Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

DYANOTE: HYPERTEXT-BASED NOTE-TAKING

Tesi di Laurea in Informatica

Relatore:
Chiar.mo Prof.
Davide Rossi

Presentata da:
Matteo Nardi

Sessione III
2012-13

Indice

Capitolo 1

Design Pattern e Django

1.1 Introduzione

1.1.1 Design Pattern

I design pattern sono soluzioni generiche a problemi ricorrenti nel design del sofware. Dobbiamo la loro introduzione nel mondo dell'informatica a Gamma, Helm, Johnson e Vlissides che, con il loro libro del 1995[1], hanno classificato ed organizzato le soluzioni ad alcuni dei problemi più diffusi. Nell'ultimo ventennio si è cercato di ampliare il catalogo dei 23 pattern, ma i principali sono rimasti quelli descritti nel libro del '95.

Un pattern è composto da diversi elementi:

Nome Serve per identificare un pattern e facilitare la comunicazione fra gli sviluppatori.

Problema Descrive il problema di design che il pattern cerca di risolvere. Indica il contesto in cui utilizzarlo ed elenca una serie di condizioni ed euristiche che aiutano a capire quando abbia senso applicarlo.

Soluzione Descrive gli elementi che formano il design: le loro relazioni, responsabilità e collaborazioni. La soluzione è come un template che può essere applicato in diversi contensti.

[1] Design Patterns: Elements of Reusable Object-Oriented Software.

Conseguenze Descrive i vantaggi e gli svantaggi dell'utilizzo del pattern. Ogni design comporta più flessibilità sotto certi punti di vista e maggiore rigidità sotto altri. Un pattern non va mai applicato inconsciamente e bisogna sempre confrontarlo con le sue alternative.

I pattern sono classificati in base al loro livello (se operano sulle classi o sugli oggetti) ed al loro scopo:

Creazionali Risolvono un problema legato alla creazione degli oggetti.

Strutturali Riguardano la composizione e la struttura da dare alle classi.

Comportamentali Indicano come le classi e gli oggetti devono interagire e distribuirsi le responsabilità.

Se usati correttemente i pattern aiutano a progettare sistemi più flessibili. Sono un utile strumento da applicare insieme alle guideline di design SOLID[2] e GRASP[3].

1.1.2 Django

Django[4] è un web framework che permette di sviluppare velocemente applicazioni lato server in Python[5]. Questo framework riduce molto la quantità di codice che deve scrivere un programmatore, mettendo a sua disposizione:

- Un ORM per i principali database relazionali.
- Un'interfaccia di amministrazione creata in automatico.
- Un flessibile meccanismo di smistamento delle richieste in base alle URL.
- Un linguaggio di template.

[2]SOLID: Single responsibility, Open-closed, Liskov substitution, Interface segregation e Dependency inversion

[3]GRASP: General Responsibility Assignment Software Patterns

[4]https://www.djangoproject.com/

[5]http://www.python.org/

- Gestione di utenti, autenticazione e permessi.
- Delle utility da linea di comando per creare e gestire velocemente l'ambiente di sviluppo.
- Supporto a internazionalizzazione, cache, RSS, gestione file statici e molto altro.

1.2 Pattern usati in Django

Come lavoro di tesi ho cercato e trovato dentro al codice di Django [6] alcuni dei più diffusi design pattern.

1.2.1 Abstract Factory - Supporto ai database

L'Object Relational Mapper di Django offre un'astrazione di alto livello su tutti i maggiori database relazionali. Nonostante in Python le librerie per l'accesso ai database offrano un'interfaccia più o meno simile[7], ogni database ha bisogno di trattamenti speciali che rendono l'ORM qualcosa di non banale.

In *db/backends* vengono definite le classi astratte su cui si baserà il codice utente ed il resto di Django. Nelle sottocartelle (*db/backends/mysql*, *db/backends/oracle*, ecc.) sono invece presenti le implementazioni concrete.

db/backends/__init__.py definisce la classe **BaseDatabaseWrapper**, che permette la gestione di un generico database. Fra le altre cose, questa classe offre un'interfaccia per creare cursori e connessioni tramite le funzioni *create_cursor* e *get_new_connection*. Si tratta di due **Factory method** (metodi astratti che creano risorse astratte) che i vari backend andranno poi ad implementare, creando oggetti concreti. Poiché questi oggetti concreti vanno utilizzati assieme ad altri oggetti della stessa tipologia, possiamo considerare *BaseDatabaseWrapper* un'**Abstract factory**: un pattern creazionale che permette di creare oggetti concreti appartenenti alla stessa famiglia.

[6] https://github.com/django/django/tree/master/django
[7] http://www.python.org/dev/peps/pep-0249/

1.2.2 Composite - Configurazione delle URL

Un'utile componente di Django è l'URL dispatcher, che permette di determinare, per ogni URL, la view corrispondente da invocare.

Ogni appplicazione Django può creare dei mappaggi, che possono poi essere inclusi nelle altre applicazioni.

```
# API endpoints
urlpatterns = format_suffix_patterns(patterns('api.views',
    url(r'^$', 'root_view'),
    url(r'^register_user/$', 'register_user_view'),
    #...
))

urlpatterns += patterns('',
    url(r'^oauth2/', include('provider.oauth2.urls',
                             namespace='oauth2')),
)
```

L'implementazione si trova in *core/urlresolvers.py* e si nota come sia stato utilizzato un composite. **RegexURLPattern** rappresenta una singola associazione Url-view (Rappresenta una foglia del composite), mentre **RegexURLResolver** rappresenta una collezione di associazioni (Si tratta del composite vero e proprio). *RegexURLResolver* contiene una lista di altri *RegexURLResolver* e *RegexURLPattern*. Entrambe le classi hanno la funzione *resolve*, che restituisce un match.

In *conf/urls/__init__.py* si trovano le funzione di utility che il codice utente userà per creare il composite.

Composite è un pattern strutturale che permette la creazione di gerarchie composte da oggetti di tipo diverso. Attraverso l'implementazione di un'interfaccia comune, ogni

elemento di questa gerarchia può essere trattato allo stesso modo, sia che si tratti di una foglia, della radice o altro. Nel nostro contesto questo pattern è particolarmente adatto perché permette sia al codice utente di includere le viste di altre applicazioni, sia a Django di trattare questa gararchia in maniera trasparente.

1.2.3 Decorator - Aggiunta di funzionalità alle view

In Django le view sono dei metodi che rispondono a richieste HTTP.

```
@require_http_methods(["GET", "POST"])
def hello(request, format=None):
    return HttpResponse("Hello world.")
```

Django mette a disposizione dei decorator (*views/decorators*), per esempio per filtrare le richieste su particolari metodi HTTP (GET, POST, ecc.), utilizzare la compressione Gzip, eseguire la view solo su certe condizioni degli Headers ecc.

Creare decorator in Python è molto semplice perché il linguaggio stesso offre una sintassi per farlo.[8]

```
def wrapper(func):
    def mult():
        return func() * 2
    return mult

@wrapper
def six():
    return 3

twelve = wrapper(six)
twelve() # 12
```

[8] https://wiki.python.org/moin/PythonDecorators

Decorator è un pattern strutturale che permette di aggiungere dinamicamente delle responsabilità ad un oggetto. Nel nostro contesto ha un duplice vantaggio, permettendo sia il riuso del codice per certe funzionalità comuni, sia di estendere una view senza bisogno di modificarla direttamente.

1.2.4 Facade

Quasi tutte le componenti di Django utilizzano il pattern Facade per nascondere le complessità al loro interno ed offrire un'interfaccia semplificata per il codice utente.

Qui di seguito inserisco un esempio preso da *contrib/admin/__init__.py*

```
# Imports...

__all__ = [
    "register", "ACTION_CHECKBOX_NAME", "ModelAdmin",
    "HORIZONTAL", "VERTICAL", "StackedInline",
    "TabularInline", "AdminSite", "site", "ListFilter",
    "SimpleListFilter", "FieldListFilter", "autodiscover",
    "RelatedFieldListFilter", "ChoicesFieldListFilter",
    "AllValuesFieldListFilter", "BooleanFieldListFilter",
]

# Un paio di metodi e costanti extra.
```

Se nell'init di un modulo Python è presente l'elemento **__all__**, questo viene usato per decidere quali simboli esportare quando viene eseguito un import come

import django.contrib.admin

Facade è un pattern strutturale che consiste nell'offrire un'interfaccia di alto livello che nasconde i dettagli di un sottosistema. Permette al codice utente di poter scegliere fra un'interfaccia semplificata che copre i casi d'uso principali ed una più complessa e potente.

1.2.5 Observer - Signals

Django include un "signal dispatcher" che permette di inviare e ricevere notifiche. L'implementazione della classe **Signal** si trova in *dispatch/dispatcher.py* e si tratta chiaramente di un Observer.

L'utilizzo è illustrato brevemente in questo codice:

```
event = django.dispatch.Signal(providing_args=["a", "b"])
# ...
event.connect(my_callback)
# ...
event.send(sender=self, a=5, b="...")
```

Observer è un pattern comportamentale che definisce una dipendenza uno a molti così che, quando lo stato di un oggetto cambia, tutti coloro che dipendono da esso vengono notificati. Questo fa sì che se se A vuole essere notificata quando avviene un evento in B, basta creare una dipendenza da A a B e non una doppia dipendenza, come avverrebbe senza l'uso di questo pattern.

1.2.6 Command - Migrazioni di schema

Di grande importanza nella gestione di un database relazionali sono le migrazioni di schema, che permettono di effettuare modifiche come l'aggiunta o la rimozione di una colonna in una tabella. Eseguire direttamente i comandi SQL sul database è molto pericoloso poiché è facile commettere errori irreversibili. Per questa ragione Django mette a disposizione dello sviluppatore un tool che semplifica ed automatizza il processo.

```
python manage.py makemigrations
python manage.py migrate
```

makemigrations cerca modifiche nei models e, se ne trova, crea il codice per effettuare le migrazioni, che verrà poi invocato da **migrate**. Ogni migrazione è reversibile e può

essere effettuata su più database, la si può quindi testare sulla propria macchina di testing prima di eseguirla sul database principale.

Ognuna delle migrazioni possibili (*db/migrations/operations/*) è una sottoclasse di **Operation** (*db/migrations/operations/base.py*), che definisce i metodi *database_forwards* e *database_backwards*.

Si tratta del pattern **Command**, un pattern comportamentale che consiste nell'incapsulare una richiesta in un oggetto, permettendone la tracciabilità e la reversibilità.

1.2.7 Interpreter - Template

Django include un meccanismo di templating per generare dinamicamente pagine HTML. Nell'implementazione (*template/*) vengono usati un interpreter ed un composite.

Il template viene prima convertito da una stringa ad una lista di tokens (tramite **Lexer.tokenize()**), i quali vengono passati alla classe **Parser**, che li compila restituendo un composite di oggetti **Node**. Ogni oggetto Node ha un metodo *render*, che prende in input un **Context** e restituisce una stringa: il template interpretato nel dato contesto.

Si tratta di un'implementazione di **Interpreter**, un design pattern comportamentale che permette di definire una grammatica e di interpretarla in un dato contesto. La grammatica viene strutturata in un composite, che nel nostro caso permette la creazione di template arbitrariamente complessi. Questo pattern rende la grammatica dei template estendibile: basta creare nuovi tipi di Node.

1.2.8 Conclusioni

Ho constatato che Django utilizza moltissimo i design pattern al suo interno. Quelli che ho trovato sono solo una parte di tutti quelli usati.

E' anche interessante notare come in Python l'implementazione di certi design pattern sia molto diversa da quella in altri linguaggi, come Java. Questo per diverse ragioni:

- Alcuni pattern sono stati inseriti direttamente dentro il linguaggio, come Decorator. Altri sono rimpiazzati da altre funzionalità particolari, come Facade.
- Le funzioni sono "first-class objects", questo significa che pattern come Observer sono molto semplificati.
- Si tratta di un linguaggio dimamico.

Capitolo 2

Dependency Injection e AngularJS

2.1 Unit Testing

Lo Unit Testing è la pratica di scrivere codice per verificare la correttezza di altro codice. A differenza degli altri tipi di testing, quello di unità focalizza la sua attenzione su piccole porzioni di codice (Da qui il nome "unit testing"). Questo permette di trovare velocemente tutti quei piccoli errori che spesso passano inosservati e portano a dispendiose sessioni di debug. Con lo Unit Testing si ha un'investimento iniziale di tempo nella scrittura del test e si ha un ritorno sotto forma di tempo risparmiato nel suo debug.

È fondamentale notare come in quasi nessun caso gli unit test possano garantire la correttezza del codice: viene verificata la correttezza di una data funzionalità con un certo input, non con tutti gli input possibili. Si tratta certamente di una forte limitazione, ma l'utilità pratica dei test resta comunque enorme.

Una delle proprietà più interessanti degli unit test è che permettono di effetuare il *refactoring* del codice in sicurezza. Per refactoring si intende la ristrutturazione di una porzione di codice mantenendone inalterate le funzionalità. Viene effettuato spesso per renderlo più leggibile, più flassibile o più efficiente. Si tratta spesso di grossi cambiamenti che introducono svariate regressioni. Nel caso fosse presente un'ampia collezione di unit

test tuttavia, il programmatore saprà di non aver introdotto bug con un buon margine di sicurezza. Proprio per questo è sconsigliabile effettuare refactoring in assenza di test.

Mentre nei linguaggi statici come il Java molti errori vengono trovati già nell'analisi del compilatore, in quelli dinamici come il Javascript si ha un compromesso in cui si sacrifica questa capacità di trovare errori subito in cambio di una maggiore libertà e potenza espressiva. È quindi naturale che l'utilizzo degli unit test sia più diffuso nei linguaggi dinamici, proprio perchè permette di trovare velocemente gli errori che in linguaggi statici sono trovati dal compilatore.

Ecco un elenco riassuntivo dei vantaggi e degli svantaggi degli unit test:

- La quantità di codice da scrivere è maggiore (spesso più del doppio).
- Molto meno tempo passato a fare debug.
- Protegge da regressioni, permettendo di fare refactoring velocemente.
- Dà una maggiore sensazione di sicurezza al programmatore.

Lo scopo degli unit test non è tanto creare codice più corretto, quanto risparmiare tempo nel medio/lungo periodo.

2.1.1 Test Driven Development

Il Test Driven Development è una metodologia di sviluppo in cui i test sono scritti prima del codice dell'applicazione. Nella sua forma più pura, il programmatore non deve mai aggiungere codice se non vi è un test che fallisce. Il vantaggio principale è chiaramente che la copertura sarà maggiore. Un altro aspetto molto interessante è che aiuta nel design dell'applicazione: scrivendo il codice di verifica prima di quello da controllare, il programmatore si mette dal punto di vista degli utilizzatori delle interfacce create, portando spesso ad API più semplici e coerenti.

2.2 Dependency Injection

La dependency injection è una tecnica per gestire le dipendenze, le quali sono fornite alla classe che le necessita a run-time come parametro del costrutture o di un qualche metodo. Rispetto al tipico approccio in cui la classe provvede a risolvere da sola le proprie dipendenze, la dependency injection offre grande flessibilità, in quanto permette di sostituire componenti a proprio piacimento (Inversion of Control Principle: le classi dipendono da interfaccie astratte, non da componenti concrete.)

Un'altra interessante proprietà di questa tecnica è che poiché le dipendenze sono elencate esplicitamente nel costruttore, il design dell'applicazione risulta più chiaro e trasparente.

Dato che questa tecnica porta comunque ad una maggiore complessità, ci si potrebbe chiedere se non convenga utilizzarla di rado e solo in quei contesti in cui si è sicuri che le dipendenze abbiano più implementazioni possibili. In realtà il motivo più diffuso per l'utilizzo della dependency injection è che spesso è l'unico modo sensato per scrivere degli unit test.

2.2.1 Dependency Injection, Unit Test e Mock Objects

Ipotizziamo di avere un programma composto da due componenti A e B, in cui A dipende da B. Senza l'utilizzo della dependency injection si hanno alcuni grandi problemi nello scrivere unit test:

- Un bug nella componente B causerebbe il fallimento dei test della componente A anche se quest'ultima non avesse difetti. Questo rende più difficile individuare gli errori in quanto non si può sapere se il bug appartiene alla componente testata o ad una delle sue dipendenze.

- Spesso si vuole testare il comportamento di A su certi comportamenti particolari di B. I nostri test tuttavia non hanno alcun controllo su B, il che rende difficilissimo far sì che questi casi si presentino.

- B potrebbe essere una connessione ad un servizio esterno (come un altro programma, un'interfaccia REST o un database). Senza dependency injection dovremmo creare delle versioni fasulle di questi elementi, cosa che richiederebbe tantissimo lavoro.

La soluzione sta nell'utilizzo di **mock objects** iniettati tramite dependency injection. I mock object sono delle istanze di classi definite nei test stessi. Queste classi implementano la stessa interfaccia delle classi che vogliono andare a sostituire (nel nostro caso B), ma hanno un'implementazione creata appositamente per far verificare certe condizioni. Questa tecnica risolve elegantemente tutti e tre i problemi elencati. Il lato negativo sta chiaramente nel codice in più da scrivere, ma spesso questo non risulta un problema:

- Certi linguaggi di programmazione dinamici permettono di sovrascrivere direttamente i metodi della dipendenza.
- Nei linguaggi statici basta scrivere un solo mock parametrico per ognuna dipendenza: nei parametri viene specificato il comportamento da assumere nelle varie casistiche (spesso si tratta semplice dei valori di ritorno da restituire per ogni metodo).
- Esistono molti framework che automatizzano questo processo.

2.3 Dependency Injection e AngularJS

AngularJS[1] è un framework Javascript creato da Google per sviluppare "single-page applications". L'uso di questo framework impone di strutturare le applicazioni in un Model-View-Controller così strutturato:

Model Si tratta dell'unico elemento su cui AngularJS non impone vincoli. Può essere sia un modulo utilizzante le funzionalità del framework (un cosiddetto "service"), sia del semplice codice Javascript.

[1] http://angularjs.org/

View Viene usato un HTML potenziato da tag e funzionalità speciali messe a disposizione da AngularJS. Fondamentalmente le pagine HTML diventano dei template che sono poi dinamicamente compilati e controllati dal framework.

Controller Rappresenta l'anello mancante fra i dati, il model, e l'interfaccia, la view. Nei controller viene infatti inserita la logica dell'applicazione. I controller non hanno alcuna dipendenza sulle view, le quali invece dipendono dai dati e le funzioni che i controller mettono a loro disposizione, in quello che AngularJS chiama "**scope**".

Questo sistema permette di definire l'interfaccia in un linguaggio dichiarativo, evitando al codice imperativo Javascript di dover gestire il DOM.

Ecco come possono essere definiti un service ed un controller in AngularJS:

```
angular.module('dyanote')
// Definizione del service 'notes'
.service('notes', function ($http, noteResource) {
  this.notes = [];
  this.loadAll = function () { ... };
  ...
})
// Definizione di un controller 'NotesCtrl'
.controller('NotesCtrl', function ($scope, $location, notes) {
  notes.loadAll();
  // Questi elementi potranno essere utilizzati dalla view.
  $scope.notes = notes.notes;
  $scope.deleteNote = function (id) { ... };
});
```

2.3.1 Unit testing in AngularJS

AngularJS è stato concepito fin dall'inizio per facilitare la testabilità delle applicazioni ed è integrato con:

- Jasmine[2], una libreria Javascript per testare codice.

- Karma[3], un test-runner per automatizzare l'esecuzione dei test.

Tuttavia ciò che contraddistingue veramente AngularJS è il suo utilizzo della dependency injection. Ogni componente definibile nel linguaggio di AngularJS (controllers, directives, services) elenca le proprie dipendenze al momento delle definizione e lascia al framework la responsabilità di risolverle. Oltre a ridurre le responsabilità delle componenti, questo permette di iniettare dei mock objects durante l'esecuzione dei test.

```
describe('Controller: NotesCtrl', function () {
  beforeEach(module('dyanote'));

  var NotesCtrl, scope;

  beforeEach(inject(function ($controller, $rootScope) {
    var notesMock = {
          notes: ...
          loadAll: ...
    };

    scope = $rootScope.$new();
    NotesCtrl = $controller('NotesCtrl', {
      $scope: scope,
      notes: notesMock
    });
  }));
  ...
```

[2]http://pivotal.github.io/jasmine/
[3]http://karma-runner.github.io/

Molto spesso non è necessario neanche creare mock objects poiché Jasmine permette di sovrascrivere il comportamento delle componenti nei vari test (Questo metodo però non permette di evitare la fase di inizializzazione della componente):

```
it('should load all notes', function () {
  spyOn(notes, 'loadAll').andReturn([...]);
  ...
  expect(scope.notes).toEqual([...]);
});
```

2.3.2 Conclusioni

Uno dei maggiori punti di forza di AngularJS sta nella sua testabilità, che è permessa da un largo uso nel framework della dependency injection.

Capitolo 3

Dyanote

Dyanote è un'applicazione di note-taking in cui le note sono organizzate in un ipertesto. Ci sto lavorando da circa un anno.

Homepage http://dyanote.com/

Client https://github.com/MatteoNardi/dyanote-client

Server https://github.com/MatteoNardi/dyanote-server

3.1 Motivazioni personali

Il mio interesse per un'applicazione del genere è nato quando iniziai ad utilizzare Tomboy[1]. Ciò che mi colpì fu la soddisfazzione che mi dava tenere ordinate le mie note: il fatto che ogni pagina fosse collegata a delle altre mi dava la sensazione di star creando qualcosa di coerente e di valore, sensazione molto diversa da quella di avere tante note scollegate come avviene nella maggior parte delle applicazioni di note-taking.

Rispetto ai sistemi basati su tag che sono così diffusi oggi, un'ipertesto si avvicina di più al modo di pensare umano, che avviene per associazione.

[1] https://wiki.gnome.org/Apps/Tomboy

Sono due le grosse limitazioni di Tomboy che mi hanno spinto a creare Dyanote: il programma funziona bene solo su Linux e non vi è nessun buon meccanismo di sincronizzazione.

3.2 Requisiti dell'applicazione

Dyanote è un'applicazione di note-taking che permette di gestire note ed appunti personali dentro ad un ipertesto.

Ecco un'elenco delle funzionalità messe a disposizione:

- Effettuare il login utilizzando email e password.
- Effettuare il logout.
- Creare nuove note.
- Eliminare note.
- Modificare le note utilizzando un editor WYSIWYG.
- Navigare fra le note cliccando sui link.
- Cercare le note tramite una barra di ricerca.
- Visualizzare le note aperte in un breadcrumb.

Le note sono composte da un titolo e da del testo formattato che può contenere del grossetto, del corsivo, dei sottotitoli, dei link e degli elenchi.

L'applicazione faciliterà lo sviluppo di alberi di note:

- Vi sarà sempre una nota principale che non potrà essere eliminata.
- Ogni altra nota deve avere un padre (una nota con un link ad essa).
- Se una nota resta senza padre, viene eliminata.

Fra le note vi potranno comunque essere link "leggeri" (non di paternità). In altre parole il grafo delle note sarà sempre connesso ed avrà sempre un albero ricoprente. Questi vincoli aggiuntivi serviranno a forzare un ordine fra le note, rendendo più facile un corretto utilizzo di Dyanote.

3.3 Tecnologie e strumenti usati

Ecco qui una panoramica delle tecnologie e degli strumenti che ho utilizzato per sviluppare Dyanote. Mi sono focalizzato sulle ragioni che mi hanno portato a sceglierli più che sulle loro descrizioni approfondite, per le quali rimando alle loro documentazioni ufficiali.

3.3.1 Git

Per gestire il codice di Dyanote mi serviva chiaramente un Version Control System, il quale mi avrebbe dato una buona gestione della cronologia, la sicurezza di poter tornare a commit precedenti e, nel caso un giorno dovessi lavorare con altri programmatori, un buon strumento di collaborazione.

Ho scelto di utilizzare Git[2], un DVCS (Distributed Version Control System) che spicca per la sua versatilità ed efficienza, caratteristiche che l'hanno reso uno dei VCS più popolari. Inoltre questa scelta mi ha permesso di usare Github[3], che offre repository gratuiti per i progetti open source (Dyanote è sotto licenza MIT).

3.3.2 Django

Ho dovuto scegliere che linguaggio e che framework utilizzare per scrivere il server. Inizialmente avevo pensato di utilizzare Go[4], che mi sembrava un ottimo linguaggio di programmazione e mi avrebbe permesso di fare deploy su AppEngine. Go ha quasi tutto quel che serve incluso nella propria libreria standard, un po' come nel Python.

[2] http://git-scm.com/
[3] https://github.com/
[4] http://golang.org/

Dopo diverso tempo ho però rivalutato questa scelta perché la maggior parte del mio tempo veniva utilizzata per scrivere funzionalità, come le gestione degli utenti, che in altri framework erano già presenti.

Nel frattempo ho conosciuto Django e mi sono accorto che offriva diverse funzionalità già pronte che mi avrebbero fatto comodo, in particolare la gestione degli utenti e l'interfaccia amministrativa automatica. A differenza di Go sono inoltre disponibili in rete moltissime componenti aggiuntive, perlopiù dovute alla marturità ed alla diffusione del framework.

3.3.3 Django REST framework

Per quanto Django non impedisca di sviluppare interfacce REST, bisogna rendersi conto che non si tratta del suo scopo principale: sviluppare siti dinamici che utilizzino form e template.

Sono disponibili diverse componenti Django di terze parti che si prefiggono questo scopo ed ho scelto di utilizzare Django REST framework[5]. Fra le funzionalità messe a disposizione spiccano:

- Integrazione con **OAuth2**, che avevo già scelto di usare come meccanismo di autenticazione poiché lo ritengo un approccio più pulito dell'uso dei cookie.
- "**Web browseable API**", funzionalità che permette di navigare l'interfaccia REST in un browser. È utile sia come documentazione, sia per fare dei test.
- Meccanismi di **serializzazione** che permettono di offrire la stessa interfaccia REST in formati diversi: XML, Json ecc. (La Web browseable API non è altro che una serializzazione a HTML) Il framework è in grado di decidere da solo che formato utilizzare in base all'header "Accept" della richiesta HTTP.
- Le View di Django sono potenziate per meglio adattarsi allo sviluppo di interfacce REST: tramite le "**Generic View**" basta specificare il modello, il serializzatore e

[5]http://www.django-rest-framework.org/

le azioni che si vogliono supportare (creare, elencare, aggiornare e/o eliminare) per avere in automatico un'interfaccia completa.

Utilizzando questi framework ho ridotto di moltissimo la quantità di codice da scrivere, velocizzando lo sviluppo ed evitando inutili bug. Al momento il server è composto da meno di 1000 righe di codice: pochissimo considerando tutte le funzionalità offerte.

3.3.4 AngularJS

Creare un'applicazione complessa in Javascript richiede molta attenzione: è facile mischiare la logica dell'applicazione con la manipolazione del DOM, creando codice rigido e disorganizzato. Per questa ragione negli anni sono nati numerosissimi framework che propongono un loro Model View Controller per risolvere questo problema.

Dopo aver passato molto tempo a studiare il linguaggio ed a valutare i vari framework[6], ho deciso di utilizzare AngularJS per le seguenti ragioni:

- È un framework completo che presuppone un determinato modo di fare le cose. Lascia al programmatore meno libertà rispetto ad un framework come BackboneJS, questo significa meno margine di errore per una persona con poca esperienza come me.
- È molto usato nell'ultimo periodo. Lungi dall'essere un buon motivo per fidarsi ciecamente, il fatto che molte persone più esperte di me abbiano deciso di usarlo è rassicurante.
- A differenza di altri framework più invasivi, come ExtJS, AngularJS cerca di aumentare le potenzialità delle tecnologie che lo circondano (CSS e HTML) piuttosto che sostituirle. Inoltre si integra bene con altre librerie Javascript (jQuery, RequireJS ecc.). Questo significa sia che posso riutilizzare le mie conoscenze precedenti, sia che se mai dovessi abbandonare AngularJS non dovrei ricominciare proprio da capo.

[6]Si è rivelato molto utile il sito http://todomvc.com/, dove la stessa applicazione di esempio è sviluppata in tanti framework diversi.

3.3.5 Karma e Jasmine

Per le ragioni che ho elencato nel capitolo 2, ho pensato fosse un bene scrivere molti unit test. Ho scelto di utilizzare gli strumenti consigliati da AngularJS: il *test-runner* Karma[7] e la libreria Jasmine[8].

Poiché AngularJS necessita di un DOM per funzionare, è necessario eseguire i test dentro un browser. Ho quindi configurato Karma in modo da fargli utilizzare **PhantomJS**[9], una specie di browser senza interfaccia grafica ("PhantomJS is a headless WebKit scriptable with a JavaScript API.").

Sono molto soddisfatto di questi due strumenti: Jasmine rende i test veloci da scrivere e Karma veloci da eseguire. Mi è spesso capitato di fare test driven development.
Se avessi deciso di non scrivere unit-test per Dyanote, lo sviluppo sarebbe stato molto più lento.

3.3.6 WYSIHTML5

Mi serviva un editor "What You See Is What You Get" che fosse:

- Completamente personalizzabile dal punto di vista grafico.
- Compatibile con il mio formato delle note, che è un sottoinsieme dell'HTML.
- Estendibile con funzionalità aggiuntive.

L'editor che più mi ha convinto è stato WYSIWHTML5[10], che sfrutta l'attributo "contenteditable" di HTML5, permettendo quindi una personalizzazione completa tramite CSS. Non sono sorti grossi problemi riguardo il supporto al mio formato e sono riuscito ad integrarlo con il mio Javascript, seppur con qualche difficoltà. A causa di qualche bug rimasto sto valutando di scrivere direttamente il mio editor WYSIWYG utilizzando

[7] http://karma-runner.github.io/
[8] http://pivotal.github.io/jasmine/
[9] http://phantomjs.org/
[10] http://xing.github.io/wysihtml5/

contenteditable, cosa che mi permetterebbe la massima flessibilità. Trattandosi tuttavia di una grossa mole di lavoro (WYSIWHTML5 ha circa 10 mila righe di codice) lascerò quest'aspetto ad un secondo momento.

3.3.7 Bootstrap e Less

Non volendo spendere immediatamente molto tempo nello sviluppo del CSS, ho deciso di utilizzare Bootstrap[11], che offre dei fogli di stile ben fatti e con una grafica che si adatta a quasi tutti i contesti. Bootstrap da un buon punto di partenza su cui poi fare modifiche per imprimere il proprio brand.

Ho inoltre deciso di utilizzare LESS, un linguaggio compilabile in CSS che offre molte funzionalità per facilitare lo sviluppo dei fogli di stile: permette infatti di includere file esterni, effettuare operazioni matematiche, dichiarare costanti e macro, ecc.
Essendo Bootstrap stesso scritto in LESS, ho potuto personalizzarlo semplicemente sovrascrivendo alcune variabili.

3.3.8 Yeoman, Grunt e Bower

Cominciare un nuovo progetto è sempre difficile: si deve decidere l'organizzazione delle directory, installare le dipendenze, scrivere il codice di inizializzazione, creare il build system ecc. Seguendo il consiglio degli sviluppatori stessi di AngularJS[12], ho deciso di usare Yeoman[13]. Si tratta di uno "scaffolding tool", ovvero uno strumento per automatizzare la creazione di nuovi progetti.

Yeoman è l'integrazione di tre componenti:

yo è il vero e proprio generatore di nuovi progetti. Il programmatore può installare uno dei "generators" (Ad esempio quello per AngularJS), ed invocarlo tramite un comando da terminale. Questo crea in automatico un'applicazione di demo pronta per essere

[11] http://getbootstrap.com/
[12] http://blog.angularjs.org/2012/09/yeoman-and-angularjs.html
[13] http://yeoman.io/

modificata. Oltre a creare nuovi progetti da zero, è possibile creare singole componenti: il comando "yo angular:controller user" ad esempio, permette di creare un nuovo controller di nome *UserCtrl*.

Bower è lo strumento usato per gestire le dipendenze: permette di aggiungere, rimuovere e aggiornare le componenti esterne. Questo permette di specificare esattamente che versioni delle librerie usare (evitando perciò problemi di incompatibilità) senza doverle includere nel VCS.

Grunt è un build system per applicazioni client-side ed è caratterizzato da un'enorme disponibilità di plugin. Tutto quello che deve dare il programmatore è installare i plugin necessari e configurarli nel *Gruntfile*. Questo permette di automatizzare un sacco di passaggi, nel caso di Dyanote Grunt viene usato con i seguenti target:

- *test*: fa partire Dyanote su un web server in locale e lancia Karma, che esegue tutti gli unit test ogni volta che viene modificato un file.
- *server*: compila i file LESS, lancia Dyanote su un web server locale, lancia il browser ed ha la responsabilità di rieseguire i passaggi tutte le volte che viene modificato un file.
- *build*: compila i file LESS, concatena e minimizza i file Javascript e CSS, copia il risultato nella cartella *dist*.
- *deploy*: fa il deploy su Amazon S3 prendendo le credenziali dalle *environment variables*.

3.4 Deployment

Uno dei problemi maggiori nello sviluppo di Dyanote è stato decidere dove e come fare il deployment, che dove soddisfare i seguenti requisiti:

Basso prezzo di partenza Se mai Dyanote avesse successo, il suo business model sarebbe simile a quello di Evernote: certi utenti pagano annualmente per sbloccare

funzionalità extra. Poiché di utenti paganti ancora non ce ne sono, mi serve una soluzione con costi iniziali irrisori.

Scalabilità Dyanote avrà picchi di utilizzo ogniqualvolta apparirà su siti di informazione. Per questo è necessaria una soluzione in grado di scalare: non ci possiamo permettere di smettere di funzionare nel momento di maggior bisogno, in cui così tanti potenziali utenti si stanno facendo un'idea sull'applicazione.

Genericità Non voglio "lock-in" della piattaforma: i bisogni di Dyanote in futuro potrebbero cambiare, quindi lo stack usato deve essere facilmente replicabile su un proprio server. Questo esclude ad esempio AppEngine.

Performance I tempi di caricamento devono essere bassi.

3.4.1 Heroku

Ho cercato una piattaforma PaaS (Platform as a Service) che supportasse Python e Django. Ho deciso di utilizzare Heroku[14] perchè soddisfa tutti i quattro requisiti ed offre un ottimo ambiente di sviluppo:

- La documentazione è sintetica ma completa.
- Il deploy viene fatto tramite Git (si integra quindi bene nel mio workflow).
- Segue le migliori pratiche nello sviluppo delle applicazioni Web[15].

Inizialmente avevo pensato di usare Heroku anche per hostare l'applicazione in AngularJS, ma ho subito avuto qualche difficoltà dovute al fatto che il client necessita di essere compilato con il suo build system (Grunt). La soluzione più semplice sarebbe stata eseguire il build in locale e fare l'upload solo della cartella *dist*, ma ciò avrebbe presupposto l'inserimento di file compilati dentro a Git. Mi sembrava una cattiva idea avere file generati nel repository principale, come non mi ispirava l'opzione di creare un repository solo per fare il deploy su Heroku.

[14] https://www.heroku.com/
[15] http://12factor.net/

Ho quindi deciso di configurare Heroku col *buildpack* misto[16], integrando così Python e Node.js (usato per lanciare Grunt).

Facendo più ricerche sul modo giusto di fare il deploy di file statici, ho scoperto che sarebbe stata una buona idea usare un server apposito, specializzato per quel compito e più efficiente. Ho quindi deciso di usare Amazon S3 per *dyanote.com* e Heroku per *api.dyanote.com*.

3.4.2 Amazon S3

Amazon S3 è un file storage web service offerto da Amazon. I costi sono bassi e le performance molto elevate, si tratta inoltre di una piattaforma molto usata, il che rende facile trovare documentazione, risorse e strumenti per integrarla al proprio workflow (In Grunt ad esempio uso un plugin apposito per fare il deployment).

Per poter sfruttare S3 per fare l'hosting di *dyanote.com*, serve un altro servizio di Amazon: Route 53. Si tratta di un DNS integrato con gli altri servizi Amazon.

Sono stato ispirato da un articolo[17] trovato in rete che proponeva una strategia per migliorare le performance.
L'articolo tratta in particolare il problema delle cache e propone una soluzione particolarmente elegante, utilizzando quello che l'autore chiama uno "*scout file*":

- Ad ogni deploy viene creata una cartella *#buildNum* che contiene tutti i file statici: javascript, html, css, immagini ecc. Ogni file in questa cartella ha gli header delle cache impostati all'infinito: un browser non richiederà mai lo stesso file.
- Nella cartella di root viene creato lo scout file: un *index.html* dove vengono linkate le altre risorse. Gli headers di questo file sono impostati per fare cache solo di qualche minuto.

In questo modo si riducono al minimo le richieste HTTP, migliorando la user-experience degli utenti e richidendo meno risorse al proprio server.

[16] https://github.com/ddollar/heroku-buildpack-multi
[17] https://alexsexton.com/blog/2013/03/deploying-javascript-applications/

3.5 Risultato

L'applicazione è disponibile all'indirizzo *http://dyanote.com/*.
Al primo accesso viene chiesto all'utente di autenticarsi con il suo indirizzo email e la sua password. Si noti che non è ancora possibile registrarsi: Dyanote è in uno stato di Beta e voglio un incremento graduale degli utenti fino a quando non avrò risolto gli ultimi bug.

Effettuato il login, all'utente è mostrata una schermata così composta:

- In alto è presente una **barra di navigazione** contenente un link alla homepage, un bottone per effettuare una ricerca testuale fra le note e un menu utente in cui si può fare il logout.
- A sinistra c'è un **breadcrumb** dove sono elencate le note attualmente aperte. Questo elenco comincia dalla nota di "Root" e termina con l'ultima nota aperta. Cliccando su uno di questi link viene mostrata la nota corrispondente tramite uno scrolling animato.
- Al centro sono presenti, una in successione all'altra, tutte **le note aperte**. Sia il titolo che il contenuto di ogni nota è modificabile tramite un rich-text editor. Le modifiche sono automaticamente sincronizzate con il server.

All'avvio viene sempre aperta la nota di "Root", per aprirne altre bisogna cliccare sui link ad esse. L'elenco delle note aperte comincia sempre con la nota principale e finisce sempre con l'ultima nota aperta, in mezzo vi sono le note che collegano queste due. Quando viene aperta una nuova nota vengono chiuse tutte le note aperte che non siano sue antenate, viene effettivamente aggiunta la nota e viene fatto scrolling su di essa. In alto a destra di ogni nota è presente un tasto per archiviarla.

A destra di ogni nota è presenta una **toolbar** contenente vari bottoni:

Link È il comando principale: crea una nuova nota ed inserisce un link ad essa. Se nell'editor è selezionato del testo al momento in cui viene invocata l'azione, la nuova nota avrà come titolo il testo selezionato, altrimenti ne verrà generato uno automaticamente.

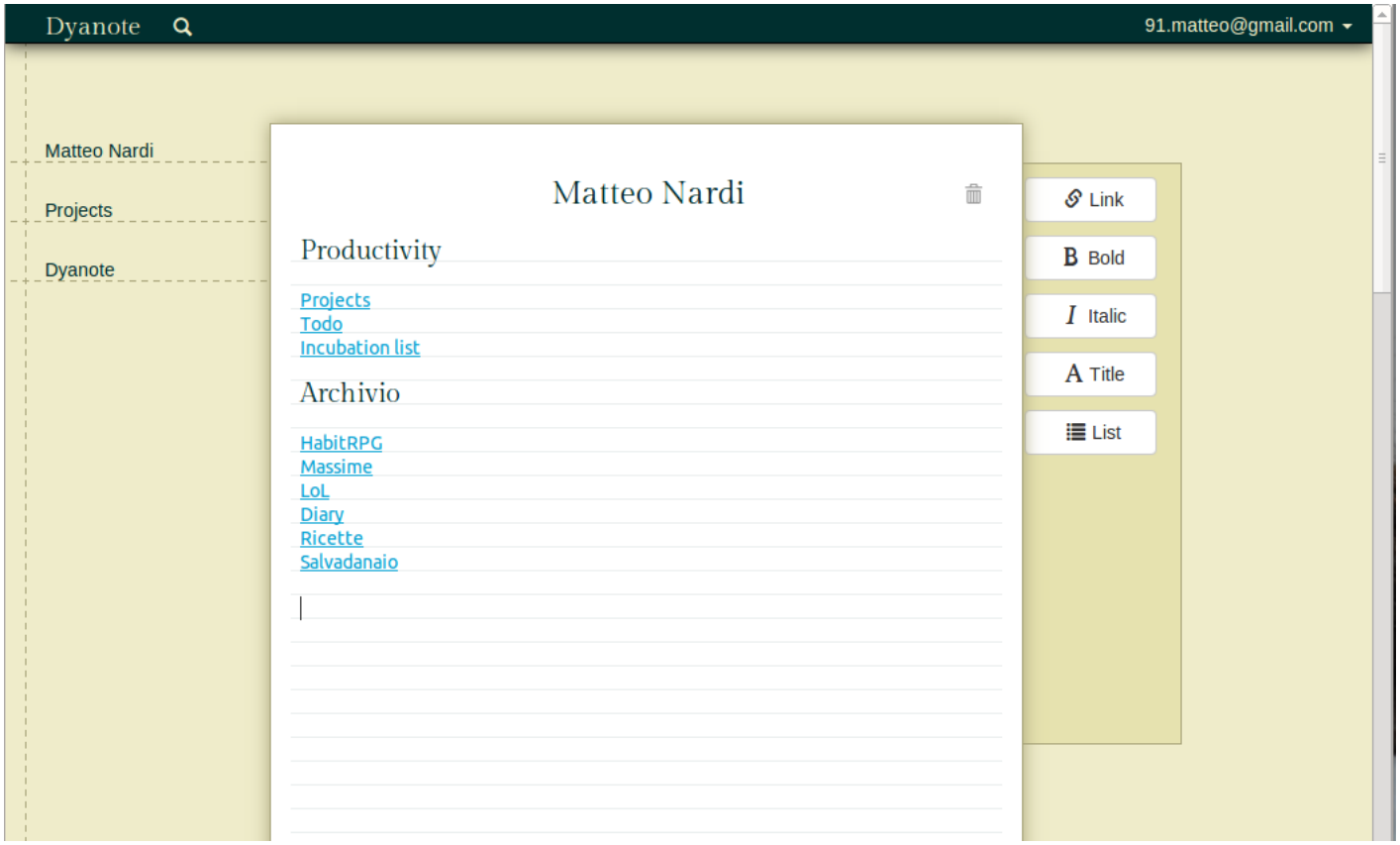
Bold Assegna uno stile grossetto al testo selezionato.

Italic Assegna uno stile italic al testo selezioato.

Title Rente il testo selezionato un "titolo" (graficamente diventa più grande e viene colorato in verde).

List Inserisce una lista puntata.

Cliccando sul bottone di **ricerca** presente nella barra di navigazione o premento la shortcut *Ctrl+F* può essere effettuata una ricerca fra le note. Basterà poi digitare le parole da cercare e verrrà istantaneamente mostrato un elenco delle note che soddisfano la ricerca (La ricerca viene aggiornata ad ogni nuovo carattere immesso). Per aprire una nota basta selezionarla nell'elenco dei risultati.



3.5.1 Conclusioni e sviluppi futuri

Dyanote ha ancora molti bug, in particolare nel rich text editor e nel supporto a browser datati. Inoltre mancano molte funzionalità che voglio implementare: un meccanismo di archiviazione, una visualizzazione grafica del grafo delle note, la possibilità di aggiungere immagini ed altri elementi decorativi ecc. Tuttavia sono molto soddisfatto del mio lavoro per due ragioni: Dyanote ha già abbastanza funzionalità per essere utile (Ho iniziato io stesso ad usarlo giornalmente per le mie note), inoltre aggiungere nuove funzionalità ora che ho preso confidenza con i framework sarà semplice e credo che entro un anno sarà pronto per il grande pubblico.