le classi Service e Controller restituiscono una Promise al valore di ritorno indicato.

Ogni classe ed il suo Model hanno i medesimi attributi.

Un'ottimizzazione è d'obbligo circa i cammini di accesso: al fine di ottenere l'insieme dei nodi nell'albero di un utente a partire da un dato nodo in esso inserito, è utile mantenere il riferimento all'albero di appartenenza in ciascun nodo. E' dunque conservato il riferimento al nodo utente in ciascuna istanza di Node.

## CD-OD 1.1



Seguono i contratti delle classi del solution domain:

**Classe AuthService:**

```
context AuthService::login(req) pre:  
    req.body.username.length() > 0  
    req.body.password.length() > 0
```

```
context AuthService::login(req) post:  
    true
```

**Classe RegistrationService:**

```
context RegistrationService::register(req) pre:  
    req.body.account != null
```

```
context RegistrationService::register(req) post:  
    true
```

**Classe AccountController:**

```
context AccountController inv:  
    prismaClient != null
```

```
context AccountController::findByUsername(username) pre:  
    username != null
```

```
context AccountController::findByUsername(username) post:  
    result.getUsername() = username
```

```

context AccountController::createAccount(account) pre:
    account != null ∧
    account.getUsername().length() > 0 ∧
    account.getPassword().length() > 0 ∧
    account.getEmail().length() > 0 ∧
    account.getUser().getResidence().length() > 0 ∧
    account.getUser().getNode().getFirstName().length() > 0 ∧
    account.getUser().getNode().getLastName().length() > 0 ∧
    account.getUser().getNode().getBirthDate() != null ∧
    account.getUser().getNode().getBirthPlace().length() > 0 ∧
    account.getUser().getNode().getSex() != null

```

```

context AccountController::createAccount(account) post:
    result != null ∧
    result.getRole().getName() = "UtenteBase"

```

### Classe DocumentService:

```

context DocumentService::findDocuments(req) pre:
    true

```

```

context DocumentService::findDocuments(req) post:
    true

```

```

context DocumentService::createDocument(req) pre:
    req.body.document != null

```

```

context DocumentService::createDocument(req) post:
    true

```

**Classe DocumentController:**

**context** DocumentController **inv:**  
 prismaClient != null

**context** DocumentController::findDocuments(filter) **pre:**  
 filter != null

**context** DocumentController::findDocuments(filter) **post:**  
 forAll(d|**result**->asSet())  
 d.getRetrievalDate() != null  $\Rightarrow$   
     d.getRetrievalDate() = filter.getRetrievalDate()  $\wedge$   
 d.getRetrievalPlace() != null  $\Rightarrow$   
     d.getRetrievalPlace() = filter.getRetrievalPlace()  $\wedge$   
 d.getOriginDate() != null  $\Rightarrow$   
     d.getOriginDate() = filter.getOriginDate()  $\wedge$   
 d.getOriginPlace() != null  $\Rightarrow$   
     d.getOriginPlace() = filter.getOriginPlace()

**context** DocumentController::createDocument(document) **pre:**  
 document != null

**context** DocumentController::createDocument(document) **post:**  
**result** != null

**Classe TreeService:**

**context** TreeService::getNodes(req) **pre:**  
 req.params.owner != null

**context** TreeService::getNodes(req) **post:**  
**true**



<b>context</b> TreeService::createNode(req) <b>pre:</b> req.body.node != null
<b>context</b> TreeService::createNode(req) <b>post:</b> true
<b>context</b> TreeService::updateNode(req) <b>pre:</b> req.params.id != null $\wedge$ req.body.node != null
<b>context</b> TreeService::updateNode(req) <b>post:</b> true
<b>context</b> TreeService::deleteNode(req) <b>pre:</b> req.params.id != null
<b>context</b> TreeService::deleteNode(req) <b>post:</b> true
<b>context</b> TreeService::bindDocument(req) <b>pre:</b> req.params.nodeID != null $\wedge$ req.params.documentID != null
<b>context</b> TreeService::bindDocument(req) <b>post:</b> true
<b>context</b> TreeService::unbindDocument(req) <b>pre:</b> req.params.nodeID != null $\wedge$ req.params.documentID != null

```
context TreeService::unbindDocument(req) post:
    true
```

### Classe NodeController:

```
context NodeController inv:
    prismaClient != null
```

```
context NodeController::getNodes(owner) pre:
    owner != null
```

```
context NodeController::getNodes(owner) post:
    forAll(n|result) n.getOwner() = owner
```

```
context NodeController::createNode(node) pre:
    node != null  $\wedge$ 
    node.getOwner() != null
```

```
context NodeController::createNode(node) post:
    result != null
```

```
context NodeController::updateNode(id, node) pre:
    id != null  $\wedge$ 
    node != null
```

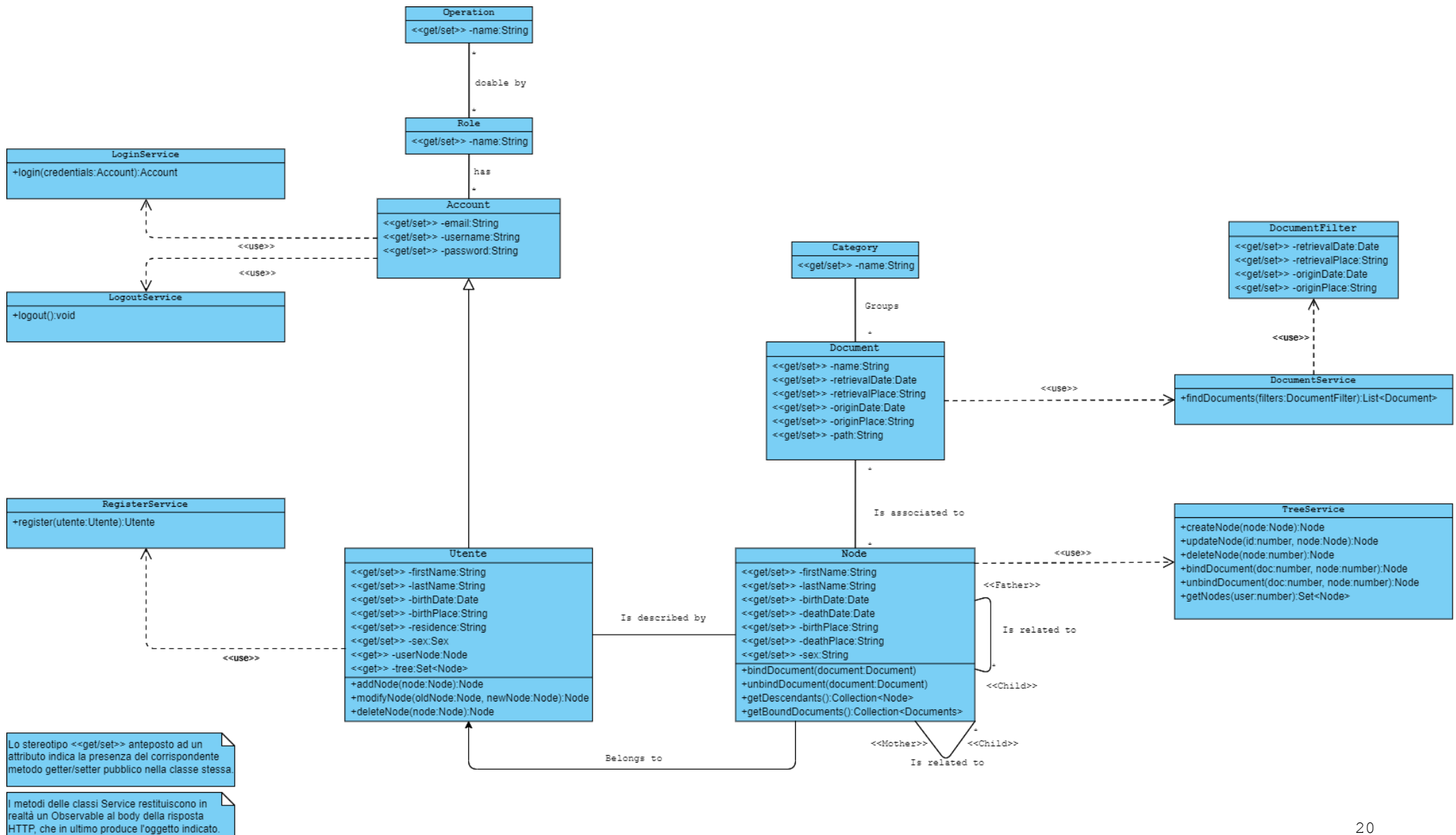
```
context NodeController::updateNode(id, node) post:
    node.getFirstName() != null  $\Rightarrow$ 
        result.getFirstName() = node.getFirstName()  $\wedge$ 
    node.getLastName() != null  $\Rightarrow$ 
        result.getLastName() = node.getLastName()  $\wedge$ 
    node.getBirthDate() != null  $\Rightarrow$ 
```

<pre> <b>result</b>.getBirthDate() = node.getBirthDate() ∧ node.getDeathDate() != null ⇒ <b>result</b>.getDeathDate() = node.getDeathDate() ∧ node.getBirthPlace() != null ⇒ <b>result</b>.getBirthPlace() = node.getBirthPlace() ∧ node.getDeathPlace() != null ⇒ <b>result</b>.getDeathPlace() = node.getDeathPlace() ∧ node.getSex() != null ⇒ <b>result</b>.getSex() = node.getSex() </pre>
<pre> <b>context</b> NodeController::deleteNode(id) <b>pre:</b>   id != null </pre>
<pre> <b>context</b> NodeController::deleteNode(id) <b>post:</b>   <b>result</b> != null </pre>
<pre> <b>context</b> NodeController::bindDocument(nodeID, documentID) <b>pre:</b>   nodeID != null ∧   documentID != null </pre>
<pre> <b>context</b> NodeController::bindDocument(nodeID, documentID) <b>post:</b>   <b>result</b>.documents-&gt;includes(documentID) </pre>
<pre> <b>context</b> NodeController::unbindDocument(nodeID, documentID) <b>pre:</b>   nodeID != null ∧   documentID != null </pre>
<pre> <b>context</b> NodeController::unbindDocument(nodeID, documentID) <b>post:</b>   !<b>result</b>.documents-&gt;includes(documentID) </pre>

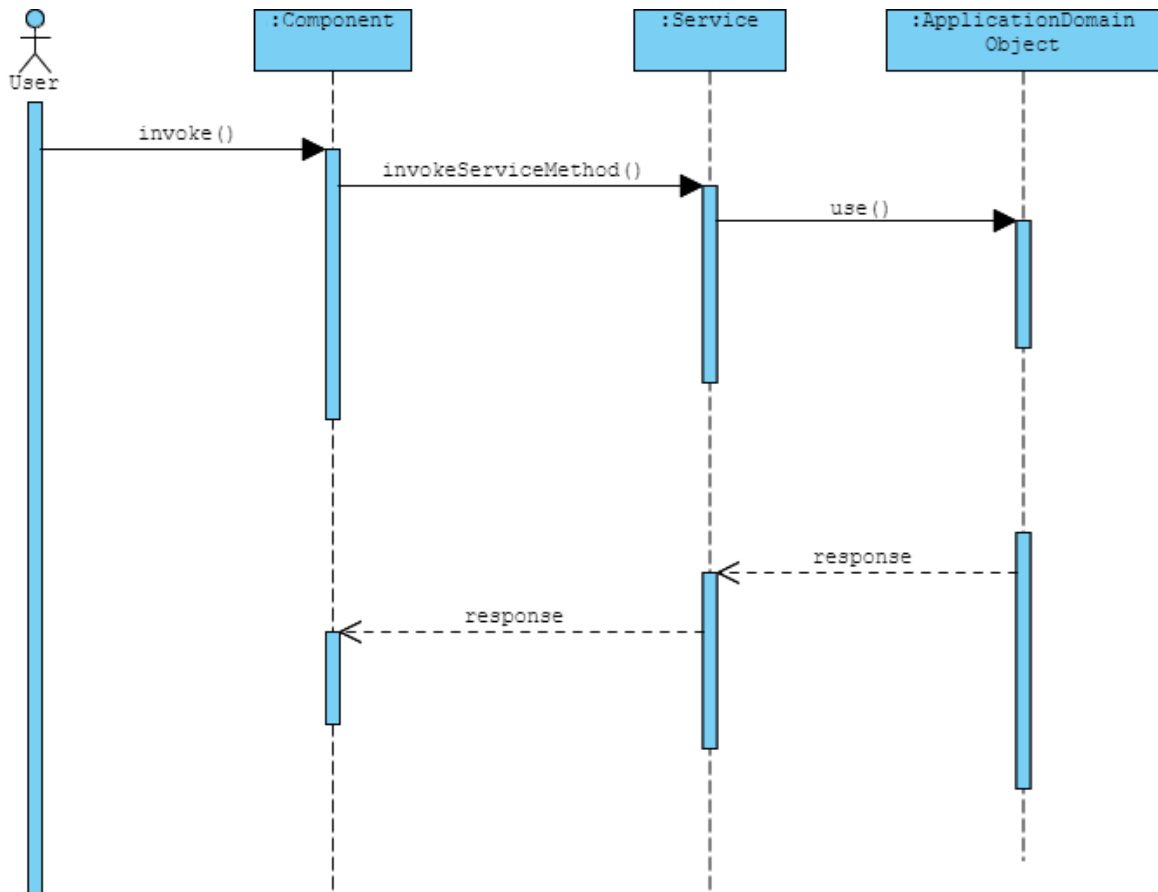
Le postcondizioni saranno naturalmente verificate mediante l'esecuzione dei test.

Il class diagram client-side "decorato" con le classi del solution domain è il seguente:

## CD-OD 2



In un certo senso dal punto di vista del client, le classi :Component Angular giocano il ruolo di Boundary (per la parte interattiva) e Control (per la parte di controllo del flusso). Le classi :Service danno invece un senso ai più astratti Manager. Si osservi, per chiarezza, il sequence diagram seguente per comprendere la prosecuzione del flusso dei dati:

**SD-OD 2**

Contratti per le classi Service client-side

**Classe AuthService:**

<b>context</b> AuthService::login(credentials) <b>pre:</b> credentials.getUsername() != null credentials.getPassword() != null
<b>context</b> AuthService::login(account) <b>post:</b> <b>result</b> != null
<b>context</b> AuthService::logout() <b>pre:</b> <b>true</b>
<b>context</b> AuthService::logout() <b>post:</b> <b>true</b>

**Classe RegistrationService:**

<b>context</b> RegistrationService::register(utente) <b>pre:</b> utente.getFirstName() != null ∧ utente.getLastName() != null ∧ utente.getBirthDate() != null ∧ utente.getBirthPlace() != null ∧ utente.getResidence() != null ∧ utente.getSex() != null ∧ utente.getEmail() != null ∧ utente.getUsername() != null ∧ utente.getPassword() != null
<b>context</b> RegistrationService::register(utente) <b>post:</b> <b>result</b> = utente

**Classe TreeService:**

```
context TreeService::getNodes(user) pre:
    user != null
```

```
context TreeService::getNodes(user) post:
    forAll(n|result) n.getOwner() = user
```

```
context TreeService::createNode(node) pre:
    node != null
```

```
context TreeService::createNode(node) post:
    result = node
```

```
context TreeService::updateNode(id, node) pre:
    id != null  $\wedge$ 
    node != null
```

```
context TreeService::updateNode(id, node) post:
    node.getFirstName() != null  $\Rightarrow$ 
        result.getFirstName() = node.getFirstName()  $\wedge$ 
    node.getLastName() != null  $\Rightarrow$ 
        result.getLastName() = node.getLastName()  $\wedge$ 
    node.getBirthDate() != null  $\Rightarrow$ 
        result.getBirthDate() = node.getBirthDate()  $\wedge$ 
    node.getDeathDate() != null  $\Rightarrow$ 
        result.getDeathDate() = node.getDeathDate()  $\wedge$ 
    node.getBirthPlace() != null  $\Rightarrow$ 
        result.getBirthPlace() = node.getBirthPlace()  $\wedge$ 
    node.getDeathPlace() != null  $\Rightarrow$ 
        result.getDeathPlace() = node.getDeathPlace()  $\wedge$ 
    node.getSex() != null  $\Rightarrow$ 
        result.getSex() = node.getSex()
```

<b>context</b> TreeService::deleteNode(node) <b>pre:</b> node != null
<b>context</b> TreeService::deleteNode(node) <b>post:</b> <b>true</b>
<b>context</b> TreeService::bindDocument(doc, node) <b>pre:</b> doc != null $\wedge$ node != null
<b>context</b> TreeService::bindDocument(doc, node) <b>post:</b> node.getBoundDocuments()->includes(doc)
<b>context</b> TreeService::unbindDocument(doc, node) <b>pre:</b> doc != null $\wedge$ node != null
<b>context</b> TreeService::unbindDocument(doc, node) <b>post:</b> !node.getBoundDocuments()->includes(doc)

### Classe DocumentService:

<b>context</b> DocumentService::findDocuments(filter) <b>pre:</b> <b>filter</b> != null
<b>context</b> DocumentService::findDocuments(req) <b>post:</b> forall(d  <b>result</b> ->asSet()) d.getRetrievalDate() != null $\Rightarrow$ d.getRetrievalDate() = filter.getRetrievalDate() $\wedge$ d.getRetrievalPlace() != null $\Rightarrow$ d.getRetrievalPlace() = filter.getRetrievalPlace() $\wedge$



```
d.getOriginDate() != null ⇒  
    d.getOriginDate() = filter.getOriginDate() ∧  
d.getOriginPlace() != null ⇒  
    d.getOriginPlace() = filter.getOriginPlace()
```

Si osservi che le classi `Service` individuate client-side e quelle osservate lato back-end, con le quali comunicano, sono omonime, ai fini dell'aumento della leggibilità.

---

### **Note sui Design Pattern utilizzati**

Alcuni elementi grafici (client-side) necessitano di essere aggiornati in risposta alla variazione di alcuni dati, ad esempio l'albero genealogico, da aggiornare in risposta alle operazioni CUD su di esso. Conseguentemente, è possibile utilizzare un *observer pattern* per gestire tali eventualità. Angular consente un'implementazione estremamente agevole di tale pattern mediante la definizione del tipo `Observable`.

---

### **Note sul mapping dei modelli e delle scelte sul codice**

In precedenza sono state inevitabilmente introdotte alcune ottimizzazioni e ristrutturazioni, implicitamente legate al mapping dei modelli sul codice. Linee guida e strategie ulteriori sono descritte nella sezione presente.

Al fine di evitare la ripetizione di letture da database e file system dei documenti, e come esplicitato nel paragrafo 2.1 del RMD caching è gestito:

- **Server side:** mediante API-cache, i documenti recuperati dal database sono mantenuti in una cache per 1 ora. All'invocazione dell'operazione `findDocuments()` del servizio `DocumentService`, `DocumentRoute` interpone un metodo che verifica la presenza dei documenti nella cache prima dell'invocazione stessa.
- **Client side:** il modulo `Node.js NgHttpCaching` consente di porre in cache le richieste HTTP: nel caso esaminato ciò

sarà effettuato relativamente alle richieste di ricerca documenti.

Le associazioni riportate nel Class Diagram saranno mappate traducendo:

- Le associazioni 1-1 come variabili di istanza riferite all'istanza associata;
- Le associazioni 1-N imponendo una collezione Set (o List, se ordinata).

I messaggi di errore saranno accompagnati dagli opportuni codici di errore previsti dal protocollo HTTP (e.g. 401 - Unauthorized).