Exigram

Object Design Document

Membri del team

Iannaccone Davide 05121 05292

Esposito Domenico 05121 05478

Data	Cambiamenti	Autori
12/Gennaio/2020	Creazione documento	Iannaccone Davide Esposito Domenico
14/Gennaio/2020	Modifica, correzione e aggiunta al documento	Iannaccone Davide
12/Febbraio/2020	Modifica e aggiunta al documento	Iannaccone Davide
17/Febbraio/2020	Rifinita formattazione al documento	Esposito Domenico

Sommario

1 Introduzione

1.1 Object Design Trade-off

Ci apprestiamo a definire gli aspetti implementativi del nostro sistema, aspetti che nei documenti precedenti (e.g. “RAD”, “SDD” etc.) sono stati tralasciati per definire in modo chiaro gli obiettivi del sistema. Con il seguente documento puntiamo a creare un modello capace di descrivere in modo preciso tutte le funzionalità individuate nei modelli precedenti.

Nel particolare, definiremo le interfacce delle classi, le operazioni, i tipi, gli argomenti e le “signature” dei sottosistemi definiti nel System Design Document.

Sono necessari però definire una serie di trade-off e di linee guida per poter lavorare in modo ordinato alla “codebase”.

Interfaccia vs Usabilità:

L’interfaccia fa dell’usabilità il suo punto di forza. Fin dal primo momento la comodità di utilizzo è stata messa in primo piano durante il design, scelta che si rispecchia fin dai primi mock -up.

Sicurezza vs Efficienza:

La sicurezza è la principale preoccupazione di ogni piattaforma online che si rispetti, ma il poco tempo a disposizione non ci permette di rendere “bullet proof” il sistema. Metteremo comunque il minimo di sicurezza possibile nel progetto, assicurandoci che il login/registrazione e tutte le altre interazioni con il database siano sicure e quanto più efficienti possibili.

1.2 Linee guida per la Documentazione di Interfacce

Per creare un progetto chiaro, pulito e univocamente compreso, gli sviluppatori dovranno sottostare ad alcune linee guida per la stesura del codice:

Code Style: È buona norma rispettare gli standard di buona programmazione Java, circa anche la formattazione del codice, nello specifico:

- Tutto il codice sorgente deve essere indentato con dei Tabs.
- Il codice deve essere formattato secondo lo stile vigente di Visual Studio Code (si consiglia vivamente di utilizzare quest'ultimo).
- Viene utilizzata una spaziatura strutturata per migliorare la leggibilità del codice.

Esempio:

```
public void method() {
    for(Object x: ObjectList) {
        //Code
    }
}
```

Naming Convention:

È buona norma utilizzare nomi che siano:

- Descrittivi
- Pronunciabili
- Di uso comune
- Non abbreviati (se non per variabili temporali)
- Utilizzando solo caratteri consentiti (A - Z, a – z, 0 - 9)

Variabili:

È buona norma rispettare gli standard di Java circa la nomenclatura delle variabili, nello specifico:

- Devono iniziare per la lettera minuscola ed ogni parole seguenti devono iniziare con una lettera maiuscola. Esempio: nomeDiVariabile.
- Nel caso in cui la variabile in questione è una costante, il nome deve essere interamente in maiuscola ed eventuali spazi devono essere sostituiti con l'underscore: '_'. Esempio: NOME_DI_COSTANTE.

Metodi:

E' buona norma rispettare gli standard di Java, circa la nomenclatura dei metodi, nello specifico:

- Devono iniziare per lettera minuscola e le parole seguenti devono iniziare con una lettera maiuscola. Esempio: nomeDiMetodo().
- Eventuali metodi di accesso e modifica delle variabili di istanza di una classe devono essere del tipo: getNomeVariabile() e setNomeVariabile().
- I commenti devono essere raggruppati secondo la funzionalità che vanno a descrivere, devono essere situati prima della dichiarazione del metodo. Utilizzando la JavaDoc si può avere una documentazione completa ed uniforme.

Classi:

È buona norma rispettare gli standard Java circa la nomenclatura delle classi, nello specifico:

- Devono iniziare per lettera maiuscola e le parole seguenti devono iniziare con una lettera maiuscola. Esempio: NomeDellaClasse.
- Devono avere un nome autodescrittivo.

1.3 Definizioni, Acronimi, Abbreviazioni e Riferimenti

Acronimi:

- RAD: Requirements Analysis Document
- SDD: System Design Document
- ODD: Object Design Document

Abbreviazioni:

- DB: Database
- API: Application Programming Interface

Riferimenti:

- Slide fornite dal Professore Andrea De Lucia.
- Libro di testo: Bruegge, A.H. Dutoit, Object Oriented Software Engineering.

2 Packages

Il nostro sistema è suddiviso in tre livelli:

Interface layer:

> Rappresenta il layer che l'utente visualizza e quello con il quale interagisce. Raccoglie i dati in input e li inoltra all'Application Logic Layer, in seguito mostra in output il risultato dell'azione svolta.

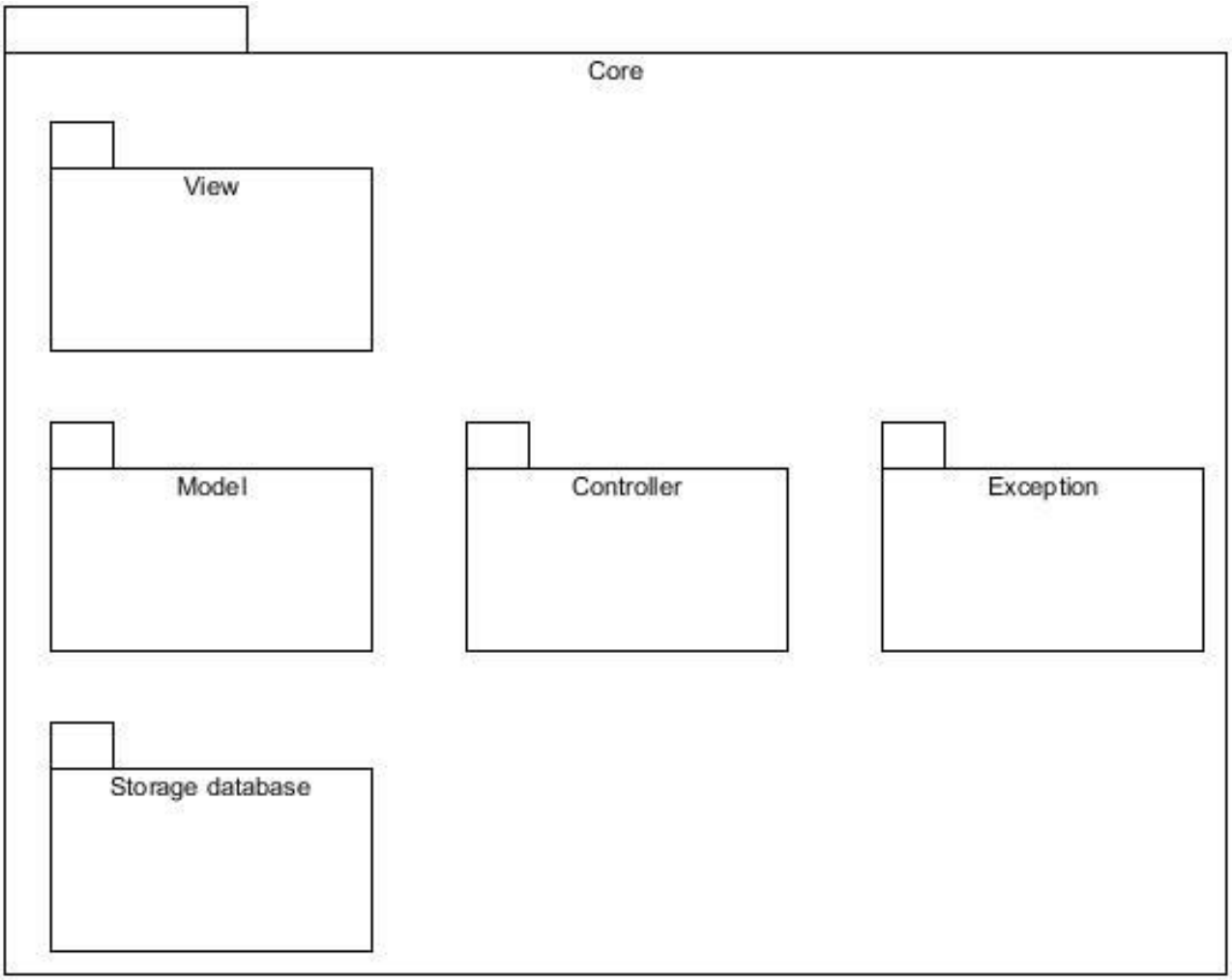
Application Logic layer:

> Elabora i dati che arrivano dal client e li restituisce all'Interface Layer, spesso utilizzando i dati persistenti gestiti dallo Storage layer. Svolge varie funzioni raggruppate nelle seguenti gestioni:
>
> 1 Gestione Post
>
> 2 Gestione Utente
>
> 3 Gestione Ricerca
>
> 4 Gestione Autenticazione
>
> 5 Gestione Amministratore

Storage layer:

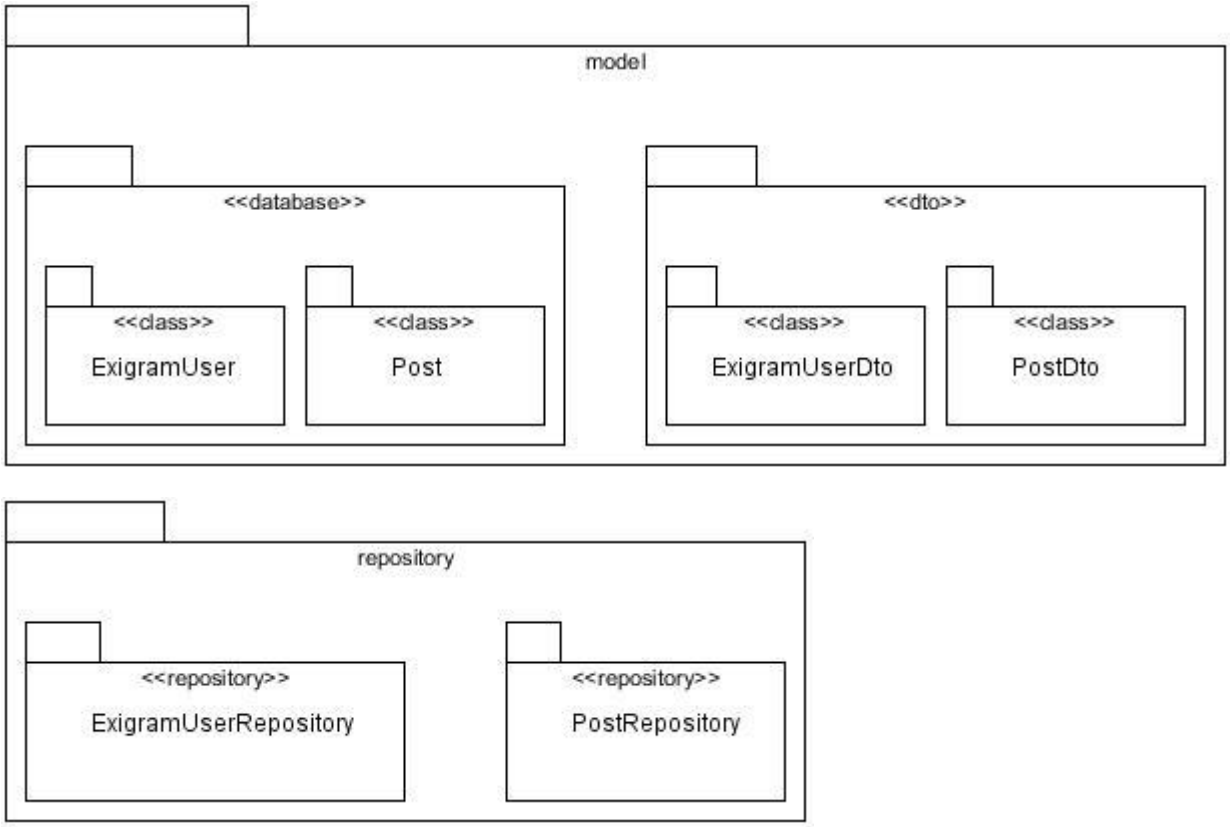
> Ha il compito di memorizzare i dati in maniera persistente, utilizzando un DBMS, riceve richieste dall'Application Logic layer e restituisci i dati del DBMS richiesti.

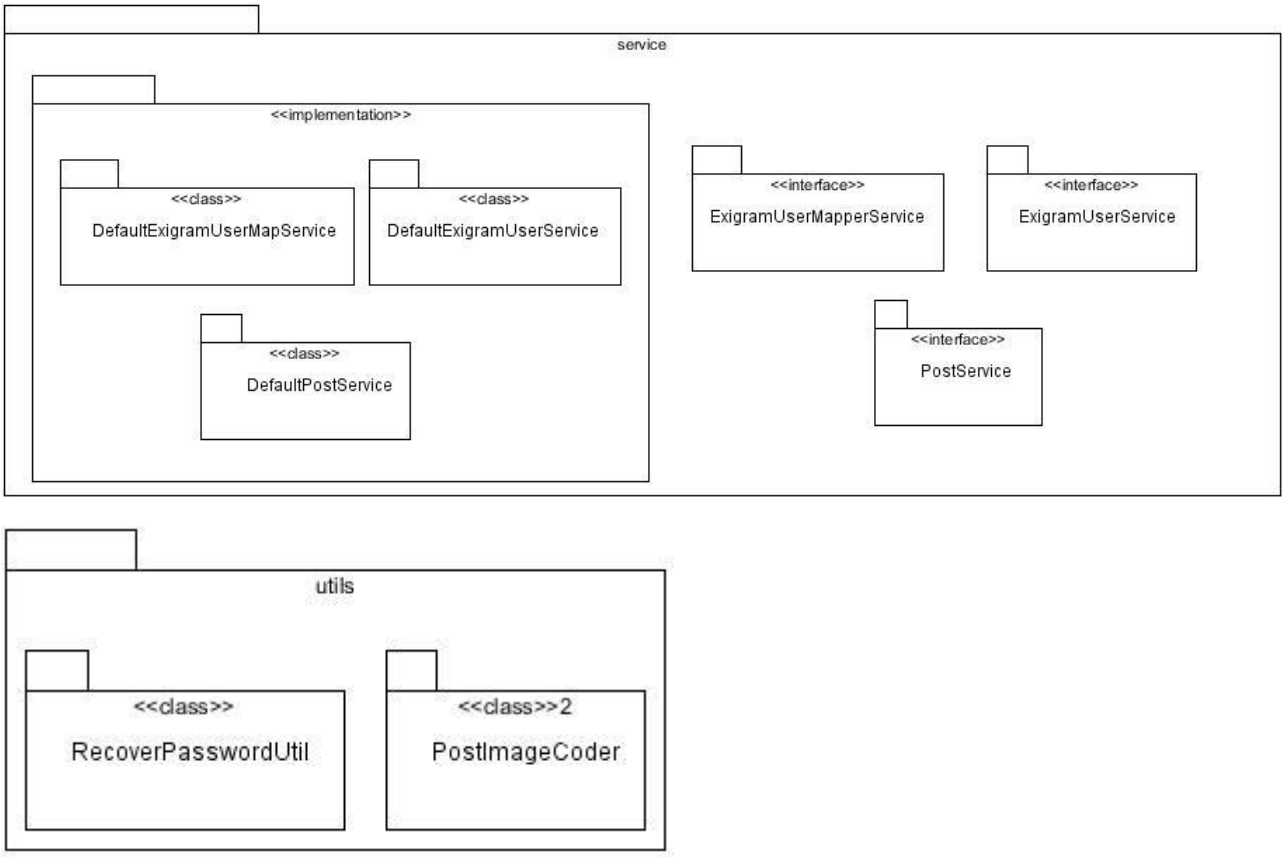
2.1 Package Core



2.1.1 Model Package

Package	Classe	Descrizione
database	ExigramUser.java	Model che rappresenta l'entity dell'utente di Exigram all'interno del database.
dto	ExigramUserDto.java	Data Transfer Object dell'utente Exigram utilizzato per trasportare dati dal back-end al front-end, omettendo alcuni dati sensibili per motivi di sicurezza.
database	Post.java	Model che rappresenta l'entity dei post degli utenti Exigram all'interno del database.
dto	PostDto.java	Data Transfer Object del post utilizzato per trasportare dati dal back-end al front-end.

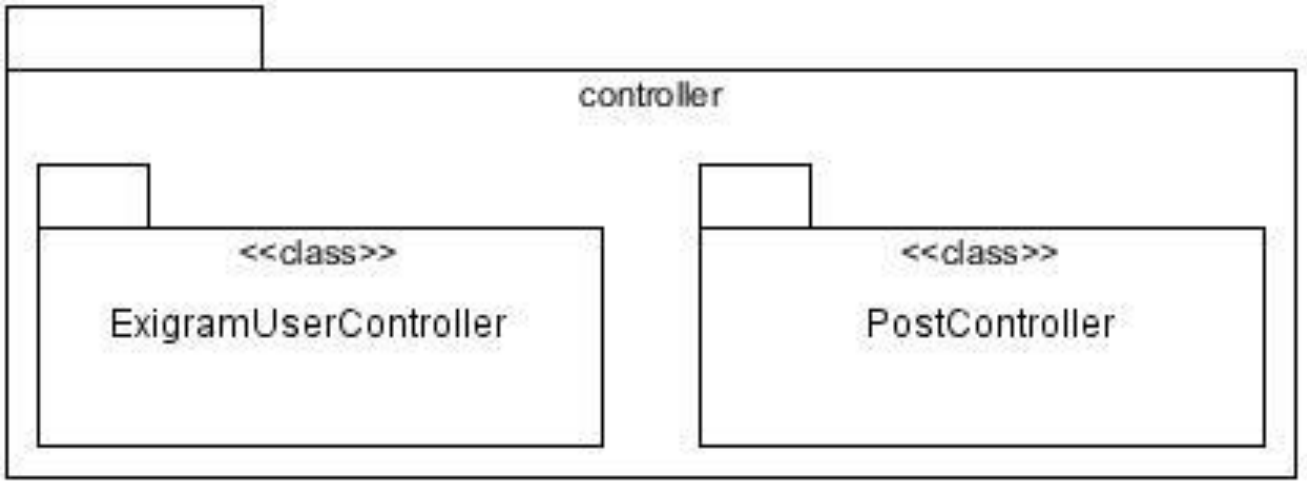



service
<<implementation>>
<<class>>
DefaultExigramUserMapService
<<class>>
DefaultExigramUserService
<<class>>
DefaultPostService
<<interface>>
ExigramUserMapperService
<<interface>>
ExigramUserService
<<interface>>
PostService

utils
<<class>>
RecoverPasswordUtil
<<class>>2
PostImageCoder

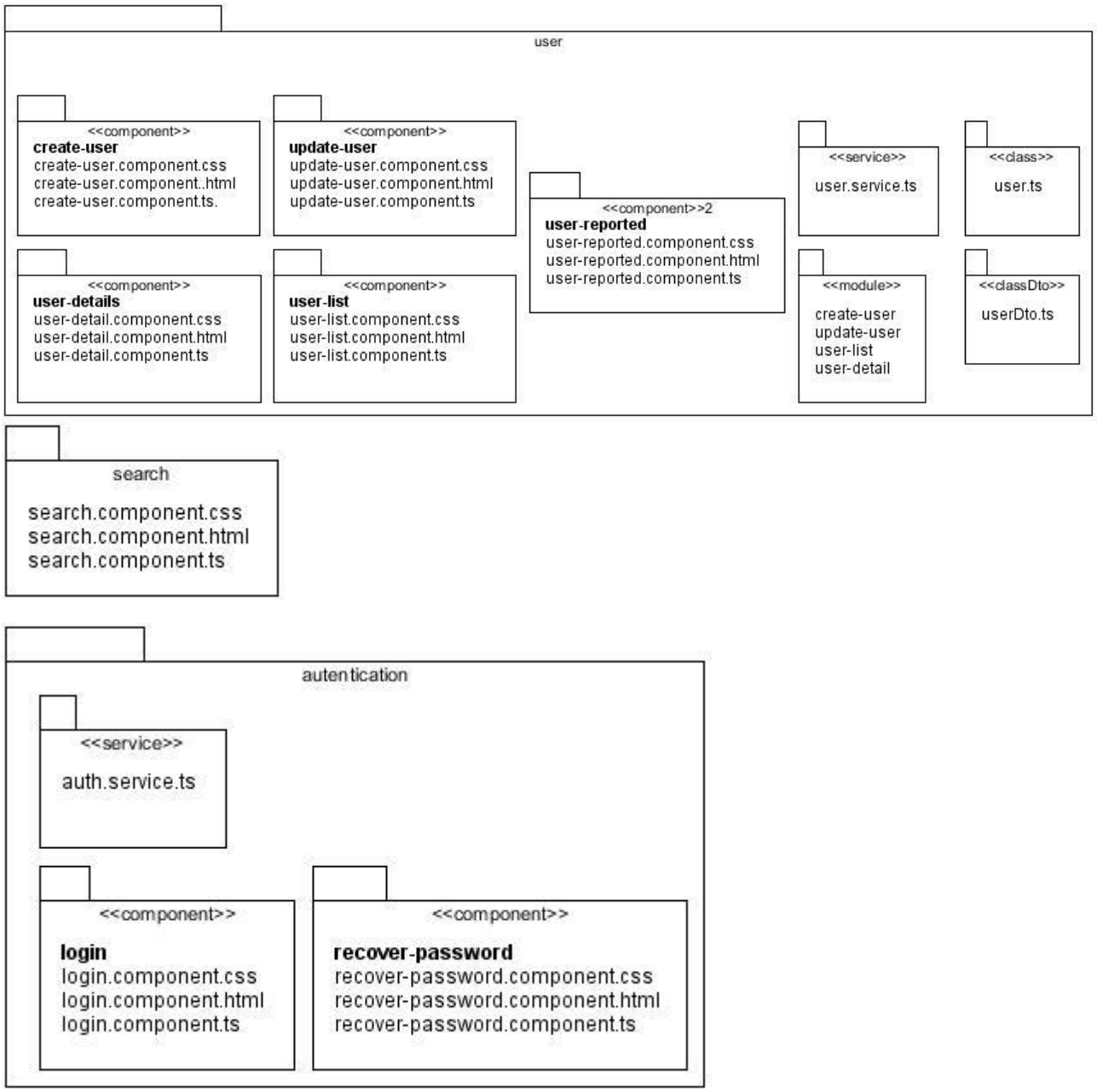
2.1.2 Controller Package

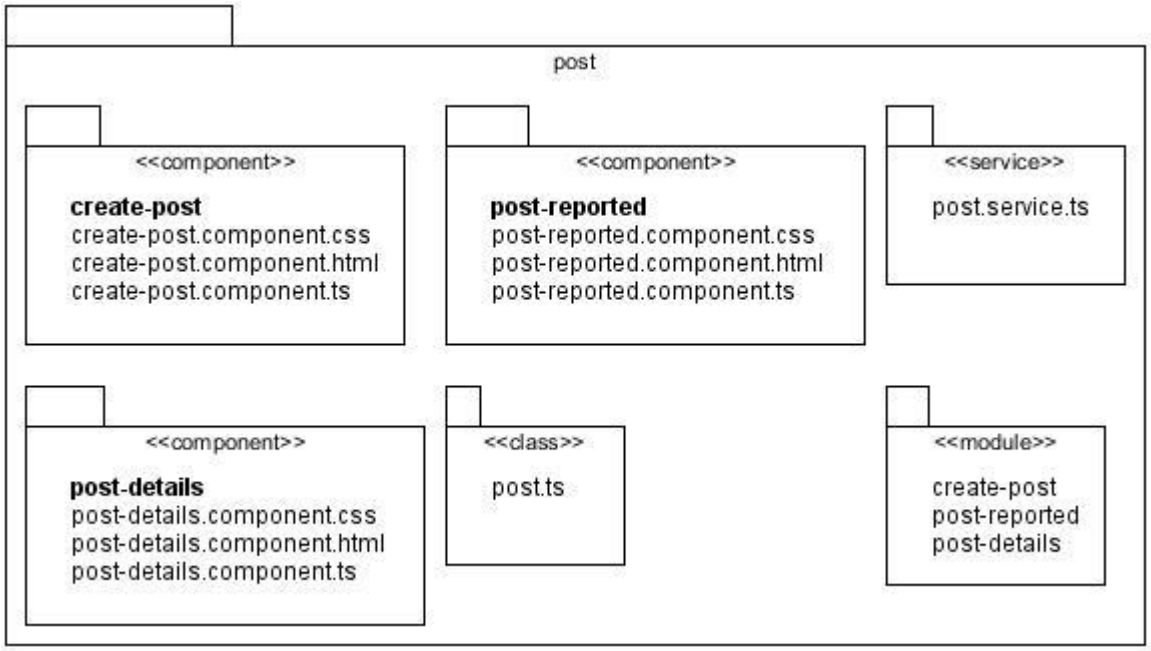
Classe	Descrizione
ExigramUserController.java	Intercetta le richieste che arrivano quando un utente Exigram viene chiamato, inviando i dati al Model corrispondente (ExigramUser.java) e ottiene i dati processati dal Model per reindirizzarli alla View.
PostController.java	Intercetta le richieste che arrivano quando un utente Exigram esegue un'operazione riguardante al Model corrispondente (Post.java) e ottiene i dati processati dal Model per reindirizzarli alla View.

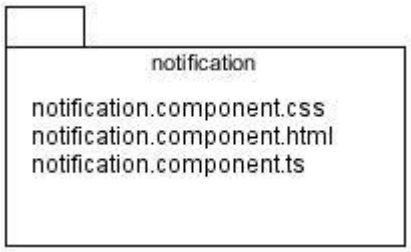


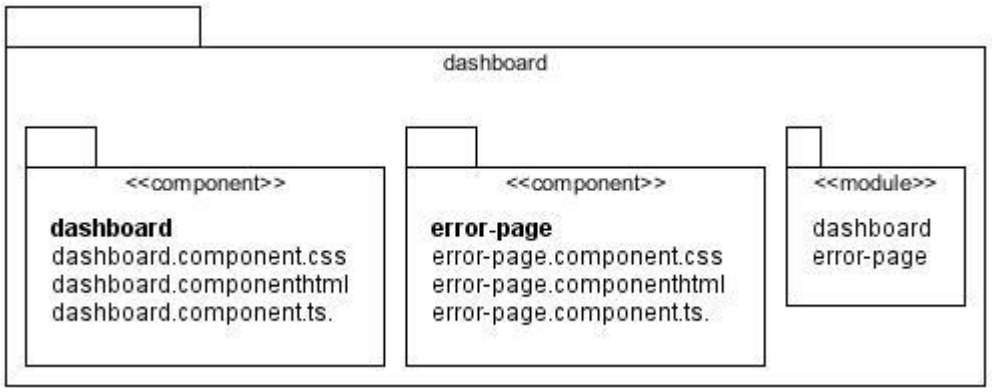
2.1.3 View Package

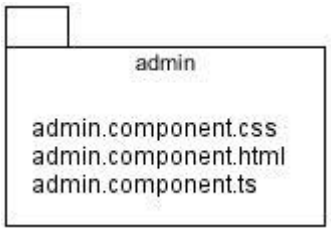
Classe	Descrizione
ExigramUserController.java	Intercetta le richieste che arrivano quando un utente Exigram viene chiamato, inviando i dati al Model corrispondente (ExigramUser.java) e ottiene i dati processati dal Model per reindirizzarli alla View.
PostController.java	Intercetta le richieste che arrivano quando un utente Exigram esegue un'operazione riguardante al Model corrispondente (Post.java) e ottiene i dati processati dal Model per reindirizzarli alla View.




post
<<component>>
create-post
create-post.component.css
create-post.component.html
create-post.component.ts
<<component>>
post-reported
post-reported.component.css
post-reported.component.html
post-reported.component.ts
<<service>>
post.service.ts
<<component>>
post-details
post-details.component.css
post-details.component.html
post-details.component.ts
<<class>>
post.ts
<<module>>
create-post
post-reported
post-details


notification
notification.component.css
notification.component.html
notification.component.ts


dashboard
<<component>>
dashboard
dashboard.component.css
dashboard.componenthtml
dashboard.component.ts.
<<component>>
error-page
error-page.component.css
error-page.componenthtml
error-page.component.ts.
<<module>>
dashboard
error-page


admin
admin.component.css
admin.component.html
admin.component.ts

3 Interfaccia Delle Classi

L’interfaccia delle classi, ogni metodo ed ogni classe è opportunamente descritto/a nella documentazione “JavaDoc” allegata.

4 Design Patterns

Per sviluppare un sistema coeso, ben definito e facilmente modificabile è stata presa come scelta uno specifico design pattern in cui sono presenti oggetti che “osservano” lo stato di un oggetto e vengono notificati nel preciso istante in cui viene cambiato lo stato dell’oggetto.

4.1 Observer Pattern

La possibilità di avere oggetti che “osservano” fornisce una molteplicità di funzionalità:

- Gli osservatori (Observer) devono potersi registrare.
- Devono poter notificare: gli Observer devono fornire un’interfaccia standard per la notifica.
- Chi evoca setState(), utilizzato per impostare lo stato di un Observer è un altro Observer.
- Il Client che ha terminato una sequenza di modifiche e ogni metodo dell’oggetto osservato che modifica lo stato invoca notify().
- Nel caso in cui un osservatore osserva più oggetti, riconosce con precisione chi ha variato lo stato attraverso il parametro di update(), come ad esempio this.