Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

PROGETTAZIONE E SVILUPPO DI UN WEB DESKTOP MEDIANTE GOOGLE WEB TOOLKIT

Relatore:
Chiar.mo Prof.
Fabio Panzieri

Presentata da:
Silvia Righini

Correlatore:
Ing. Alberto Torrini

Sessione III
Anno Accademico 2010/2011

Indice

Introduzione

Questa tesi descrive la progettazione e lo sviluppo di un prototipo di *Rich Internet Application* presso Wincor Nixdorf, azienda tedesca che offre prodotti e servizi ai settori bancario e di vendita al dettaglio, presente in Italia dal 1999.
Una *Rich Internet Application* (*RIA*), è un'applicazione web in tutto e per tutto simile a un'applicazione come siamo abituati a conoscerla, installata sul computer dell'utente finale. Nonostante l'aspetto grafico delle RIA le accomuni alle applicazioni *desktop*, esse sono fruibili attraverso il browser e non richiedono installazione. La loro "ricchezza" è costituita dalla maggiore interattività e conseguente coinvolgimento per l'utente che sono in grado di fornire rispetto alle tradizionali applicazioni web.

Nella realtà italiana di Wincor Nixdorf, è nato nel 2006 NRGine, un progetto il cui scopo è semplificare e velocizzare la produzione di software. NRGine è un framework che permette l'implementazione di applicazioni *Enterprise Resource Planning* (*ERP*), con supporto alla *Service Oriented Architecture* e dotate di un'interfaccia grafica personalizzata e accessibile via browser.
Tale interfaccia web è la RIA oggetto di questa tesi. È modellata come un ambiente *desktop*, da cui il nome *web desktop*: l'intera finestra del browser dedicata alla visualizzazione della pagine web è utilizzata come spazio contenitore per una serie di elementi, tipicamente finestre. Tali finestre sono selezionabili a partire da un menu di avvio al quale si accede da una barra in basso, e contengono griglie o form che presentano i dati propri dell'applicazione.

Questo web desktop è stato sviluppato in AJAX, e per l'aspetto grafico si affida alla libreria JavaScript ExtJS, sviluppata e commercializzata dalla

Sencha Inc., che fornisce una collezione di elementi grafici, come menu, finestre, e pulsanti, già pronti per essere inseriti in un'applicazione web.
L'insieme di tecnologie che va sotto il nome di AJAX permette di sviluppare applicazioni web fortemente interattive, dal momento che JavaScript è eseguito direttamente nel browser dell'utente finale. Le caratteristiche di JavaScript non lo rendono però il candidato ideale per sviluppare applicazioni di grandi dimensioni. È un linguaggio molto flessibile e debolmente tipizzato, in cui non è immediato forzare determinate *best practices*.
L'interprete JavaScript non è in grado di effettuare controlli rigorosi, dal momento che i suoi costrutti più dinamici non sono analizzabili staticamente. Per le stesse ragioni è scarsamente supportato dagli strumenti di sviluppo, che non possono offrire funzionalità utili come ad esempio l'autocompletamento.
Diversi browser, inoltre, forniscono implementazioni diverse.

Sebbene l'uso di ExtJS avesse semplificato il codice del web desktop, tale soluzione non era stata considerata da Wincor Nixdorf ottimale: il mantenimento, ed eventuali ulteriori sviluppi, del web desktop di NRGine era complesso e oneroso e i responsabili del *presentation layer* avevano valutato l'eventualità di un significativo cambio di rotta, e si erano interessati a una diversa tecnologia.
La tesi che mi è stata proposta prevedeva dunque di sviluppare un prototipo di un diverso web desktop, analogo dal punto di vista grafico e funzionale al web desktop di NRGine già esistente, ma implementato mediante *Google Web Toolkit* (GWT).

GWT è un framework sviluppato da Google, nato nel 2006 in risposta alle stesse problematiche che Wincor Nixdorf ha riscontrato nella sua applicazione web in AJAX. I suoi ideatori, Bruce Johnson e Joel Webber, da un lato erano rimasti colpiti dalle potenzialità di AJAX e delle applicazioni con esso sviluppate, ma dall'altro non erano persuasi che tale insieme di tecnologie fosse davvero valido nel contesto di applicazioni sempre più complesse e in cui molti programmatori fossero coinvolti, le quali avrebbero richiesto la possibilità di astrarre, di riutilizzare codice, e di creare architetture modulari.

La loro proposta è stata quindi quella di un *cross compiler*, un compilatore che compilasse codice Java in JavaScript che potesse essere eseguito su browser diversi con sistemi operativi diversi. Con GWT, gli sviluppatori possono creare le loro applicazioni web in Java, utilizzando gran parte della semantica e della sintassi Java, solo una minima parte delle librerie, ma con l'aggiunta di API e di una libreria di elementi grafici fornite da Google.

Durante la fase di sviluppo, il programmatore potrà beneficiare di tutti gli strumenti di sviluppo presenti sul mercato per Java, con tutte le loro funzionalità, ed eseguirà l'applicazione come Java bytecode all'interno della Java Virtual Machine, grazie ad una modalità parte del *toolkit* detta appunto *development mode*.
Anche il debug potrà dunque avvenire in Java, con la possibilità di controllare l'esecuzione passo passo e di settare *breakpoints*.
A sviluppo terminato, l'applicazione sarà schierabile in *production mode*: si utilizzerà il compilatore GWT per compilare la parte client dell'applicazione in JavaScript altamente ottimizzato, compresso, e *cross browser* che quindi verrà scaricato dal browser dell'utente finale ed eseguito, come JavaScript ovviamente, nel suo browser, come una normale applicazione AJAX.

Come accennato, il web desktop di NRGine era stato sviluppato con AJAX e una libreria di nome ExtJS: la stessa compagnia, Sencha Inc., che ha creato tale libreria ha rilasciato nel 2008 una libreria analoga per GWT: ExtGWT. Ciò significa che gli stessi elementi grafici, o widget, con il medesimo aspetto grafico, sono ora disponibili per applicazioni web basate su GWT. Questo è stato ovviamente un incentivo notevole per Wincor Nixdorf a voler provare il nuovo framework di Google: il nuovo web desktop avrebbe potuto essere graficamente identico al precedente, pur con un cambio completo nella tecnologia sottostante, che si auspicava avrebbe reso l'intera applicazione più mantenibile.

La proposta di tesi prevedeva una prima fase di documentazione su GWT e ExtGWT, seguita da una fase di sviluppo in cui avrei dovuto produrre un prototipo di web desktop, nello specifico una schermata di login, un desktop,

e due tipologie di finestre.
Il lavoro è iniziato a ottobre 2011, e nel momento in cui redigo questa tesi, gran parte delle interfacce richieste è stata implementata.

Per la comunicazione tra client e server si è utilizzato il meccanismo di RPC asincrone fornito da GWT. Le implementazioni dei servizi lato server interagiscono poi con altri servizi del framework NRGine già presenti, in modo analogo a come avveniva per il web desktop in ExtJS.

La parte grafica è stata, come era previsto, implementata tutta a partire dai widget forniti da ExtGWT. Anche GWT fornisce una collezione di widget, ma non sono stati utilizzati, anche se in alcuni casi avrebbero potuto esserlo, perché l'obiettivo era produrre un web desktop il più possibile simile al precedente. La documentazione ExtGWT è scarsa, ma una presentazione di interfacce grafiche con relativo codice è presente. È stato quindi in alcuni casi non immediato capire determinate classe o metodi, o peggio quale fosse l'approccio migliore per una data implementazione nel caso diversi approcci fossero possibili, ma i problemi che si sono presentati sono stati affrontati e risolti.
Il prototipo utilizza l'architettura *model-view-controller* fornita da ExtGWT, e diversi elementi dell'interfaccia grafica interagiscono tra loro mediante eventi gestiti da un controllore.

Questa tesi è così organizzata: in primo luogo, nel capitolo 1 descriverò lo scenario di Wincor Nixdorf, il framework NRGine e la proposta di tesi che mi è stata fatta.
Nel capitolo 2 introdurrò il concetto di *Rich Internet Application* ed illustrerò le tecnologie coinvolte, ossia ExtJS, *Google Web Toolkit* e ExtGWT. In particolare mi soffermerò, nella sezione 2.3, su GWT, le riflessioni che hanno portato alla sua nascita, i problemi che si propone di risolvere, la sua architettura, il suo compilatore e le sue caratteristiche principali.
Nel capitolo 3 riassumerò quanto si è stabilito nella fase di progettazione: illustrerò l'architettura del prototipo, e una serie di scelte progettuali che si sono prese, ad esempio relativamente alla gestione dei guasti e degli errori,

alla comunicazione client-server e all'internazionalizzazione.
Seguirà nel capitolo 4 la descrizione del prototipo realizzato. Trattandosi di una tesi di sviluppo, tale capitolo è il più ampio della relazione.
Descriverò gli strumenti utilizzati e l'organizzazione del codice, fornirò alcuni dettagli su specifici aspetti dell'implementazione e infine descriverò le interfacce grafiche prodotte.
In chiusura, nel capitolo 5 riporterò una valutazione del prototipo, e nel capitolo 6 le mie conclusioni e alcuni possibili sviluppi futuri.

Capitolo 1

Lo scenario

In questo capitolo descriverò Wincor Nixdorf, l'azienda dove ho avuto l'occasione di svolgere questa tesi, e, pur in termini generali, il loro framework NRGine, in cui il prototipo di web desktop che ho implementato si inserisce.
Illustrerò poi la proposta di tesi iniziale e le fasi previste per il suo svolgimento.

1.1 Wincor Nixdorf

Wincor Nixdorf [Win12] è un'azienda leader nel mondo nei settori della consulenza e dello sviluppo di soluzioni per il settore bancario e di vendita al dettaglio.
L'azienda nasce, nella sua prima forma, "Nixdorf Computer AG", nel 1952 ad opera di Heinz Nixdorf. Nel 1990 viene acquisita da Siemens AG, la quale integra nella sua organizzazione le divisioni "Retail Solutions" e "Banking Solutions". Segue una grande espansione nel 1995, con un'estensione dell'offerta e una maggiore internazionalizzazione.
Nel 1999 l'organizzazione viene rilevata dalla finanziaria Kohlberg Kravis Roberts e Goldman Sachs Capital Partners, e diviene infine una società indipendente, col nome Wincor Nixdorf.
Oggi Wincor Nixdorf conta oltre nove mila dipendenti nel mondo, ha proprie filiali in 42 paesi ed è rappresentata da partner in oltre 60. Dal maggio

2004 è quotata alla Borsa di Francoforte e le azioni sono scambiate nel Prime Standard della Frankfurt Stock Exchange.

Dal momento della sua separazione da Siemens, Wincor Nixdorf ha aggiunto alla propria offerta i servizi IT: alla soluzioni hardware che già proponeva ha aggiunto software, consulenza e servizi correlati, come installazione e manutenzione.
L'azienda si rivolge soprattutto al settore bancario e di vendita al dettaglio (*retail*), e i suoi prodotti principali sono i sistemi di pagamento self-service, automatici ed elettronici (POS).
Recentemente, si sta affermando anche come fornitore di servizi integrati, in modo particolare al di fuori della natìa Germania, e sta rivolgendo il suo interesse anche verso altri settori più piccoli, quali lotterie, stazioni di servizio, catene di ristorazione e aziende postali con reti estese di filiali.

Sulla scena italiana, Wincor Nixdorf è presente dal 1999, con tre società, Wincor Nixdorf, Wincor Nixdorf Retail e Wincor Nixdorf Retail Consulting che offrono servizi, prodotti e consulenza sempre ai settori bancario e retail. Ha cinque sedi sul territorio, Milano, Roma, Bologna, Potenza e Grosseto, e un organico di circa 300 persone.

1.2 Il framework NRGine

Il progetto NRGine nasce nel 2006, in considerazione dell'evoluzione in corso delle tecnologie informatiche, che, come Wincor Nixdorf correttamente prevede, si sposteranno sempre di più dal desktop al browser. Si tratta di un framework per uso interno orientato a semplificare e velocizzare la produzione software.
Si tratta di un insieme di scelte architetturali, metodologie standard e *best practice*, servizi infrastrutturali, strumenti e pattern, che permettono

- l'implementazione di applicazioni di classe ERP (*Enterprise Resource Planning*)

- la dotazione alle applicazioni di un'interfaccia grafica personalizzata e accessibile da qualunque device
- un pieno supporto alla *Service-Oriented Architecture*
- la standardizzazione della procedura di deploy e dell'ambiente run-time
- la compatibilità con database diversi
- la compatibilità con gli standard di mercato

Si tratta di un progetto molto vasto e consolidato, diviso in tre layer: *business*, *logic* e *presentation*. La struttura generale del framework, con le relazioni di dipendenza tra i diversi livelli, è riportata in figura 1.1. Il livello business si occupa dell'accesso ai dati, e fornisce una serie di servizi per accedervi.
Il livello logico implementa tutta la logica di dominio propria del framework, mentre il livello presentation, il quale interagisce sia col livello business che col livello logico, contiene l'interfaccia grafica che permette all'utente di interagire coi dati.
Tale interfaccia assume la forma di un web desktop accessibile da qualunque device (compatibilmente ovviamente con politiche di sicurezza): attraverso il web desktop è possibile visualizzare i dati applicativi, e manipolarli. Oltre a NRGine, tutte le applicazioni create con esso utilizzano come ambiente grafico tale web desktop.
Esso è composto essenzialmente di un menu dal quale è possibile selezionare la finestra contenente i dati che si desidera visualizzare. Le diverse finestre contengono form e/o griglie, e sono generate dinamicamente da NRGine a partire dalle caratteristiche dei dati applicativi.

1.3 La proposta di tesi

Il web desktop di NRGine è stato sviluppato in AJAX con l'ausilio della libreria JavaScript ExtJS, che approfondirò nella sezione 2.2. Tale soluzione si è rivelata nel tempo insufficiente per le esigenze di Wincor Nixdorf: il web

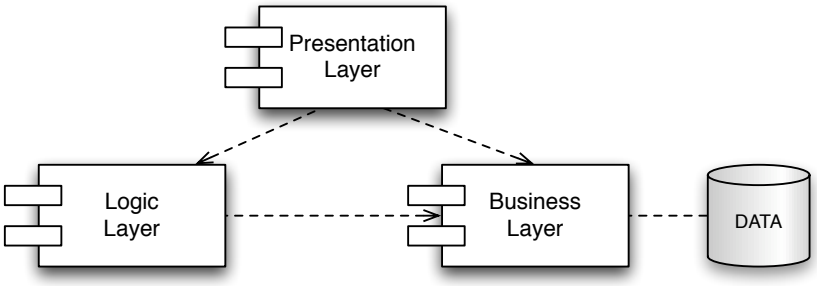

Figura 1.1: Architettura di NRGine. Il livello business interagisce coi dati applicativi. Il livello logico contiene la logica di dominio. Il livello di presentazione fornisce un'interfaccia grafica per la visualizzazione dei dati.

desktop è un'applicazione di non piccole dimensioni, e la sua gestione in JavaScript non è stata considerata soddisfacente dagli sviluppatori coinvolti, i quali hanno preso in considerazione l'idea di rivolgersi verso soluzioni alternative.
Il candidato per sostituire ExtJS è stato individuato in GWT, corredato della libreria ExtGWT; entrambi saranno illustrati nelle sezioni 2.3 e 2.4, insieme con le caratteristiche che rendono questa coppia una valida scelte per sostituire ExtJS.

Il progetto che mi è stato proposto è stato dunque di indagare lo sviluppo di un prototipo di web desktop per NRGine mediante GWT e ExtGWT. Tale prototipo avrebbe dovuto essere graficamente analogo al web desktop già esistente e in uso, e avrebbe dovuto implementare una versione ridotta ma esemplificativa delle sue funzionalità. Nello specifico, avrebbe dovuto includere

- una schermata di login
- una "desktop"
- un menu di avvio
- un esempio di finestra con griglia
- un esempio di finestra con form

Questa lista avrebbe coperto, coi suoi elementi, tutti gli oggetti grafici fondamentali del web desktop NRGine, così da poter permettere a Wincor Nixdorf di valutare se GWT/ExtGWT rispondesse alle sue esigenze e vi fosse convenienza nell'effettuare una migrazione.
L'aspettativa su ExtGWT era che tale soluzione riducesse gli errori derivanti dallo sviluppo di codice direttamente in JavaScript, consentendo una maggiore produttività, e rendesse il codice di più facile manutenzione.

La *road map* che mi è stata proposta consisteva, in sintesi, delle seguenti fasi:

1. acquisizione di competenze su GWT, creazione e compilazione di applicazioni web enterprise
2. acquisizione di competenze su ExtGWT, con particolare riguardo alle GUI utilizzate in NRGine
3. visione del modello *model-view-controller*
4. visione di esempi già sviluppati
5. creazione di un web desktop con ExtGWT
6. creazione di griglie e form con ExtGWT

Una sintesi delle informazioni raccolte durante le fasi di acquisizione di competenze verrà riportata nel capitolo 2, mentre il lavoro effettuato per il completamento delle fasi 5 e 6 verrà illustrato nel capitolo 4.

Capitolo 2

Framework AJAX per Rich Internet Applications

In questo capitolo illustrerò brevemente cosa sono le RIA, *Rich Internet Applications*, le applicazioni nella cui categoria il nostro web desktop rientra, e quali sono le tecnologie con le quali sono comunemente sviluppate.
Passerò poi ad analizzare più nel dettaglio ExtJS, nella sezione 2.2, la libreria JavaScript scelta inizialmente per l'interfaccia grafica di NRGine.
Nella sezione 2.3 introdurrò il framework GWT, e ne descriverò le caratteristiche più salienti e gli aspetti innovativi. Infine, nella sezione 2.4 illustrerò la libreria ExtGWT scelta per lo sviluppo del nuovo prototipo oggetto della tesi.

2.1 Una breve panoramica sulle RIA

Il termine *Rich Internet Applications* (RIA) fa la sua prima apparizione nel 2002 in un libro bianco della Macromedia [All02], per definire un concetto che era in realtà già noto in precedenza pur con nomi diversi: applicazioni web non dissimili dalle comuni applicazioni desktop.
Così come "Web 2.0", anche RIA è un termine generico che include, o può includere, applicazioni web anche molto diverse tra loro. La "ricchezza" propria delle RIA fa riferimento all'esperienza dell'utente: forse la sola idea condivisa alla base delle RIA è che esse siano maggiormente coinvolgenti

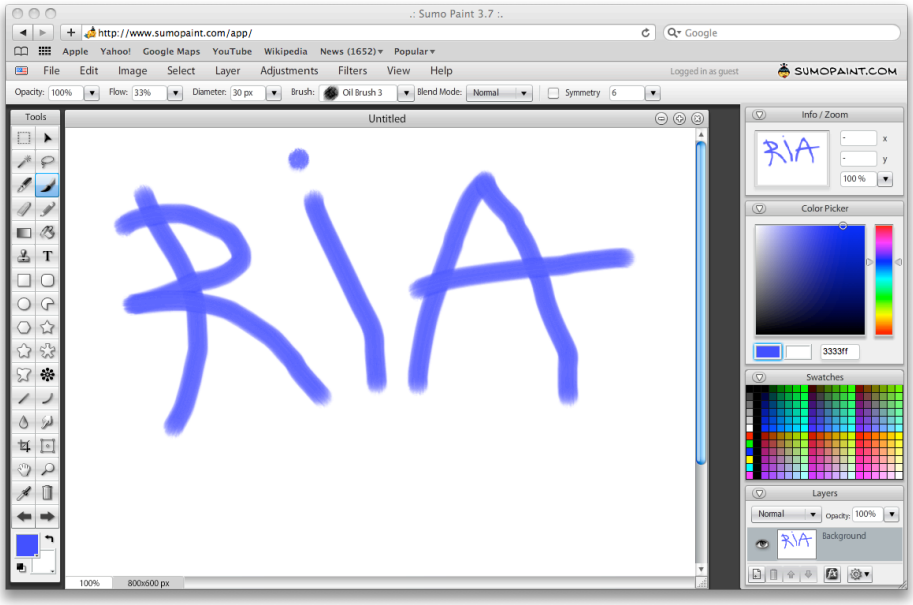

Figura 2.1: SumoPaint, editor di immagini online, è una RIA scritta in Flash.

rispetto ad altre offerte del web, grazie a una marcata interattività, e che "accorcino" le distanze tra web e desktop.
Le RIA non sono state sempre una possibilità concreta: solo con l'aumento di velocità delle connessioni internet, e con la diffusione di tecnologie più sofisticate che l'HTML degli albori del web, è diventato possibile offrire servizi più complessi e articolati. Esempi di RIA "tipiche" possono essere editor di testo online, visualizzatori di files, simil desktop... o applicazioni come SumoPaint[1] in figura 2.1.

Le tecnologie per sviluppare RIA sono svariate: tra le più popolari troviamo Adobe Flash, JavaFX e Microsoft Silverlight, ma anche Adobe AIR, Adobe Flex, AJAX, Curl, GWT, and Mozilla XUL. Alcune di queste richiedono agli utenti di installare software addizionale sulla loro macchina locale per poter utilizzare l'applicazione web: è il caso ad esempio di Flash e Silverlight, i quali richiedono un adeguato plugin per il browser.

[1] `http://www.sumopaint.com/app`

Come già accennato, in questa tesi non si è affrontata la scelta di una determinata tecnologia per la RIA da realizzare. Una RIA era già presente: il web desktop di NRGine (1.2), sviluppato mediante ExtJS, che, come vedremo nella sezione 2.2, è essenzialmente AJAX.
L'applicazione sviluppata in questo progetto è un prototipo dello stesso web desktop, implementato mediante una diversa tecnologia indicata da Wincor Nixdorf, GWT, che descriverò nella sezione 2.3.
Non è stato dunque effettuato un confronto tra le diverse tecnologie disponibili, né tanto meno una valutazione sulla loro idoneità a essere utilizzate per il progetto oggetto della tesi.

Tuttavia, posta la scelta di ExtJS effettuata in passato da Wincor Nixdorf, le ragioni della proposta dello sviluppo di nuovo prototipo in GWT sono chiare. In primo luogo, vi è l'esistenza della libreria ExtGWT, intimamente legata a ExtJS: illustrerò ExtGWT e approfondirò il rapporto tra le due librerie, nella sezione 2.4.
In secondo luogo vi sono invece le difficoltà intrinseche nel gestire un'applicazione AJAX. Vorrei dunque ora concentrarmi su AJAX e lo sviluppo di RIA con questa tecnologia.

AJAX, *Asynchronous JavaScript and XML*, è un termine che fa riferimento a un gruppo di tecnologie: include XHTML e CSS per l'aspetto presentazionale, DOM per la rappresentazione dei dati e l'interazione, XMLHttpRequest per il trasferimento di dati tra client e server, XML e XLST per tale scambio, e JavaScript, il linguaggio di scripting principe per le pagine web.
AJAX permette lo sviluppo di applicazioni web con ottime performance, dal momento che JavaScript viene eseguito direttamente nel browser dell'utente finale, e non richiede nessun software addizionale per essere eseguito sui browser moderni, diversamente dai sopracitati Flash e Silverlight.

JavaScript è un linguaggio flessibile, dinamicamente tipato, che può permettere di scrivere in modo veloce e, spesso, sintetico.
Può però non essere la soluzione ideale per grandi applicazioni, o quando diversi programmatori sono coinvolti.

La sua grande flessibilità e tolleranza agli errori sono positive in quanto possono permettere all'utente finale di utilizzare comunque un'applicazione web nonostante questa presenti problemi; per contro questo suo permissivismo fa sì che la scrittura di codice pulito e corretto stia tutta alla buona volontà del programmatore, diversamente da quanto accade in altri linguaggi che forzano determinate *best practice* con errori anche fatali.
L'interprete JavaScript non è rigoroso: i costrutti più dinamici di JavaScript non sono analizzabili staticamente. La flessibilità di JavaScript rende impossibile anche l'autocompletamento del codice, in quanto in diverse circostanze gli stessi simboli possono avere significati diversi. Questo fa sì che JavaScript sia scarsamente supportato dagli strumenti di sviluppo, rispetto ad altri linguaggi, e sia complesso da analizzare e debuggare.
L'interpretazione cambia da browser a browser, facendo sì che il codice per un'applicazione web sia arricchito con parti necessarie solo a garantire la corretta comprensione da parte di browser diversi, e che sia necessario testare tale codice appunto sul numero più ampio possibile di browser per assicurarsi il suo corretto funzionamento.
Come vedremo nella sezione 2.3, queste sono le ragioni che hanno portato alla nascita di GWT [Joh09].

2.2 ExtJS

ExtJS [Sen12b] è una libreria JavaScript per la costruzione di applicazioni web. Ha fatto la sua prima apparizione nel 2006 come una serie di estensioni per la libreria *Yahoo! User Interface* (YUI), sviluppata da Jack Slocum. Le estensioni furono poi riunite e organizzate in una libreria che prese il nome di *yui-ext*, distribuita con licenza open source BSD.
La libreria guadagnò velocemente popolarità, e nello stesso anno cambiò nome in Ext. Un compagnia venne fondata, e la licenza divenne doppia, open source per usi personali, educativi e in generale no-profit, e proprietaria per usi commerciali [Ohl11].
Ad oggi, ExtJS è uno dei prodotti offerti dalla Sencha Inc. [Sen12c], compagnia nata nel 2010. La stessa Sencha sviluppa ExtGWT, la libreria utilizzata per questa tesi, che verrà illustrato nella sezione 2.4.

I prodotti Sencha sembrano avere effettivamente incontrato un certo successo: la compagnia pubblicizza 300 mila membri registrati alla sua community, e oltre un milione e mezzo di sviluppatori nel mondo.

ExtJS è attualmente alla versione 4, rilasciata nell'aprile 2011 con tre diverse licenze: GPLv3 per coloro che utilizzano la libreria in progetti sempre GLPv3, una licenza commerciale per coloro che sfrutteranno la libreria per applicazioni commerciali, e infine una licenza commerciale OEM per usare ExtJS all'interno di una libreria di sviluppo da ridistribuire commercialmente. Ma passiamo ora ad illustrare cosa è, nello specifico, ExtJS.

ExtJS è, abbiamo detto, una libreria JavaScript. È orientata agli oggetti ed estensibile e non dipende da alcuna libreria esterna. Lo scopo del suo sviluppo è semplificare ai programmatori lo sviluppo di RIA (vedi sezione 2.1), fornendo un'ampia collezione di componenti grafici per interfacce utenti propri dei desktop detti *widget*, come finestre, pulsanti, menu e griglie di dati.
Tali widget non sono solo esteticamente gradevoli: implementano già una serie di funzionalità comuni: rispondono agli eventi, come il passaggio del mouse o la selezione, le finestre supportano il drag and drop e il ridimensionamento, le griglie semplificano il caricamento dinamico dei dati, e così via. Gli elementi del DOM di questi oggetti grafici sono facilmente accessibili, e configurabili a piacere.
Un esempio di widget disponibile è riprodotto nella figura 2.2: si tratta di una finestra di dialogo che presenta tre possibili scelte.
In figura 2.3 è riportato il codice JavaScript/ExtJS per ottenere una simile finestra: come si può osservare è estremamente semplice, in quanto ExtJS si fa carico di tutti i dettagli, come le dimensioni della finestra, la presenza di un'icona che identifichi la tipologia di messaggio visualizzato, la possibilità di chiudere la finestra o spostarla, e così via. Il programmatore deve quindi solo scegliere, per una finestra di dialogo come nell'esempio, l'intestazione e il testo da utilizzare, e le azioni da intraprendere a seconda dell'opzione scelta dall'utente mediante i tre pulsanti.

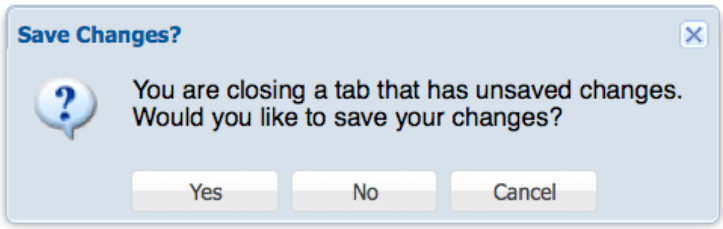

Figura 2.2: Una finestra di dialogo ottenibile con poche righe di codice grazie alla libreria ExtJS

```
Ext.get('mb').on('click', function(e){
  Ext.MessageBox.show({
  title:'Save Changes?',
  msg: 'You are closing a tab that has unsaved changes.<br/>
      Would you like to save your changes?',
  buttons: Ext.MessageBox.YESNOCANCEL,
  fn: showResult,
  animateTarget: 'mb',
  icon: Ext.MessageBox.QUESTION
  });
});
```

Figura 2.3: Codice JavaScript/ExtJS per aprire la finestra di dialogo in figura 2.2

I widget ExtJS sono anche *cross browser*: tutti i browser di maggior diffusione, Internet Explorer, Mozilla Firefox, Apple Safari e Opera, sono supportati dalla libreria.
ExtJS mitiga dunque alcuni degli svantaggi del programmare in AJAX, e, proponendo soluzioni giù pronte per necessità comuni, permette ai programmatori di concentrarsi sulla personalizzazione dei widget o sullo sviluppo di funzionalità più complesse.

Una lista, non esaustiva, dei widget forniti da ExtJS è la seguente:

- campi di testo e textarea, campi numerici, e relativi controlli

- campi e calendari per inserimento di date
- list box e combo box
- radio buttons e checkbox
- griglie di dati
- forms
- menu e barre degli strumenti
- gestori di layout, che permettono di definire la disposizione degli elementi
- finestre, finestre di dialogo, schede (*tabs*)
- alberi di dati

Nonostante i vantaggi ottenibili dall'uso di ExtJS, un'applicazione ExtJS è comunque un'applicazione JavaScript, con i conseguenti svantaggi che ho illustrato nella sezione 2.1. Nella prossima sezione vedremo dunque in cosa consiste la proposta di GWT e quali miglioramenti promette.

2.3 GWT

Google Web Toolkit o GWT [Goo12], è un kit di strumenti di sviluppo per la costruzione e l'ottimizzazione di applicazioni web complesse.
Le RIA a cui ho precedentemente accennato (sezione 2.1) rientrano certamente nella categoria, ma gli usi di GWT non sono limitati alla programmazione di interfacce, bensì includono ogni tipo di funzionalità JavaScript. Creato da Google Inc., è stato per la prima volta annunciato nel maggio del 2006, in occasione della conferenza annuale sulle tecnologie Java *JavaOne*, organizzata da Sun Microsystems [Sun06], e contemporaneamente ne è stata rilasciata la prima versione.
Il framework è distribuito gratuitamente con licenza open source Apache 2.0. Al momento si trova alla versione 2.4, rilasciata lo scorso settembre 2011.
Su Wikia [Wik] si trova una lista di applicazioni web sviluppate attraverso GWT che possono fornire una panoramica sulle sue potenzialità.

Il suo obiettivo dichiarato, come framework di sviluppo, è di facilitare lo sviluppo web rendendo AJAX più accessibile: permettendo al programmatore di non interessarsi a JavaScript, ai dettagli del trasferimento di dati client-server via HTTP, né tanto meno alle caratteristiche dei diversi browser [Voi07].
Il concetto alla base di GWT è di scrivere le applicazioni web in Java, e successivamente compilarle, grazie al compilatore GWT, in JavaScript altamente ottimizzato e *cross browser*.
JavaScript viene quindi messo da parte, a favore di un linguaggio, Java, fortemente tipato, con interfacce ben definite, e che favorisce un approccio modulare. Rispetto a JavaScript, Java rende più semplice gestire grandi progetti, che richiedono di condividere codice, produrre documentazione in modo semplice e veloce, e via di seguito.

I suoi due fondatori, Bruce Johnson e Joel Webber, raccontano di aver osservato il successo ottenuto da servizi quali Google Maps e Gmail, e di non aver potuto non riconoscere quanto gli utenti apprezzassero le applicazioni web, e quanto queste fossero più interattive e avvincenti rispetto a ciò che il web offriva in precedenza.
Se da un lato apprezzavano le caratteristiche di JavaScript che avevano permesso questo risultato, non erano altrettanto entusiasti di come questo si ponesse di fronte a concetti quali astrazione, riutilizzo e modularità, di grande importanza in progetti di ampie dimensioni. Da qui l'idea di un *cross compiler* da Java a JavaScript, un compilatore che permettesse di scrivere codice Java e compilarlo in JavaScript adatto a essere eseguito sui cinque browser più diffusi, coi tre più diffusi sistemi operativi [Joh09].

I vantaggi di questo approccio sono diversi: il livello di astrazione ulteriore fornito da GWT su AJAX permette di alleggerire il carico di lavoro al programmatore: niente più XmlHttpRequest, ma un semplice meccanismo di Remote Procedure Calls, una libreria di componenti dell'interfaccia grafica riutilizzabili, un accesso più semplice agli elementi del DOM.
È poi possibile integrare con GWT JavaScript scritto a mano, usando la JavaScript Native Interface (JSNI) [Gooe].

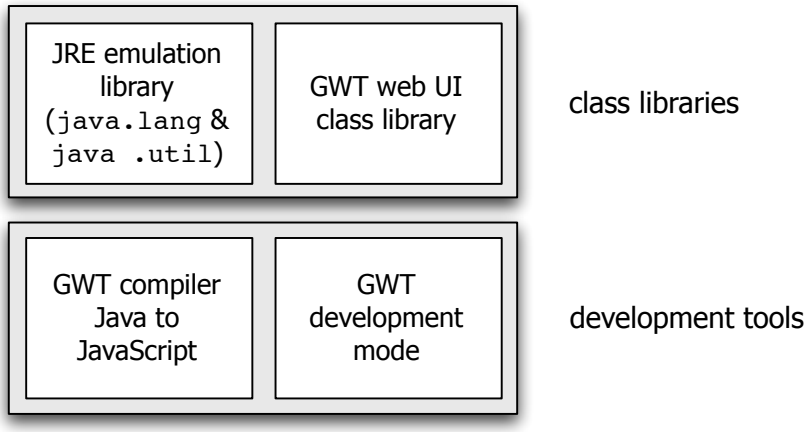

Figura 2.4: Architettura di GWT [Goob].

2.3.1 Architettura di GWT

GWT ha quattro componenti: due librerie e due strumenti di sviluppo, come mostrato in figura 2.4.
Il compilatore è responsabile della traduzione dal linguaggio Java al linguaggio JavaScript, mentre la modalità di *development*, permette di eseguire le applicazioni GWT all'interno della JVM. Approfondirò il compilatore nella sezione 2.3.3, e la modalità di *development* nella sezione 2.3.2.

Per quanto riguarda le due librerie, la JRE emulation library contiene le implementazioni delle classi più comunemente utilizzate della libreria standard di Java, più parte delle classi di **`java.lang`** e **`java.util`**. Maggiori dettagli sulle classi emulate saranno forniti nella sezione 2.3.4.
La libreria GWT web UI contiene la collezione di classe e interfacce utili per la creazione di widget [Gooc]. Tra questi abbiamo

- radio buttons, checkbox e altri pulsanti
- liste, menubar, menu ad albero
- campi di testo e textarea, campi numeri, date picker
- pannelli, finestre e finestre di dialogo, popup

- tabelle e griglie

Nel complesso, si può osservare come tale lista sia analoga a quella, già presentata nella sezione precedente, dei widget forniti da ExtJS. Vedremo nella prossima sezione come ExtGWT fornisca una terza collezione di widget di nuovo analoghi come ruoli, ma in particolare simili a quelli di ExtJS anche come aspetto grafico.

Dettagli ulteriori su GWT saranno forniti nella sezione 4, dove presenterò il prototipo e approfondirò i dettagli della sua implementazione, e, dunque, di GWT.

2.3.2 Development mode e production mode

Le applicazioni GWT possono essere eseguite in due diverse modalità: *development mode* o *production mode*. La prima modalità è quella utilizzata nella fase di sviluppo: l'applicazione è eseguita come Java bytecode all'interno della JVM, ed è possibile dunque eseguire il debug passo passo come in una normale applicazione Java. Nessun JavaScript non è coinvolto.
Nella modalità di "produzione", invece, l'applicazione è eseguita come JavaScript ed HTML, compilati a partire dal codice sorgente Java col compilarore GWT. Questa modalità è quella in cui le applicazioni sono eseguite quando schierate.

Durante la fase di sviluppo dunque, l'esecuzione del codice, la compilazione e il debug avvengono tutti come per una normale applicazione Java, con tutti i vantaggi che ne derivano: un'ampia serie di strumenti è infatti disponibile per il supporto alla programmazione in Java, mentre lo stesso non è vero per JavaScript. Problemi di tipo e *typos* sono risolti a tempo di compilazione, si possono sfruttare le potenzialità del refactoring e il fatto che i pattern orientati agli oggetti rendono il codice più comprensibile e facile da comunicare. Documentazione Javadoc può essere generata velocemente.
Plugin sono inoltre disponibili per facilitare lo sviluppo di applicazioni GWT con le più diffuse IDE per Java: per NetBeans esiste GWT4NB, mentre per Eclipse *Cypal Studio for GWT*, e l'ufficiale *Google Plugin for Eclipse*.

La combinazione Eclipse/Google Plugin è probabilmente la più utilizzata ed efficiente per lo sviluppo di applicazioni GWT: è possibile la creazione di progetti, l'invocazione del compilatore GWT, la creazione di configurazioni per il compilatore, la validazione del codice, il debug passo passo con breakpoints.

2.3.3 Il compilatore GWT

Al momento di schierare l'applicazione, il compilatore GWT effettua la traduzione da Java a JavaScript. Il JavaScript ottenuto è ottimizzato per i diversi browser, e altamente performante: nel suo lavoro il compilatore rimuove codice morto, allinea metodi, ottimizza le stringhe e così via. Il fatto che Java sia un linguaggio molto più rigoroso di JavaScript permette al compilatore di essere in possesso di informazioni aggiuntive, che può sfruttare per eseguire le ottimizzazioni.
Sempre grazie al passaggio da Java a JavaScript, si risolve anche il conflitto tra codice performante e facile da mantenere. Normalmente codice sintetico e orientato alla velocità non corrisponde a codice chiaro e comprensibile, ossia mantenibile. Con GWT è possibile concentrarsi sullo sviluppo di un codice Java che sia facile da mantenere: velocità e performance saranno lasciate a carico del compilatore.
Sempre a tale proposito, il compilatore produce codice compresso, di dimensioni ridotte, e offuscato. Possiamo vedere un esempio di un simile JavaScript prodotto dal compilatore in figura 2.5. È anche possibile richiedere, per ragioni di debug, la produzione di un codice leggibile, come riportato in figura 2.6. Per dare un'idea della differenza, questo secondo codice è molto meno compresso del primo, e può avere dimensioni anche molto più che doppie.
Il compilatore può anche essere istruito a dividere l'applicazione in frammenti JavaScript multipli, che saranno scaricati separatamente dal browser, così da velocizzare il caricamento di grandi applicazioni.

2.3.4 Compatibilità con classi e librerie Java

Ovviamente non qualunque cosa venga scritta in Java è traducibile dal compilatore GWT in JavaScript. Il compilatore richiede di fornire un file xml di configurazione detto modulo, nel quale indicare, tra le altre cose,

```
function Zb(){}
_=_b.prototype=Zb.prototype=new U;_.gC=function ac(){return
    s8b};_.wd=function bc(a){W7b(KTb(a),370).Go(false);Og(this
    .b)};_.cM={240:1,312:1};_.b=0;function dc(){}
function ec(a){this,a;Y.call(this);dc()}
function cc(){}
_=ec.prototype=cc.prototype=new U;_.gC=function fc(){return
    t8b};_.wd=function gc(a){W7b(KTb(a),370).Go(false);Cg()};_
    .cM={240:1,312:1};function ic(){}
function jc(a){this,a;Y.call(this);ic()}
```

Figura 2.5: Frammento di codice JavaScript prodotto dal compilatore GWT con **`-style OBFUSCATED`**

```
_.toString$ = function toString_0(){
  return $getName_1(this.getClass$()) + '@' + toHexString(this
      .hashCode$());
}
;
_.toString = function(){
  return this.toString$();
}
;
_.typeMarker$ = nullMethod;
_.castableTypeMap$ = {};
function $$init_0(){
}
```

Figura 2.6: Frammento di codice JavaScript prodotto dal compilatore GWT con **`-style PRETTY`**

quali packages conterranno codice da tradurre.
Le classi in tali packages potranno attingere alla maggioranza della sintassi e della semantica Java, con alcune differenze, ad esempio si deve fare attenzione nell'usare numeri di tipo `long`: questi sono emulati attraverso una coppia di interi a 32-bit, dunque una massiccia presenza di operazioni tra `long` può effetti negativi sulla performance per via appunto dell'emulazione [Good].
Per quanto riguarda le librerie, solo una minima parte delle classi di Java 2 Standard Edition, e Java 2 Enterprise Edition, sono supportate da GWT. In particolare, lo è buona parte di `java.lang` e `java.util`. La documentazione di GWT fornisce un elenco delle classi e dei metodi supportati [Gooa].

2.3.5 GWT e gli altri framework AJAX

Confrontare GWT agli altri framework AJAX non è un'operazione semplice. Di certo GWT condivide con gli altri framework l'obiettivo di base: permettere di sviluppare applicazioni web dinamiche in modo più facile.
Tuttavia, la sua proposta di compilare Java in JavaScript è unica. Per Bret Taylor, senior *product manager* presso Google, data la crescente complessità dello sviluppo di applicazioni web, assumere un approccio ibrido dal punto di vista dei linguaggi di programmazione come quello proposto da GWT è inevitabile.

Ma le opinioni sono varie. Secondo Charles Kendrick, co-fondatore di *Isomorphic Software*[2], compagnia che si occupa di sviluppo di applicazioni web per il business con tecnologie AJAX, GWT può essere interessante per soggetti diversi dai tipici programmatori AJAX, i quali tipicamente hanno esperienza con PHP e JavaScript, ma non con Java. Chi sia già esperto in programmazione Java, e sia poco desideroso di cimentarsi con JavaScript e tutte le piccole peculiarità dei diversi browser potrebbe trovare GWT una proposta risolutiva, anche se ovviamente GWT non risolve tutte le problematiche dello sviluppo web: è comunque necessario testare i risultati sui diversi browser, fosse anche solo per il fatto che l'interpretazione dei CSS presenta differenze da browser a browser.

[2] `http://www.smartclient.com/`

Dal punto di vista di Kendrick, GWT può essere visto un poco come una reazione al timore che può suscitare AJAX. Dal momento che gli strumenti per lo sviluppo in AJAX sono in fase di miglioramento, e gli sviluppatori di browser sembrano iniziare a prestare maggiore attenzione alle esigenze degli sviluppatori AJAX, esiste il rischio che scegliere di programmare JavaScript in Java sia una scelta che porti solo a essere marginalizzati.
Non è detto che sia così: lo sviluppatore e blogger Ed Burnette [3] ritiene che ci sia anche la possibilità che GWT divenga una minaccia per altri framework, togliendo loro ragione di essere, oppure potrebbe divenire loro complementare, dal momento che grazie alla JavaScript Native Interface è possibile effettuare chiamate da GWT verso altre librerie [Got07].

2.4 ExtGWT

ExtGWT (nota anche come GXT) [Sen12a] è una libreria per GWT, sviluppata sempre da Sencha Inc. come ExtJS (vedi sezione 2.2), e rilasciata nel luglio 2008[4], che fornisce anch'essa un'ampia collezione di widget, templates e layout. Tutti i componenti sono compatibili coi browser più diffusi, Explorer, Firefox, Safari, Chrome e Opera, e sono conformi alla specifica del W3C sull'accessibilità WAI-ARIA[5], e alla cosiddetta "Sezione 508"[6], una serie di linee guida sull'accessibilità pubblicate dal governo americano.

Essenzialmente ExtGWT è ExtJS in ambiente GWT, almeno dal punto di vista grafico. ExtGWT fornisce sostanzialmente gli stessi widget forniti da ExtJS: gli stessi nomi dei campi o dei metodi sono uniformi tra le due librerie.
La scelta di utilizzare ExtGWT non è stata mia, bensì era la richiesta fondamentale alla base della proposta di tesi di Wincor Nixdorf: sviluppare *mediante ExtGWT* un web desktop graficamente analogo a quello già in loro possesso di NRGine (vedi sezione 1.2), sviluppato con ExtJS. Tuttavia, dato

[3] `http://www.zdnet.com/blog/burnette`
[4] `http://www.sencha.com/blog/ext-gwt-v10-released`
[5] `http://www.w3.org/TR/wai-aria/`
[6] `http://www.access-board.gov/sec508/guide/1194.22.htm`

```
buttonBar.add(new Button("Click me",
            new SelectionListener<ButtonEvent>() {
  public void componentSelected(ButtonEvent ce) {
    MessageBox box = new MessageBox();
    box.setTitle("Save Changes?");
    box.setMessage("You are closing a tab that has unsaved
        changes. Would you like to save your changes?");
    box.setButtons(MessageBox.YESNOCANCEL);
    box.setIcon(MessageBox.QUESTION);
    box.addCallback(l);
    box.show();
  }
}));
```

Figura 2.7: Codice GWT/ExtGWT per aprire la finestra di dialogo in figura 2.2. Si può osservare la somiglianza col codice ExtJS per lo stesso scopo, riportato in figura 2.3

appunto il grande parallelismo, in quanto a interfacce utente e a funzionalità, tra le due librerie, sarebbe stato difficile fare una scelta differente.
Le motivazioni che hanno portato Wincor Nixdorf a orientarsi verso la valutazione della sostituzione del web desktop esistente con uno nuovo sebbene analogo, sono state affini a quelle che abbiamo già visto nella sezione 2.3 essere le ragioni alla base dello sviluppo da parte di Google di GWT stesso: la complessità di una grande applicazione scritta in AJAX, con la conseguente difficoltà a mantenerla e ad estenderla.

Le somiglianze tra ExtJS e ExtGWT sono solo nella forma e non nella sostanza: la prima, come abbiamo visto, è JavaScript, mentre la seconda è una libreria per Google Web Toolkit, dunque Java: oltre all'estetica delle interfacce utenti, per il programmatore si tratta di un universo completamente differente.
L'esistenza sul mercato di ExtGWT permette a Wincor Nixdorf un cambio di tecnologia per il loro web desktop, senza un cambio nell'aspetto e nel *look and feel* delle interfacce.

Al momento ExtGWT si trova alla versione 2.2.5, rilasciata nel settembre 2011. Una versione 3 è stata annunciata, e ne è stata distribuita la sua terza *beta release* lo scorso 14 febbraio 2012. Nessuna data è stata annunciata per il rilascio della versione stabile.

Capitolo 3

Progettazione

Dopo un primo periodo trascorso documentandomi sulle tecnologie che avrei utilizzato, ho rivolto la mia attenzione alla progettazione del prototipo che avrei in un secondo momento sviluppato.
Questo capitolo riassume le informazioni raccolte durante la fase di progettazione e le scelte effettuate.
Nella sezione 3.1 riformulerò l'obiettivo del progetto di tesi, illustrando, seppure in modo generale, i requisiti del prototipo. Nella sezione 3.2 illustrerò l'architettura proposta, in termini di macro componenti.
Proseguirò poi con lo spiegare le scelte progettuali relative alla gestione dei guasti, alla performance e alla gestione degli errori, rispettivamente nella sezione 3.3, 3.4 e 3.5. Terminerò indicando, nella sezione 3.6 quali scelte ho preso sotto l'aspetto della sicurezza, e, nella sezione 3.7, come ho deciso di introdurre un meccanismo di internazionalizzazione.

3.1 Requisiti

Il progetto di tesi richiedeva lo sviluppo di quattro interfacce grafiche: una finestra per la procedura di login, un web desktop con un menu, attivabile dal classico pulsante start in basso a sinistra sullo stile a cui Windows ci ha abituato, una finestra con griglia e una finestra con forms. Bozze per tali interfacce sono raffigurate in figura 3.1.
Queste interfacce non saranno progettate da zero bensì ricalcheranno in modo

abbastanza preciso le interfacce del web desktop già in uso, con solo alcune modifiche che mi sono state esplicitamente richieste per migliorare l'usabilità.

Solo le interfacce avrebbero dovuto essere prodotte: per tutte le funzionalità ci si sarebbe dovuti rivolgere ai livelli *business* e *logic* di NRGine, illustrati in sezione 1.2, che già le fornivano. Ovviamente riscrivere determinate parti era ammissibile, ma l'idea era riutilizzare quanto già disponibile e concentrarsi sulle interfacce.

I requisiti per ogni singola interfaccia grafica non saranno qui riportati, per due ragioni: in primo luogo, tali requisiti sono stati formulati in modo informale, come sottoinsieme delle caratteristiche e delle funzionalità delle interfacce preesistenti, e in secondo luogo per ragioni di privacy di Wincor Nixdorf.
Una descrizione generale delle interfacce e delle loro funzionalità sarà fornita nella sezione 4.5.

La caratteristica essenziale che il prototipo avrebbe dovuto avere, oltre ovviamente all'aderenza ai requisiti appena citati, è la mantenibilità.
Per ottenerla, come linee guida generali si è stabilito di:

- limitare le connessioni tra i diversi componenti del sistema, e anche tra le diverse classi, di modo da ridurre le dipendenze reciproche, e far sì che una modifica incidesse il meno possibile su altri componenti/classi.
- utilizzare noti schemi di design (*design pattern*) e mantenere la struttura del progetto semplice da comprendere.
- documentare estensivamente il progetto internamente, anche mediante lo strumento del Javadoc[1].

Tali indicazioni trovano applicazione oltre che nella struttura dell'architettura, che andrò ora a descrivere, anche nell'organizzazione del codice vero e proprio, la quale sarà invece affrontata nel capitolo 4.

[1] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

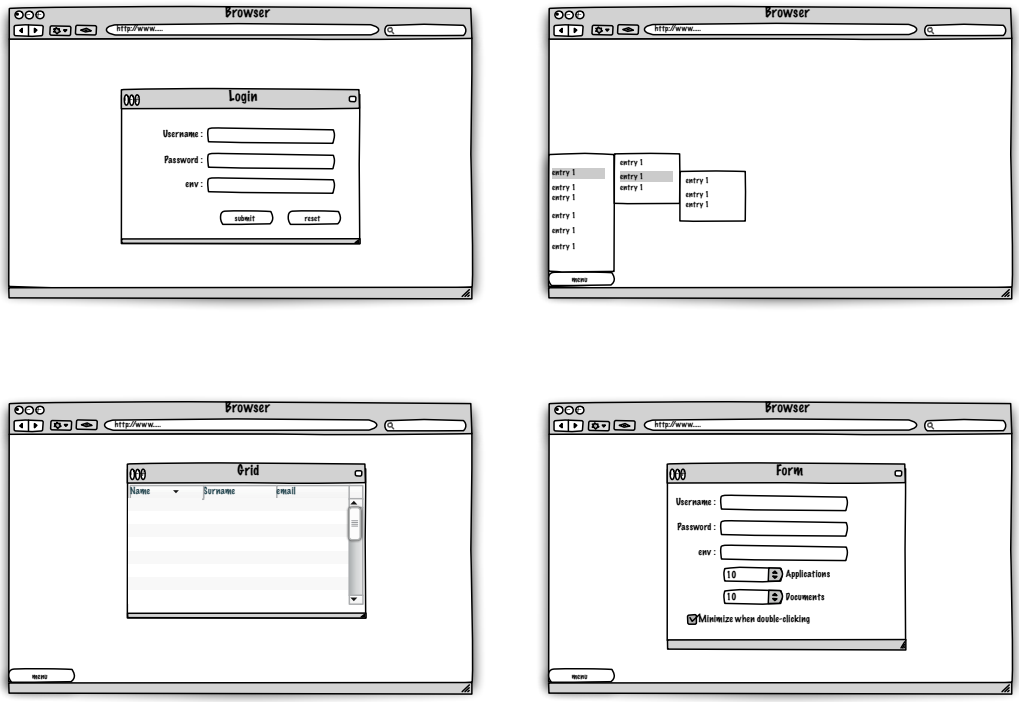

Figura 3.1: Bozza dell'interfaccia grafica. La prima immagine in alto a sinistra mostra lo schizzo della schermata con la finestra di login, mentre l'immagine in alto a destra mostra il web desktop con il menu espanso. Nella riga più in basso, a sinistra abbiamo l'abbozzo di una finestra con griglia (dentro il web desktop), mentre a destra di una finestra con form.

3.2 Architettura

GWT consiglia un certo tipo di struttura per i progetti sviluppati con esso, la quale verrà approfondita nella sezione 4.3, e fornisce un meccanismo di comunicazione basato su *Remote Procedure Call* asincrone (RPC), di cui tratterò più nel dettaglio nella sezione 4.4.1, che permette di comunicare tra client e server attraverso lo scambio di oggetti Java.
Considerando ciò, ho deciso di organizzare il progetto secondo la struttura illustrata in figura 3.2.

Il progetto si comporrà di due livelli: un livello che possiamo chiamare *service*, e un livello *presentation* vero e proprio. Il livello presentazionale gestirà l'interazione con l'utente finale, e costituirà il contenitore per tutte le interfacce grafiche e la logica relativa alle loro interazioni reciproche.
Il livello *service* avrà invece il ruolo di intermediario tra il livello presentazionale e i due livelli *business* e *logic* di NRGine, fornendo al primo i dati applicativi da presentare in una formulazione a lui comprensibile, e l'accesso a determinati servizi.
Il livello *service* interagirà sia col livello business che col livello logico, in modo analogo a come accade col web desktop precedente, e come è stato illustrato nella sezione 1.2. Tale scelta è stata motivata dalla necessità di non effettuare alcuna modifica al framework NRGine in livelli diversi da quello presentazionale, e allo stesso tempo dalla decisione di non replicare eventuali funzionalità già fornite.
Questi vincoli purtroppo hanno fatto sì che livello logico e *service* siano sì separati formalmente, ma vi siano tra i due confini labili per quanto riguarda la divisione delle competenze, dal momento che alcuni aspetti propri della logica si troveranno anche nel livello *service*. Comunque, auspicabilmente una riorganizzazione successiva del codice, sia del nuovo prototipo che di NRGine, potrà rendere l'architettura più coerente col modello che si è stabilito.

In figura 3.3 troviamo una seconda rappresentazione dell'architettura, nella quale sono evidenziati i meccanismi di comunicazione.
Il livello presentazionale effettuerà richieste, per i dati da visualizzare e per

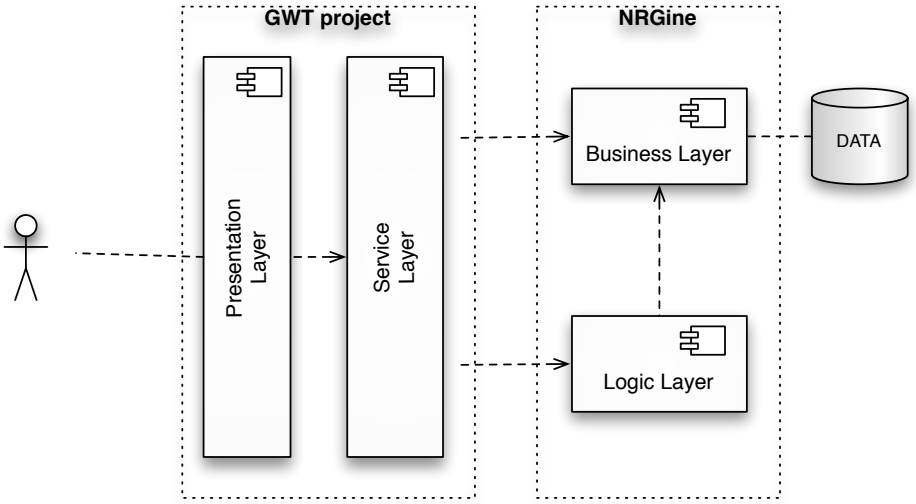

Figura 3.2: Architettura del prototipo. Il progetto sviluppato è costituito di due livelli, *presentation*, che contiene le interfacce grafiche, e *service*, che ha il ruolo di intermediario tra il livello *presentation* e il framework NRGine originale.

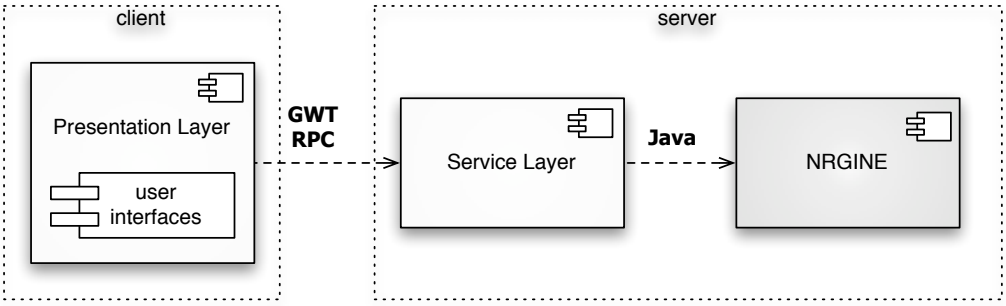

Figura 3.3: Architettura del prototipo. Il livello *presentation*, che esegue sul browser dell'utente finale, comunica con il *service layer* attraverso il meccanismo di RPC asincrone di GWT. Il *service layer* comunica invece con framework NRGine attraverso semplici invocazioni di metodi Java.

l'esecuzione di operazioni, come ad esempio il login, nella forma di invocazioni di metodi del livello *service* mediante il meccanismo RPC di GWT. Infatti, il livello presentazionale, dal momento che consiste essenzialmente delle interfacce grafiche compilate in JavaScript, in un certo senso si può dire "risiederà" nel browser dell'utente finale, dunque lato client.
Il meccanismo di RPC prevede che in un componente del livello presentazionale si trovino le interfacce dei servizi, le cui corrispondenti implementazioni costituiranno il livello *service*.
Il livello service invece si troverà sul server insieme col resto del framework NRGine, che potrà interpellare mediante semplici invocazioni di metodi Java.

3.3 Gestione dei guasti

Come abbiamo appena visto, la comunicazione tra il livello *presentation*, che sostanzialmente esegue sul browser dell'utente finale, e il resto del framework, avverrà mediante chiamate RPC asincrone, un meccanismo di comunicazione a livello applicazione che garantisce una certa affidabilità.

Eventuali guasti di comunicazione, intesi come impossibilità a ottenere risposta dai servizi presso il server, comporteranno l'impossibilità di completare le operazioni richieste.
Ogni chiamata a servizi non sarà bloccante, in quanto appunto il meccanismo di RPC di GWT è asincrono, dunque le interfacce grafiche saranno comunque almeno in parte utilizzabili anche in caso di problemi di comunicazione.

3.4 Performance e gestione delle risorse

Nessun tipo di analisi è stata svolta per quanto riguarda

- la capacità di server e servizi di rispondere a più richieste allo stesso tempo.
- la capacità di server e servizi di rispondere alle richieste entro determinati tempi massimi.

- la scalabilità del sistema.
- la gestione delle risorse, né lato client né lato server.

3.5 Gestione degli errori

Gli errori possibili internamente al sistema potranno avvenire nel *service layer* o nel *presentation layer*.
Gli errori nel *service layer* potranno essere causati da impossibilità di reperire dati, o di portare a termine una determinata operazione. In entrambi i casi, un errore di questo tipo non sarà in alcun modo risolvibile con un'azione da parte dell'utente finale. Tali errori saranno registrati attraverso il meccanismo di logging e notificati al *presentation layer* attraverso un'opportuna eccezione. Sulla base dell'eccezione ricevuta, il *presentation layer* notificherà l'errore avvenuto all'utente finale con un messaggio ragionevolmente semplice e generico. Tale messaggio sarà provvisoriamente mostrato nella forma di una finestra di alert.

Nel *presentation layer* gli errori potranno avvenire come conseguenza di errori nel *service layer*: in questo caso saranno gestiti come appena illustrato. Il meccanismo di RPC di GWT fa anche sì che il fallimento di un'operazione, come ad esempio il fallimento della procedura di login per via di credenziali non valide, sia restituito dal *service layer* al *presentation layer* nella forma di un'eccezione. Eccezioni ad hoc saranno create per gestire errori noti e inerenti alla logica del programma, come appunto il caso di autenticazione con credenziali non corrette.
Tali errori noti a priori verranno notificati all'utente in modi specifici e diversi a seconda del tipo di errore e dell'interfaccia che ha chiamato il servizio che lo ha generato.

Per quanto riguarda gli errori di comunicazione, anche questi seguiranno una delle due strade precedenti: potranno essere notificati tramite alert, quando inaspettati, oppure in un modo proprio dell'interfaccia grafica quando in qualche modo previsti. La differenza tra errori inaspettati e previsti è un

concetto blando che si raffinerà durante la costruzione del prototipo. In linea di massima l'idea alla base sarebbe che determinate operazioni potranno essere considerate come momenti di controllo della comunicazione, mentre durante altre operazioni si presupporrà che la comunicazione avvenga senza intoppi.
Per fare un esempio concreto, al momento della procedura di login si prenderà in considerazione di non ricevere risposta dal server: in questo caso l'errore sarà in qualche modo "previsto" e verrà notificato internamente alla finestra. Tuttavia, se la procedura di login dovesse andare a buon fine, sarà ragionevole pensare che il server risponda correttamente anche alla successiva richiesta di dati per costruire il web desktop: un errore di comunicazione in questa fase sarà inatteso, e verrà notificato con un alert.

Per chiamate ai servizi corrispondenti alle operazioni cosiddette di controllo saranno anche specificati dei timeout, nell'idea che se la risposta dovesse tardare troppo a giungere, sarebbe più opportuno non proseguire affatto col fornire la funzionalità.

La notifica di errore via alert sarà fornita da un metodo statico comune a tutte le classi, per dare un'idea di gestione centrale degli errori. Tuttavia, tale meccanismo semplicistico sarà da considerarsi assolutamente provvisorio.
L'obiettivo, in un secondo momento, sarà eliminare via via le notifiche di errore via alert, per sostituirle con notifiche di errore fornite dalle diverse interfacce grafiche in un modo a loro proprio.

3.6 Sicurezza

L'aspetto della sicurezza non è stato preso in considerazione, dunque solo piccoli accorgimenti saranno presi in tal senso.
Cercherò di fare una prima valutazione sulla validità dell'input dell'utente nel momento in cui viene fornito, ossia nel *presentation layer*, onde evitare di

riportare input non validi al *service layer*, e mi atterrò alle linee guida sulla sicurezza in GWT[2].

3.7 Internazionalizzazione

Al fine di tenere il più possibile aspetti per così dire configurabili, come ad esempio il testo per le intestazioni delle finestre, separati dalle classi Java vere e proprie, si è utilizzato il meccanismo dell'internazionalizzazione statica fornito da GWT, che verrà approfondito nella sezione 4.4.4. Il suo uso ha come gradito effetto collaterale quello di predisporre il prototipo per un'eventuale traduzione in una diversa lingua.

[2] `http://code.google.com/intl/it/webtoolkit/doc/latest/DevGuideSecurity.html`

Capitolo 4

Implementazione

In questo capitolo descriverò i dettagli dell'implementazione del prototipo. Nella sezione 4.1 illustrerò il modello di processo che si è utilizzato per lo sviluppo. Nella sezione 4.2 elencherò gli strumenti che si sono utilizzati per lo sviluppo, per poi passare, nella sezione 4.3, a descrivere la struttura del codice vera e propria: organizzazione dei pacchetti, descrizione dei più importanti tra essi, e di alcune singole classi.
Nella sezione 4.4 approfondirò alcuni dettagli dell'implementazione, e nella sezione 4.5 mostrerò le schermate delle interfacce grafiche e riassumerò alcune delle loro caratteristiche e funzionalità principali.

4.1 Modello di processo

Dal momento che il prototipo era fortemente incentrato sulle quattro interfacce a cui si è accennato in precedenza (vedi figura 3.1), è stata una scelta molto naturale decidere in favore di uno sviluppo per fasi, in ognuna delle quali si è seguito un modello di processo classico, a cascata [Pre05]: raccolta dei requisiti per una specifica interfaccia con il dipendente di Wincor Nixdorf che mi ha seguita durante questa tesi, progettazione e modellazione dell'interfaccia, costruzione e test, e infine presentazione del risultato.
Le fasi sono state tre: la prima si è concentrata su una schermata con una finestra di dialogo per il login, la seconda sulla creazione di un pannello che riempisse tutto lo spazio del browser e somigliasse a un vero desktop, con un

menu attivabile dal classico pulsante start in basso a sinistra sullo stile a cui Windows ci ha abituato, e infine la terza fase si è rivolta a un prototipo di finestra con un griglia.
Una quarta fase è tuttora in corso mentre scrivo questa relazione, e si tratta dell'aggiunta di ulteriori funzionalità alla finestra con griglia. Un altro tipo di finestra contenente un form era in programma inizialmente, e avrebbe quindi costituito una quinta fase, ma, dal momento che la costruzione della finestra con griglia si è dilungata più del previsto, si è concordato di tralasciare tale finestra.

4.2 Strumenti utilizzati

L'applicazione è stata sviluppata con l'IDE Eclipse, la meglio integrata con GWT: come già accennato nella sezione 2.3 Google ha, contestualmente a GWT, sviluppato anche un plugin per Eclipse che permette la compilazione e il debug delle applicazioni in modo facile e veloce. Il testing dell'applicazione è poi avvenuto schierandola su un web server Tomcat.
Per il controllo del versionamento si è utilizzato inizialmente il sistema di controllo distribuito Mercurial[1], e successivamente si è passati a CVS, il sistema utilizzato da Wincor Nixdorf. Il secondo è supportato da Eclipse senza bisogno di plugin aggiuntivi, mentre per il primo si è utilizzato il plugin **`HgEclipse`**[2].

La filosofia GWT di "ottimizzare aggressivamente la performance" e di "non fare a runtime ciò che puoi fare durante la compilazione"[3] significa molte cose, codice più ottimizzato, più veloce, ma anche lunghi tempi di compilazione: per la compilazione del nostro prototipo i tempi andavano da un paio di minuti ad anche sette o otto minuti.

[1] `http://mercurial.selenic.com/`
[2] `http://www.javaforge.com/project/HGE`
[3] `http://code.google.com/intl/it/webtoolkit/makinggwtbetter.html`

Per rendere più veloce la compilazione e il deploy si è fatto uso di alcuni *build files* Ant[4], e delle configurazioni messe a disposizione del compilatore[5].

Nessun tipo di documentazione addizionale è stata prodotta oltre a questa tesi e al Javadoc[6], che ho cercato di rendere completo ed esaustivo.

Ho scelto di utilizzare la libreria **`gwt-log`** sviluppata da Fred Sauer e rilasciata con licenza Apache 2.0 [Sau12], per il logging lato client. Tale libreria fornisce una serie di funzionalità molto utili, tra cui un pannello in cui registra i log direttamente all'interno dell'applicazione sviluppata. Tale pannello si può vedere nella figura 4.7. La libreria è veloce da introdurre, e permette, come ogni framework di logging che si rispetti, di disattivare il logging per uno o più livelli di gravità, in modo semplice.
Inizialmente si è scelto di usare **`gwt-log`** per via delle limitate possibilità che pareva offrire GWT sotto questo aspetto: in realtà era la documentazione a essere carente da questo punto di vista, poiché la maggioranza delle funzionalità di **`gwt-log`** sono da qualche versione disponibili anche in GWT. Dunque, tra le necessità future per questo prototipo, vi è anche quella di eliminare la dipendenza da tale libreria.

4.3 Organizzazione del codice

Per le applicazioni GWT esiste una struttura consigliata, descritta nella documentazione ufficiale.

Le risorse statiche si trovano nella directory **`war`**, come richiesto da GWT. Nella sottocartella **`war/WEB-INF`** si trova il *deployment descriptor*[7] **`web.xml`**. Si tratta di un file necessario alle applicazioni Java web che permette di

[4] `http://ant.apache.org/`

[5] `http://code.google.com/intl/it/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html#DevGuideCompilerOptions`

[6] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

[7] `http://code.google.com/intl/it/appengine/docs/java/config/webxml.html`

specificare come gli URL sono mappati nelle servlet, quali URL richiedono autenticazione, e informazioni simili.

Un progetto GWT deve poi possedere un file xml di configurazione, detto modulo, nel quale si specifica tutto quello che serve al progetto, principalmente la classe che verrà chiamata al momento del caricamento del modulo, e i package che contengono il codice che deve essere compilato in JavaScript. Può essere posizionato in qualsiasi package nel *classpath*, ma è buona norma posizionarlo nel package principale: nel nostro caso si trova direttamente nella directory **`src`**.

Il modulo specifica:

- i moduli che eredita. Si usa
  `<inherits name="logical-module-name"/>`
- il nome della classe che costituisce il punto di ingresso, o **`EntryPoint`** dell'applicazione, che vedremo tra breve, con
  `<entry-point class="classname"/>`
- un eventuale path ulteriore dove si possono trovare i sorgenti da compilare in JavaScript (oltre alla directory **`client`**),
  `<source path="path"/>`
- un eventuale path ulteriore per le risorse pubbliche,
  `<public path="path"/>`
- altre informazioni quali regole di binding

Le linee guida prevedono inoltre che il codice lato client sia separato dal codice lato server (se presente) in quanto appunto il primo andrà compilato in JavaScript, diversamente dal secondo.

Nel progetto implementato dentro **`src`** abbiamo dunque tre directory:

- **`client`** contiene il codice Java che andrà compilato in JavaScript e costituirà le interfacce grafiche vere e proprie. In questa directory si trova la classe "di ingresso": si tratta di una classe che deve implementare

l'interfaccia **com.google.gwt.core.client.EntryPoint** e in particolare il metodo **onModuleLoad**, che viene chiamato quando il modulo viene caricato.

- **server** contiene il codice lato server, essenzialmente le implementazioni dei servizi RPC
- **shared** contiene il codice dei *data transfer objects* e delle **Exception**. Queste classi sono utilizzate, come il nome del package suggerisce, sia dal client che dal server, e saranno anch'esse tradotte in JavaScript.

La figura 4.1 fornisce una panoramica sui package che compongono il progetto, e le loro relazioni reciproche. Scenderò ora maggiormente nel dettaglio e approfondirò il contenuto dei tre macropackage.

4.3.1 Client

Lato client abbiamo la classe che implementa l'interfaccia **EntryPoint**, e i seguenti packages:

- **service** contiene le interfacce sincrone e asincrone per i servizi, necessarie per utilizzare il meccanismo di RPC messo a disposizione da GWT. Dettagli su tale meccanismo saranno forniti nella sezione 4.4.1.
- **eventsmanager** contiene le classi che si occupano di gestire gli eventi e effettuare il passaggio del controllo tra i diversi elementi dell'interfaccia grafica. Dettagli sulla gestione degli eventi saranno forniti nella sezione 4.4.3.
- **ui** contiene le finestre e il desktop veri e propri. Tale package contiene un numero ridotto di classi, essendo questo un prototipo, ma in una versione del web desktop NRGine definitiva tale numero salirebbe e la previsione sarebbe di dividere il package in diversi sottopackages.
- **Constants** contiene le interfacce per accedere ai file di proprietà. Una spiegazione del funzionamento di tali interfacce verrà fornita nella sezione 4.4.4.

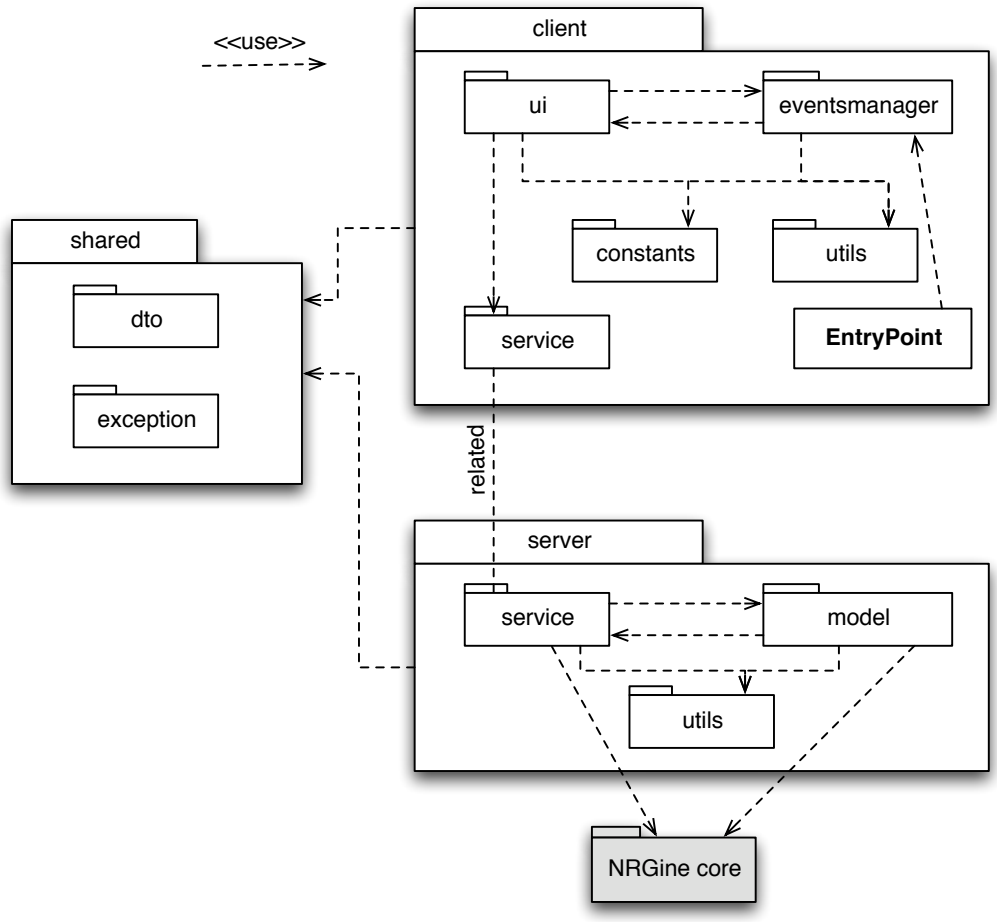

Figura 4.1: Panoramica sui packages che compongono il prototipo, e le loro relazioni di dipendenza. Come si può osservare le classi del package **`shared`** sono utilizzate da ambedue i package **`client`** e **`server`**, dal momento che contengono appunto oggetti di trasferimento. Al codice del framework NRGine si fa riferimento solo nelle classi del package **`server`**.

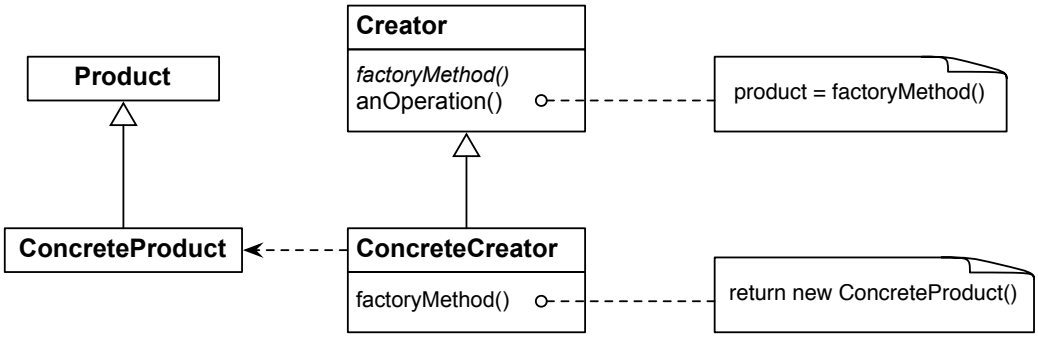

Figura 4.2: Schema del *factory method*.

- **`utils`** contiene classi di ausilio varie, come metodi per il *logging*.

La figura 4.3 riporta le principali classi che fanno parte del package **`client.ui`**, quello di maggior peso. Ognuna di tali classi corrisponde a un elemento dell'interfaccia grafica: **`LoginWindow`** alla finestra di login, **`CustomDesktop`** al nostro desktop, **`CustomWindow`** a una generica finestra che si apre a partire da una elemento del menu del desktop, **`CustomGrid`** a un layout che contiene una griglia, e **`UsersWindow`** a una specifica finestra che visualizzerà dei dati specifici, in questo caso i dati relativi a contenuti di tipo utente.
Le classi di questo package non comunicano molto tra loro: la maggioranza delle modifiche all'interfaccia grafica è mediato dalle classi del package **`eventsmanager`**.
Le classi di questo package sono dunque dei *widget* veri e propri: ognuna di loro, infatti, è stata sviluppata come estensione di una classe della libreria ExtGWT.

Non è riportato nel diagramma delle classi in figura 4.3, ma la creazione delle finestre nel nostro web desktop avviene attraverso una *factory*. La *factory* è l'elemento centrale dell'omonimo *design pattern*, *factory method*.
Tale *pattern* è uno schema che permette di definire un'interfaccia per la creazione di un oggetto, delegando alle sottoclassi la scelta di quale classe concreta istanziare. Il modello elementare del factory method è illustrato in figura 4.2 [GHJV09]. Una rappresentazione della sua applicazione nel prototipo si trova invece in figura 4.12.

Gli elementi del menu del web desktop contengono le informazioni necessarie per identificare un certo tipo di finestra con determinati dati.
L'evento di selezione su un elemento del menu include queste informazioni, e viene inoltrato al gestore degli eventi, il quale a sua volta le fornisce ad una **`CustomWindowFactory`**. Tale factory è in grado di scegliere quale finestra creare: nel nostro prototipo la sola opzione è **`UsersWindow`** ma chiaramente la previsione è di avere una collezione di finestre diverse.
`UsersWindow` specifica quali dati vuole visualizzare, e come: in questo caso mediante un *layout* di tipo griglia.
La finestra che sarà poi restituita e inserita nel **`CustomDesktop`** sarà di tipo **`CustomWindow`**. I dettagli di questo meccanismo saranno approfonditi nella sezione 4.4.3.
Le diverse **`CustomWindow`** saranno accomunate da menu simili, e altre caratteristiche comuni.

4.3.2 Server

Lato server abbiamo la seguente suddivisione:

- **`service`** contiene le implementazioni dei servizi; qui è dove la maggioranza del lavoro viene svolto.
- **`model`** contiene oggetti utili a semplificare le classi in **`service`**: alcune informazioni restituite dalle funzioni del core di NRGine vengono immediatamente incapsulate in oggetti di più comoda gestione appartenenti a questa classe.
- **`utils`** contiene classi di ausilio.

Non approfondirò ulteriormente le classi che fanno parte di questo package, dal momento che trattano dell'interazione del prototipo con NRGine, e dunque di sue caratteristiche specifiche.

4.3.3 Shared

Infine, il package **`shared`**, come già annunciato, contiene classi comuni a server e client: i *data transfer objects*, gli oggetti che client e server si scam-

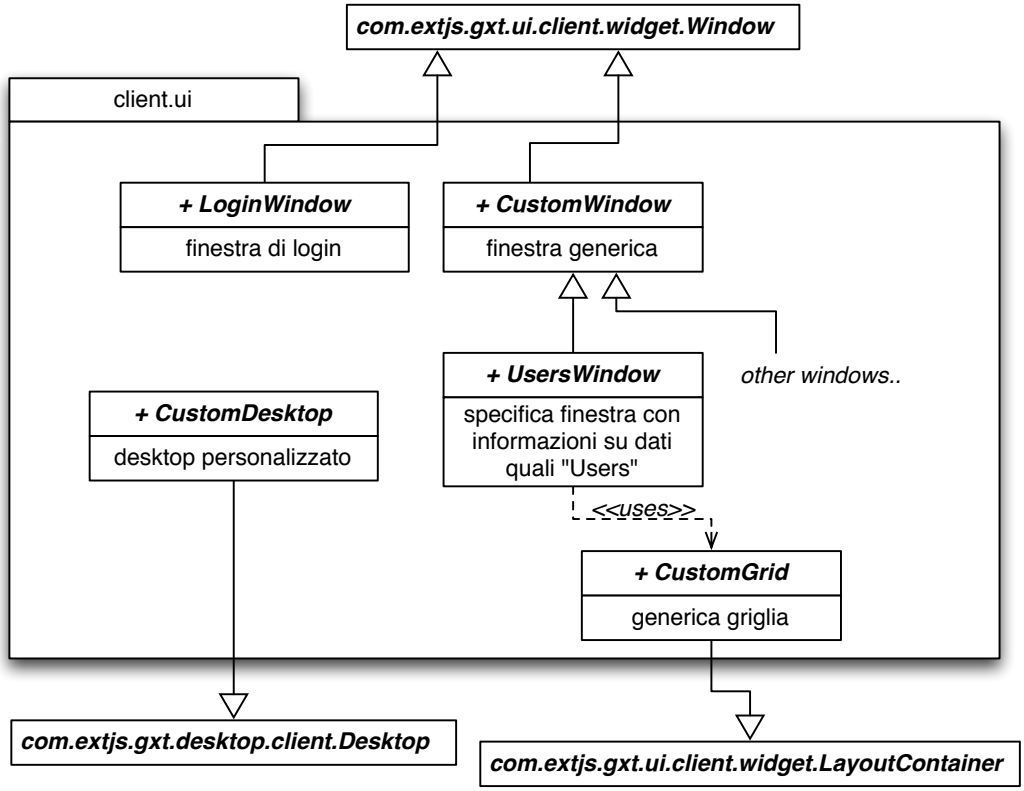

Figura 4.3: Principali classi del package **`client.ui`**. Le classi corrispondono a elementi dell'interfaccia grafica, e dunque a widget

biano, e le eccezioni generate dal server ad uso del client.
Come si può osservare dalle relazioni di dipendenza ho cercato di mantenere il codice il più disaccoppiato possibile. In modo particolare il core di NRGine comunica solo con i servizi.
Nessun oggetto proprio del core viene mai passato lato client, ma solo specifici, e ragionevolmente astratti, *data transfer objects*. Questa scelta è stata in parte inevitabile: gli oggetti passati al client devono essere tradotti in JavaScript. Devono dunque trovarsi un package che il compilatore è istruito a compilare, e devono rispettare i vincoli propri del codice traducibile, ossia utilizzare solo la sintassi base e le librerie ammesse. Inoltre, devono essere correttamente serializzabili e deserializzabili dal client secondo quanto vedremo nella sezione 4.4.2.
Tuttavia, il *decoupling* porta anche benefici in sé, anche in considerazione del fatto che alcuni oggetti del core di NRGine esistono come retaggi di versioni precedenti, e se ne prevedono modifiche anche sostanziali, ma anche in considerazione del fatto che . In questo modo, eventuali cambiamenti non dovrebbero avere impatto sul codice delle interfacce grafiche, ma solo sul modo in cui i servizi creano gli oggetti che restituiscono verso il client.

4.4 Dettagli implementativi

In questa sezione approfondirò alcuni dettagli dell'implementazione. Illustrerò il meccanismo di RPC asincrone fornito da GWT che permette la comunicazione tra client e server, e approfondirò quali oggetti possono essere serializzati correttamente per GWT, e possono essere quindi essere trasferiti tra client e server.
Illustrerò poi il modello *model-view-controller* che si è utilizzato, e come si sono parametrizzate le stringhe di testo.

4.4.1 Comunicazione con un server

GWT permette di comunicare con un server in modi diversi: si può interagire con una servlet Java usando il suo meccanismo di *Remote Procedure Calls*

(RPC), o possono usare le classi del package **`com.google.gwt.http.client`** per comunicare via richieste HTTP.

Nelle situazioni in cui è possibile eseguire Java sul server, le RPC sono il modo più comodo di comunicare. RPC permette la chiamata a un metodo remoto situato sul server, in modo non dissimile a come si effettuerebbe una chiamata locale. In altre parole, il client chiama il servizio passandogli istanze di oggetti Java come parametri, e il server risponde in modo analogo.
Nel nostro prototipo la comunicazione tra client e server è stata effettuata utilizzando tale meccanismo, che ora quindi illustrerò.

Tre elementi sono coinvolti nell'invocazione di servizi del server: il metodo messo a disposizione dal server, il codice del client che invoca il servizio, e gli oggetti Java che costituiranno il contenuto della comunicazione tra i due. Tali oggetti devono poter essere serializzati correttamente sia dal client che dal server: maggiori dettagli sulla serializzazione in GWT si trovano nella sezione 4.4.2.

Nel modello RPC classico, il client ha accesso all'interfaccia di un servizio remoto, la cui implementazione si trova sul server. Quando desidera accedere al servizio chiama i metodi dell'interfaccia, e il meccanismo sottostante si occupa di interrogare il server.
In GWT la situazione è leggermente più complessa poiché il meccanismo di RPC messo a disposizione è totalmente asincrono. Fornire un meccanismo sincrono valido non sarebbe stato possibile per via delle caratteristiche di JavaScript, linguaggio in cui viene eseguito il codice lato client, e sarebbe stato in completo contrasto con l'idea di sviluppo di applicazioni web interattive che costituisce l'obiettivo di GWT.
Dunque, avremo due interfacce, una sincrona e una asincrona, e un'implementazione dell'interfaccia sincrona, come si può vedere dallo schema in figura 4.4.
Le due interfacce risiedono sul client, l'implementazione sul server.

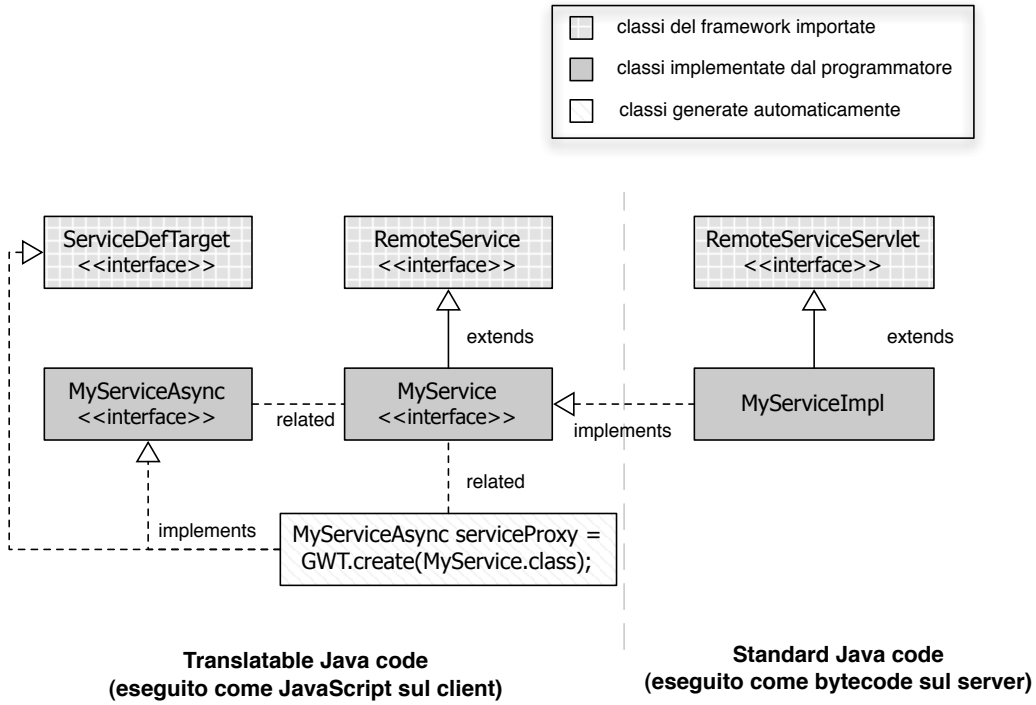

Figura 4.4: Componenti del meccanismo RPC di GWT. Le tre interfacce nella riga più in alto fanno parte di GWT. Le due interfacce e la classe nella riga centrale sono quelle che il programmatore dovrà scrivere. **`MyServiceImpl`** implementerà l'interfaccia **`MyService`**, mentre l'interfaccia **`MyServiceAsync`** non avrà implementazione ma costituirà solo la controparte asincrona di **`MyService`**. Il client, quando desidererà accedere al servizio, farà riferimento all'interfaccia asincrona. GWT si occuperà di creare automaticamente una classe proxy a partire dalle due interfacce.

L'interfaccia sincrona **MyService** estenderà l'interfaccia **com.google.gwt.user.client.rpc.RemoteService**.
L'implementazione del nostro servizio oltre ad esso implementerà l'interfaccia **RemoteServiceServlet**, si troverà sul server e non sarà mai tradotta in JavaScript, il che significa che sarà libera dai vincoli sulle classi utilizzabili che valgono invece per il codice lato client. In ultima analisi tale implementazione è essenzialmente una servlet, con la differenza che invece che estendere **HttpServlet** estenderà **RemoteServiceServlet**, la quale gestisce automaticamente la serializzazione dei dati che vengono trasmessi, e ovviamente invoca il metodo corretto.

L'interfaccia asincrona dovrà avere, per convenzione, lo stesso nome dell'interfaccia sincrona, con l'aggiunta di *Async*, ed esporrà gli stessi metodi, ma ogni metodo avrà un parametro in più rispetto al metodo corrispondente nell'interfaccia sincrona, un oggetto di tipo **com.google.gwt.user.client.rpc.AsyncCallback<T>**, dove T è il tipo di ritorno del metodo nell'interfaccia sincrona.
Se la nostra interfaccia sincrona fosse

```
public interface MyService extends RemoteService {
  public String getData();
}
```

avremmo la corrispondente interfaccia asincrona

```
public interface MyServiceAsync {
  public void getData(AsyncCallback<String> callback);
}
```

Tale oggetto **AsyncCallback** è il cuore del meccanismo: sarà lui infatti ad essere notificato al momento del completamento della chiamata RPC.
Esso ha due metodi: **onFailure(Throwable)** e **onSuccess(T)**. Il client creerà un oggetto **AsyncCallback** e fornirà un'implementazione per i suoi due metodi. Poi userà l'interfaccia asincrona per accedere al servizio, alla quale passerà il *callback*:

```
myService.getData(new AsyncCallback<String>() {

  public void onFailure(Throwable caught) {
    doThisOnFailure(caught);
  }

  public void onSuccess(String result) {
    doThatOnSuccess(result);
  }
});
```

Dopo tale invocazione, il client ritorna immediatamente. Quando la servlet avrà terminato il suo lavoro invocherà uno dei due metodi del *callback* passatogli: **`onSuccess`** se ha terminato correttamente, oppure **`onFailure`** se si sono verificati errori.

Questo è il meccanismo con cui sono state implementate le comunicazioni dal lato client al lato server nel nostro prototipo. Le implementazioni dei servizi a loro volta invocano servizi propri del core di NRGine per portare a termine i loro compiti, ma questa seconda comunicazione avviene tra classi Java tutte risiedenti sul server e dunque non utilizza RPC.
Il fatto che le chiamate RPC GWT siano asincrone ha delle conseguenze per quanto riguarda la programmazione delle nostre interfacce grafiche, che per essere costruite richiedono informazioni che risiedono sul server: è stato necessario entrare nell'ottica di tale funzionamento, e gestire la costruzione delle interfacce come se fosse stata guidata da eventi, come appunto il ritorno di un *callback*.

4.4.2 Serializzazione

In generale il sistema di RPC di GWT supporta **`java.io.Serializable`**, ma alcune precisazioni sono dovute.
Inizialmente è stata introdotta in GWT un'interfaccia **`IsSerializable`** per identificare le classi che potevano essere serializzate. Tale interfaccia è vuo-

ta, come **java.io.Serializable**, e per essere implementata richiede solo la presenza nella classe di un costruttore senza argomenti.

La ragione per mantenere due interfacce separate era che la serializzazione effettuata da GWT è molto più semplice rispetto alla serializzazione standard di Java. Permettere di usare **java.io.Serializable** avrebbe dato l'impressione erronea che le due serializzazioni fossero identiche: i programmatori avrebbero potuto sopravvalutare le capacità della serializzazione GWT, e avrebbero anche potuto trovarsi nella situazione di doversi preoccupare di dettagli, come i Serial Version ID, che in realtà la semplificata serializzazione GWT avrebbe ignorato.

Nonostante ciò resti valido, GWT a partire dalla versione 1.4 ha iniziato a supportare ambedue le classi, per andare incontro alle richieste degli sviluppatori che chiedevano che non fosse loro imposto di far implementare l'interfaccia **IsSerializable** a tutte le loro classi che già implementavano **java.io.Serializable**.
È quindi ora possibile utilizzare tale interfaccia, con la condizione di inserire i tipi che la implementano in una *whitelist* all'interno di un file di *policy* per la serializzazione che viene prodotto durante la compilazione GWT. Tale file di policy deve poi essere schierato sul web server come risorsa pubblica, accessibile dalle **RemoteServiceServlet** attraverso **ServletContext.getResource()**.
Non si deve pensare che il "supporto" di **java.io.Serializable** sia altro che una concessione fatta per l'interoperabilità: GWT tratta questa interfaccia come un sinonimo di **IsSerializable**, e, come già detto, le considerazioni precedenti restano valide.
Questo significa anche nessuna delle classi che implementano **Serializable** nel JRE la implementa nella JRE emulata di GWT: classi come **Throwable** o **StackTraceElement**, che pure sono emulate da GWT[8] non possono essere trasferite via RPC dal momento che il client non è in grado di serializzarle o deserializzarle, nonostante appunto implementino l'interfaccia **java.io.**

[8] http://code.google.com/intl/it/webtoolkit/doc/latest/RefJreEmulation.html

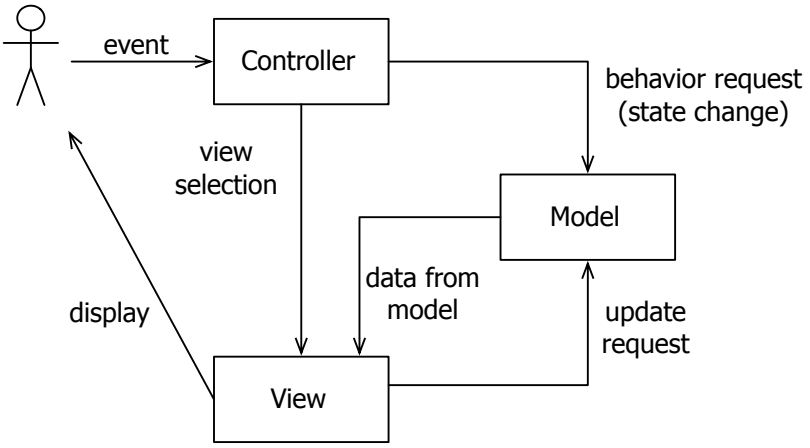

Figura 4.5: La struttura dell'architettura *model-view-controller* (da [Pre05])

`Serializable`.
È invece possibile trasferire classi come **`String`** o **`Number`**, dal momento che nella JRE emulata hanno serializzatori specifici per ogni campo, e sono quindi comprensibili dal client.

Nel prototipo le classi del package **`server`** comunicano coi servizi del core di NRGine, e comunicano col lato client mediante istanze di classi create ex novo: non abbiamo avuto quindi necessità di interoperabilità o compatibilità, e i nostri *data transfer object* implementano direttamente l'interfaccia GWT **`IsSerializable`**.

4.4.3 Architettura Model-view-controller

L'architettura *Mode-View-Controller* [Pre05] [Mic] è un modello di infrastruttura che permette di separare l'interfaccia utente dai contenuti e dalle funzionalità dell'applicazione attraverso tre livelli o componenti separati: modello, vista, e controllore. Il modello racchiude i contenuti e la logica applicativa, la vista costituisce l'interfaccia vera e propria, permette di presentare i contenuti e manipolarli secondo la logica. Il controllore agisce da intermediario tra i due. Le viste possono essere più di una.

Vi sono diverse descrizioni di quest'architettura, con minime ma significative differenze in come i diversi livelli interagiscono tra loro. Nella versione riportata in figura 4.5 il controllore intercetta le richieste dell'utente, che gli giungono sotto forma di eventi, e li inoltra al modello, se questi implicano un cambio di stato, e alla vista, se questa deve rispondere in qualche modo. La vista può richiedere dati aggiornati dal modello, e il modello a sua volta necessita la vista per la presentazione dei suoi dati.

GWT propone [Ram10], per lo sviluppo di grandi applicazioni, l'utilizzo di una versione del Model View Controller chiamata Model View Presenter. In questa versione il presentatore è una specie di controllore, ma specifico per una certa vista. Ogni vista potrà avere o meno un suo proprio presentatore, che conterrà la logica specifica per la vista. La logica che invece non dipende da alcuna vista resterà nel controllore.
Tuttavia, date le dimensioni ridotte del prototipo, ho scelto di non accogliere questo suggerimento bensì di limitarmi a utilizzare l'implementazione *light-weight* di MVC proposta da ExtGWT. Nonostante questa sia stata oggetto di critiche [Gér09] per via della dipendenze reciproche tra viste e controllori, ho ritenuto che per un'implementazione iniziale, e per un progetto come questo nel quale l'obiettivo principale fosse dimostrare di poter avere un prototipo funzionante, l'uso di tale pattern potesse essere più che accettabile.

Le comunicazioni avvengono attraverso il lancio di eventi: diversi sono definiti da ExtGWT, come **`LoadEvent`** o **`SelectionEvent`**, in più è chiaramente possibile definirne di nuovi specifici per l'applicazione.
I controllori si registrano per una specifica lista di eventi, ai quali vogliono rispondere. Gli eventi sono intercettati dal **`Dispatcher`** che provvede a inoltrarli ai controllori che hanno chiesto di essere notificati. I controllori devono anche essere resi noti al dispatcher stesso.

Vediamo quindi come si è utilizzato il modello MVC all'interno del prototipo. Abbiamo tre diversi tipi di eventi: **`Start`** indica il caricamento del modulo, **`LoginSuccessful`** indica il successo di una procedura di login, e **`WindowRequest`**. Un solo controllore, **`AppController`** è presente: esso esten-

de la classe **com.extjs.gxt.ui.client.mvc.Controller** ed è registrato per essere notificato nel caso di tutti e tre gli eventi dell'applicazione.
In una versione più estesa dell'applicazione controllori multipli potrebbero essere presenti: per ora questa necessità non c'è, **AppController** conta meno di un centinaio di righe.
La successione degli eventi, e le risposte che questi innescano, sono come segue:

- al momento del caricamento della pagina web, il metodo **onModuleLoad** della classe **EntryPoint** viene eseguito. In tale metodo si ottiene un riferimento al **Dispatcher**, che è una classe di tipo singleton, gli si associa un nuovo oggetto di tipo **AppController** e infine si fa partire un evento **Start**

- **AppController**, ricevendo **Start**, crea la finestra di login

- quando un utente esegue con successo la procedura di login, la stessa **Window** invia al **Dispatcher** un evento **LoginSuccessful**

- il **Dispatcher** inoltra l'evento all'**AppController**, il quale crea il desktop.

- espandendo il menu, l'utente può selezionare **MenuEntry** che aprono dei sottomenu, o che corrispondo a finestre. Nel primo caso, nessun evento è generato, e i sottomenu sono caricati dal desktop stesso, che effettua la chiamata RPC. Nel secondo caso, un evento **WindowRequest** è lanciato.

- **AppController**, ricevendo **WindowRequest**, fa una richiesta ad una factory **CustomWindowFactory** a cui passa le informazioni ricevute con l'evento. La factory è da queste in grado di creare la finestra corretta, che viene aggiunta al desktop. Ulteriori dettagli su questo passaggio saranno dati nella sezione 4.5.3.

Ulteriori miglioramenti sono possibili a questo modello. L'architettura *model-view-controller* di ExtGWT potrebbe essere completamente rimossa,

e sostituita da un'implementazione nuova che riducesse le dipendenze reciproche, o, ancora meglio, direttamente dal modello MVP proposto da Google [Ram10].

4.4.4 Internazionalizzazione delle stringhe

GWT offre tre diverse tecniche di internazionalizzazione per le stringhe: internazionalizzazione statica, dinamica, e attraverso la classe **`Localizable`**. In questo progetto di tesi non era prevista un'eventuale internazionalizzazione dell'applicazione, ma la tecnica dell'internazionalizzazione statica è stata comunque utilizzata per parametrizzare messaggi rivolti all'utente e altre stringhe che potessero comparire all'interno del codice.

Delle tre tecniche, l'internazionalizzazione statica è la più semplice ma anche la più efficiente in termini di performance a tempo di esecuzione. Utilizza interfacce Java fortemente tipate e file di proprietà **`.properties`**[9]. L'idea è creare file **`.properties`**, con coppie parametro/valore, e per ognuno di essi un'interfaccia Java che estende una tra le seguenti tre interfacce tag (ossia interfacce vuote, che si limitano a decretare l'appartenenza della classe e un certo insieme):

- **`com.google.gwt.i18n.client.Constants`**,
- **`com.google.gwt.i18n.client.ConstantsWithLookup`**
- **`com.google.gwt.i18n.client.Messages`**.

L'interfaccia **`Constants`** è la più semplice: estenderemo questa interfaccia se vogliamo limitarci ad estrarre dal file di proprietà un valore costante a tempo di compilazione. Ad esempio per ottenere dal file **`MyConstants.properties`** la proprietà corrispondente a

```
usernameText = Utente
```

[9] `http://docs.oracle.com/javase/tutorial/essential/environment/properties.html`

dovremo creare nella nostra interfaccia un metodo che si chiami come il nostro parametro

```
1 public interface MyConstants extends Constants {
    String usernameText();
3 }
```

Quando nel codice vorremo fare riferimento alla stringa "usernameText" non avremo che da creare riferimento all'interfaccia **`MyConstants`** e utilizzare il metodo **`usernameText()`**

```
  MyConstants myConstants = GWT.create(MyConstants.class);
2 usernameTextField.setFieldLabel(myConstants.usernameText());
```

Di tutto il resto, ossia di associare il metodo al parametro, e di recuperare il valore, si occuperà GWT. File di proprietà e interfacce corrispondenti dovranno trovarsi nello stesso package, il quale a sua volta deve trovarsi all'interno del path noto al modulo come contenente codice da compilare in JavaScript.
Annotazioni possono essere utilizzate per specificare che il metodo deve cercare un parametro con un nome diverso, indicatogli, o per indicare un valore di default.
Il valore di ritorno può essere **`String`** come nell'esempio, ma anche **`String[]`**, **`int`**, **`float`**, **`double`**, **`boolean`** o **`Map`**. Al momento della compilazione GWT verificherà prima di tutto che esista una coppia parametro/valore per ogni metodo, e che il valore fornito corrisponda al tipo specificato per il valore di ritorno, così da evitare potenziali errori di tipo a runtime.

L'interfaccia **`ConstantswithLookup`** è uguale a **`Constants`**, ma aggiunge una famiglia di metodi del tipo **`getString(String)`** per permettere di ottenere un valore, dato il nome del parametro, direttamente a runtime.
L'interfaccia **`Messages`** permette di creare invece stringhe con sostituzione di parametri, e anche di modificare l'ordine dei parametri a seconda delle diverse lingue. Il messaggio nel file **`.properties`** si dovrà conformare a

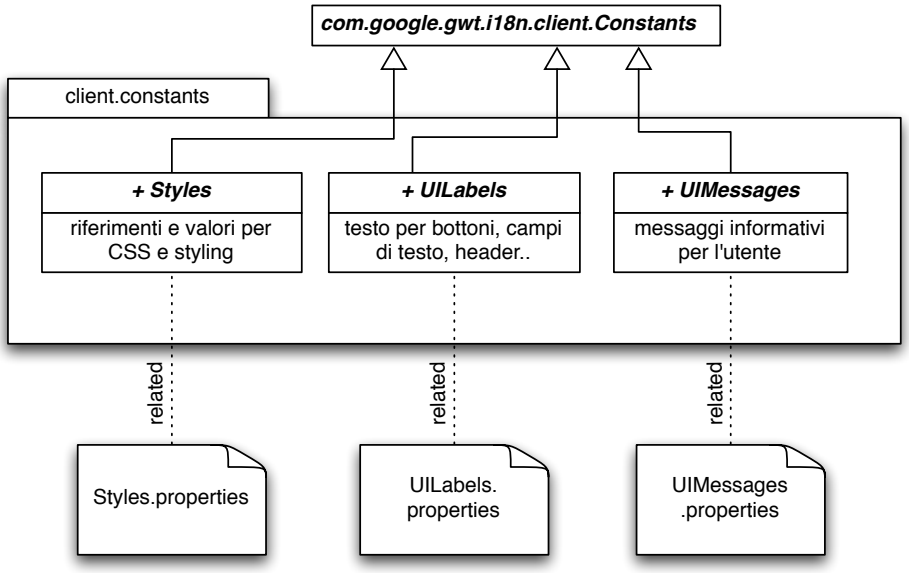

Figura 4.6: Contenuto del package **`client.constants`**. Nello stesso package sono raccolte le interfacce e i corrispondenti file **`.properties`**. Al momento tutte e tre le interfacce estendono la stessa interfaccia **`Constants`**.

`Java.text.MessageFormat`, e il metodo corrispondente avrà tanti parametri quanti quelli nella stringa associata.

Al momento nel prototipo sono presenti tre interfacce, raccolte nel package **`constants`**, le quali estendono tutte **`Constants`**. In figura 4.6 è riportato il contenuto del package e le relazioni reciproche.
`Styles`, contiene stringhe di testo quali valori per gli attributi **`class`** o **`id`** nei CSS, o path per il reperimento delle icone, e in generale informazioni di tipo presentazionale o di collegamento tra gli elementi grafici e i CSS. **`UILabels`** contiene stringhe quali etichette per i pulsanti, le intestazioni delle finestre o i campi di testo.
`UIMessages` contiene messaggi di informazione ed errore per l'utente. Quest'ultima interfaccia è ovviamente candidata per un cambiamento, passare a estendere **`Messages`** piuttosto che **`Constants`**, ma non vi è stata per il momento tale necessità.

4.5 Interfacce grafiche

In questa sezione presenterò le interfacce sviluppate, e i widget di ExtGWT che si sono utilizzati per la loro implementazione.
Le immagini che ritraggono le interfacce saranno offuscate per proteggere i dati di Wincor Nixdorf.

4.5.1 Schermata di login

La schermata di login è la prima interfaccia che è stata sviluppata, temporalmente parlando, e ha dunque costituito il primo elemento grafico sviluppato con GWT/ExtGWT.
La creazione dell'interfaccia è ordinata dall'**`AppController`**, che intercetta un evento che indica l'accesso all'applicazione web, il quale è a sua volta lanciato nella classe che implementa l'interfaccia **`com.google.gwt.core.client.EntryPoint`**.
La figura 4.7 mostra la finestra di login su quello che sarà il *background* del nostro web desktop, che riempie tutto lo spazio del browser dedicato alla visualizzazione delle pagine web. In basso il pannello presente è il **`DivLogger`** della libreria di logging **`gwt-log`** (vedi sezione 4.2).
La figura 4.8 mostra invece la stessa finestra con un messaggio di errore.

La finestra di login è chiaramente un elemento grafico molto semplice: estende il widget ExtGWT **`Window`**, ha un layout di tipo **`FormLayout`** nel quale sono inseriti tre **`TextField`** e due **`Button`**. Tutte queste classi fanno parte del package ExtGWT **`com.extjs.gxt.ui.client.widget`**.
Per l'indicazione di un eventuale errore in fase di login si è utilizzato un campo **`com.extjs.gxt.ui.client.widget.Text`**, inizialmente nascosto e reso visibile solo in caso di necessità. Dal momento che non è possibile inserire un campo di tipo **`Text`** all'interno di un **`FormLayout`** lo si è inserito all'interno di un oggetto **`com.extjs.gxt.ui.client.widget.form.AdapterField`**, il quale può far parte del layout.

Per portare a termine la procedura di login la finestra interpella un servizio remoto mediante RPC, alla quale fornisce le credenziali inserite dall'utente.

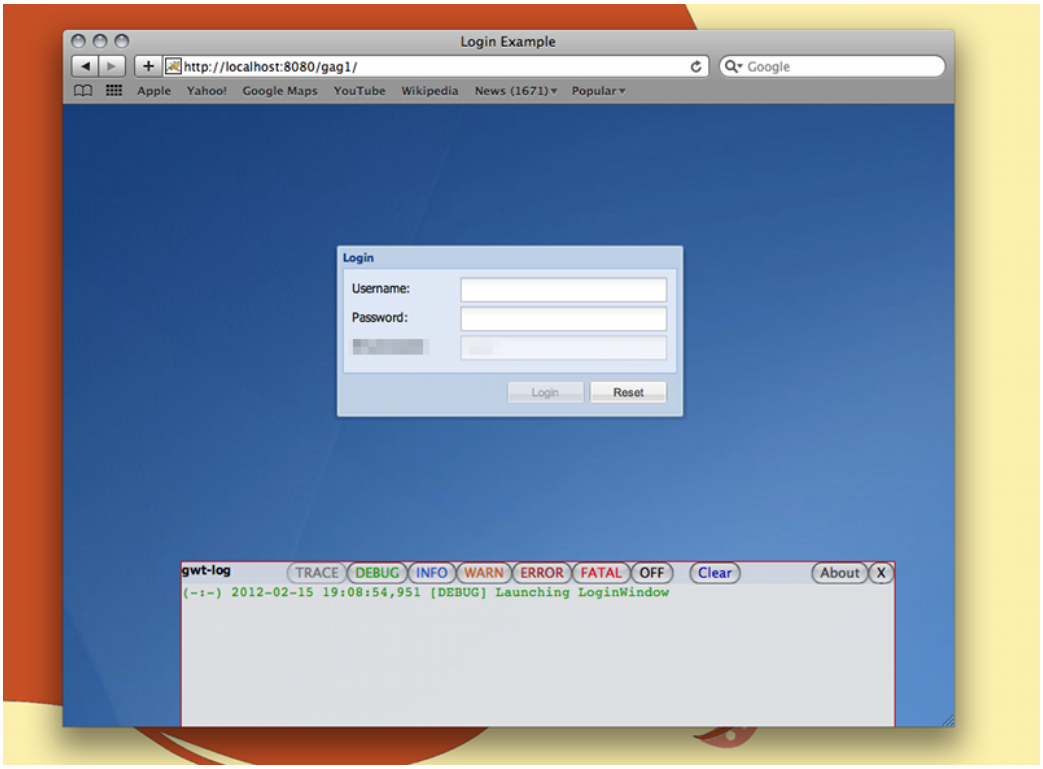

Figura 4.7: Esempio della schermata di login all'interno del browser. Il pannello in basso è il **`DivLogger`** della libreria **`gwt-log`** (vedi sezione 4.2)

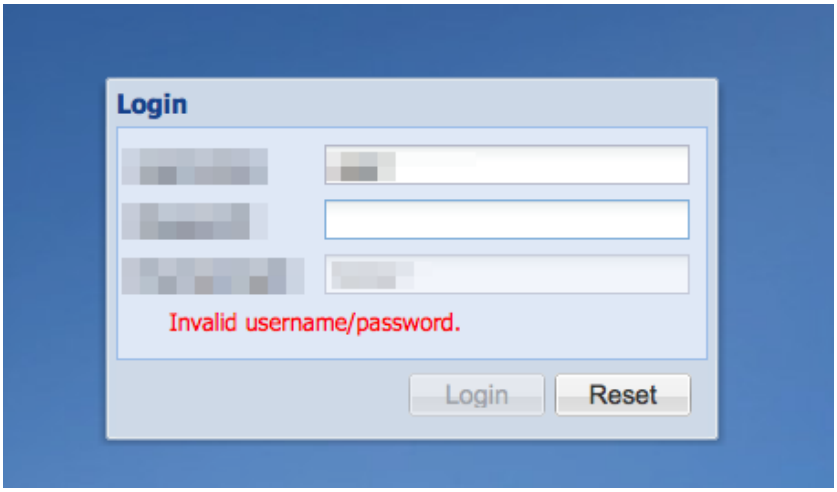

Figura 4.8: Schermata di login con un esempio di messaggio di errore

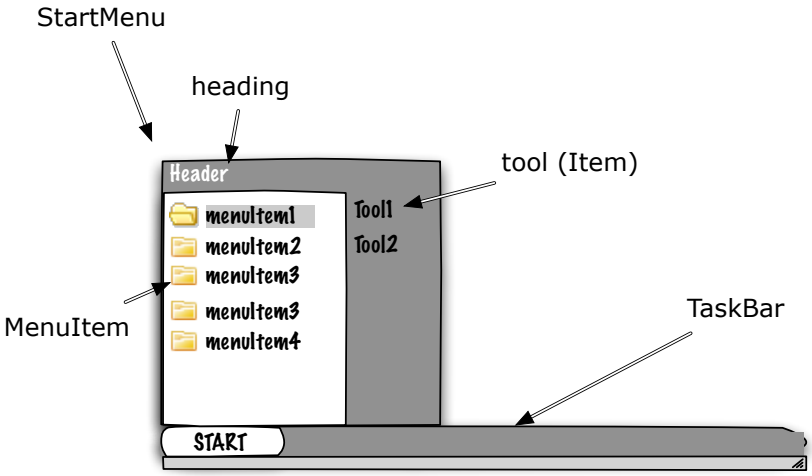

Figura 4.9: Illustrazione degli elementi che compongono uno **`StartMenu`**

Il servizio risponde con un'eccezione nel caso l'autenticazione sia fallita, la quale viene riportata nel campo apposito a cui ho appena accennato.
Nel caso l'autenticazione sia terminata con successo la finestra di dialogo viene nascosta e un evento che indica la corretta terminazione della procedura di login è lanciato.

4.5.2 Menu

Si è poi passati a implementare un web desktop vero e proprio. Fortunatamente, di una simile interfaccia era già presente un'implementazione, fornita da ExtGWT come esempio di possibile uso della sua collezione di widgets.
Il nostro web desktop estende quindi **`com.extjs.gxt.desktop.client.Desktop`** . Si tratta di un web desktop minimale: nessuna icona o finestra aperta di default, e solo un oggetto **`com.extjs.gxt.desktop.client.TaskBar`**, ossia un menu di avvio (**`com.extjs.gxt.desktop.client.StartMenu`**) seguito dalla liste delle finestre aperte.
La particolarità del nostro web desktop sta nel modo in cui viene creato lo **`StartMenu`**. Vediamo prima quali sono gli elementi che compongono un tale menu, facendo riferimento all'immagine in figura 4.9. Uno **`StartMenu`** ha

un'intestazione (*heading*), e un menu cosiddetto degli strumenti (*tool*). La parte centrale del menu raggruppa invece gli elementi del menu veri e propri, del tipo **`com.extjs.gxt.ui.client.widget.menu.MenuItem`**.

Mentre il menu degli strumenti è standard, costruito staticamente (conterrà elementi come *logout*, o altri tipi di *utility*), sia l'intestazione che i **`MenuItem`** sono caricati dinamicamente dal core di NRGine, in quanto corrispondono a oggetti del *business layer*.
Dunque, quando viene richiesta la creazione del desktop, una prima chiamata RPC viene fatta dal client in direzione del server, richiedendo il valore per l'intestazione, e gli elementi del menu principale. Gli elementi del menu potranno essere di due tipi: o essere a loro volta contenitori per un sottomenu (*sub-menu*), o rappresentare delle finestre, che saranno aperte nel caso siano selezionate.
La selezione di un **`MenuItem`** del primo tipo porta a una nuova RPC, che richiede le informazioni per visualizzare il sottomenu. La selezione invece di un **`MenuItem`** del secondo tipo porta alla lancio di un evento di tipo **`RequestWindow`**. Ogni **`MenuItem`** contiene le informazioni necessarie per capire che tipo di finestra viene richiesta. Nel caso la finestra sia già presente nel desktop, questa viene resa visibile e portata sopra le altre.

La figura 4.10 mostra lo **`StartMenu`** espanso, con un **`MenuItem`** che sta caricando il rispettivo sottomenu. In figura 4.11 osserviamo come tale sottomenu sia stato caricato, e sia visibile un ulteriore menu, composto in questo caso di sole finestre.
Per via delle chiamate RPC necessarie per il caricamento degli elementi del menu, la visualizzazione di un sottomenu, o di una finestra, può essere istantanea ma anche richiedere un certo tempo. Nel frattempo comunque, l'interfaccia resta in grado di rispondere a ulteriori input da parte dell'utente.

4.5.3 Griglie

L'ultima interfaccia a cui mi sono dedicata è la finestra con griglia. La creazione di una simile finestra può avvenire a seguito della selezione di un

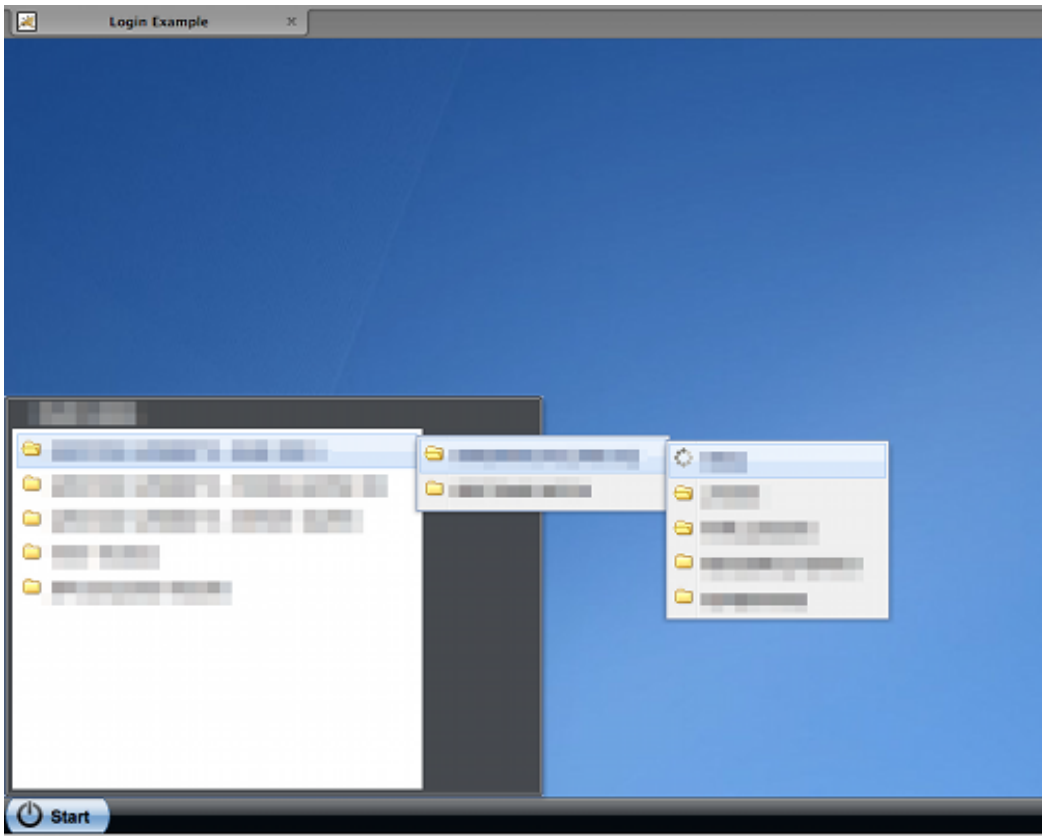

Figura 4.10: Menu del web desktop. I sottomenu sono caricati dinamicamente: si osservi l'icona che indica che il sottomenu corrispondente al primo elemento del terzo menu aperto sta caricando.

elemento del menu del web desktop, come già descritto, e come illustrato dalla figura 4.12.
Una finestra con griglia sarà una generica finestra di tipo **`CustomWindow`** (semplicemente minimizzabile, massimizzabile e collassabile, non modale, e con un menu stardard) alla quale è aggiunto un componente tale **`CustomGrid`**.

`CustomGrid` estende **`com.extjs.gxt.ui.client.widget.ContentPanel`**, e ad essa è aggiunto un oggetto di tipo **`Grid<BeanModel>`**, ossia una griglia che contiene elementi di tipo **`com.extjs.gxt.ui.client.data.BeanModel`**. Al momento gli elementi visualizzati nelle griglie sono estensioni di **`BeanModel`**, ma questo non è pratico, dal momento che significherebbe prendere tutta una serie di elementi propri del *business layer*, creare un oggetto **`BeanModel`** corrispondente per ciascuno di loro, wrappare le informazioni in tale oggetto e passare questo al client.
Alcuni approcci alternativi sono tuttora in corso d'esame: una soluzione potrebbe trovarsi nelle interfacce ExtGWT **`BeanModelTag`** o **`BeanModelMarker`**, che permettono di identificare una Bean che può essere usata per generare istanze di tipo **`BeanModel`**.

La griglia per essere creata necessita di due elementi: un **`ListStore<BeanModel>`** e un **`ColumnModel`**. Il **`ColumnModel`** è un descrittore delle colonne della grid, del loro id, titolo e dimensione. Tali informazioni si richiedono a un servizio specifico, che le restituisce al client mediante un *dto*, dal momento che il **`ColumnModel`** è un oggetto proprio dell'interfaccia e non può essere creato lato server. Dal *dto* il client crea il **`ColumnModel`**.
`ListStore` è un tipo di **`Store`**, ossia un oggetto che incapsula una cache di oggetti **`BeanModel`**.
La nostra griglia è di tipo *paging*, ossia i dati sono visualizzati partizionati in pagine di dimensione definita: per ottenere la visualizzazione di una pagina il client effettua una richiesta al server che gli restituisce solo la pagina richiesta. Data la potenziale grande mole di dati per una griglia, il client non effettua la ripartizione per pagine da sé.

Come creare il **ListStore**?
Prima di tutto creeremo un oggetto proxy, che fungerà da intermediario tra il client e il servizio remoto che è in grado di restituire gli oggetti **BeanModel**. Tale servizio dovrà accettare come parametro un oggetto **PagingLoadConfig**, che sarà portatore di due informazioni: il numero di elementi richiesti per pagina e l'indice del primo elemento per la pagina corrispondente, e dovrà restituire un oggetto **PagingLoadResult<BeanModel>**, ossia un elenco degli elementi da visualizzare. Nel nostro caso, dal momento che le comunicazioni col server avverranno via RPC, il proxy sarà di tipo **com.extjs.gxt.ui.client.data.RpcProxy**.

Una volta creato il proxy, a partire da esso e da un oggetto **BeanModelReader** si potrà creare il loader, di tipo **PagingLoader**. Al loader si collegherà una **PagingToolBar**, la barra che permette di muoversi avanti e indietro tra le diverse pagine. Infine, col loader potremo generare il **ListStore**.
Dunque, nel momento in cui si crea la finestra, o si fa la richiesta, attraverso la toolbar, per una nuova pagina, ciò che accade è che una nuova **PagingLoadConfig** viene generata, passata al proxy e da questo al servizio remoto. Quando il servizio remoto risponde con il **PagingLoadResult** lo store viene aggiornato e di conseguenza la griglia.

Le colonne della griglia hanno già di default un menu di colonna col quale si può scegliere quali colonne visualizzare e quali nascondere, e se ordinare gli elementi di una pagina in base al valore di una colonna.
A seconda del tipo di dato visualizzato in una colonna, è anche possibile aggiungere dei **FilterConfig** col quale visualizzare solo le righe che corrispondono a un dato criterio. Tale funzionalità in fase di aggiunta.

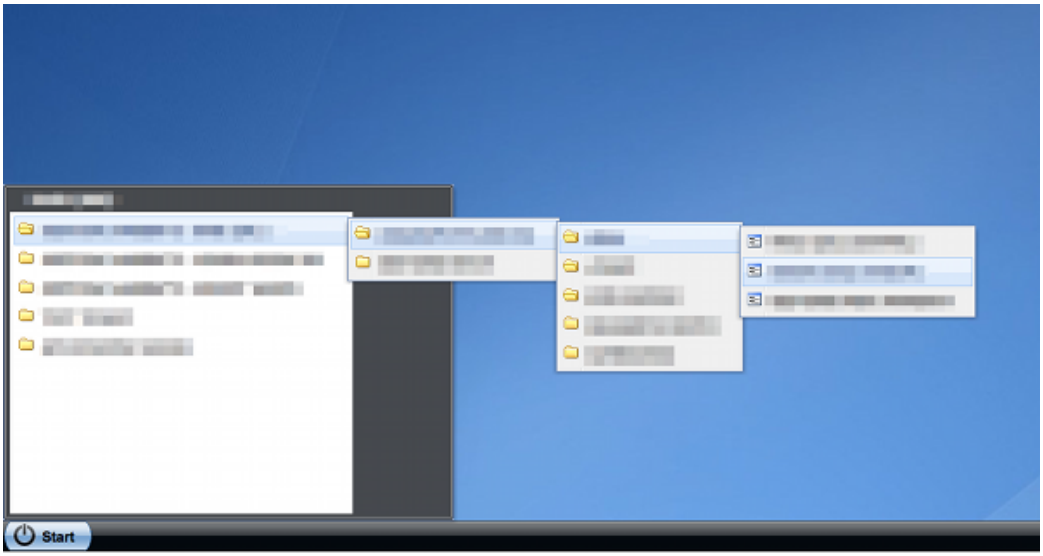

Figura 4.11: Menu del web desktop. Gli elementi del menu possono aprire dei sottomenu o delle finestre. Qui nell'ultimo menu aperto in ordine di profondità osserviamo come le icone dei **`MenuItem`** indichino che ognuna di esse aprirà una finestra e non un nuovo sottomenu.

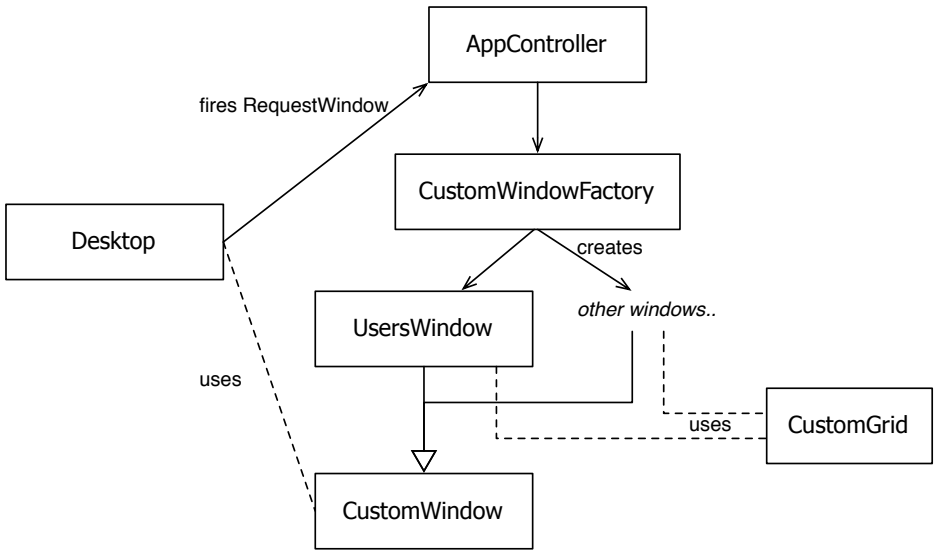

Figura 4.12: Creazione di una finestra con griglia. Il desktop lancia un evento di tipo **`Requestwindow`** che viene intercettato dall'**`AppController`**. L'**`AppController`** fa richiesta alla **`customWindowFactory`** per la creazione di un oggetto di tipo **`CustomWindow`**. Le informazioni proprie dell'evento conterranno anche i dettagli su quale **`CustomWindow`** creare. In questo esempio, si tratta di una **`UsersWindow`** che utilizza per la visualizzazione dei dati una **`CustomGrid`**

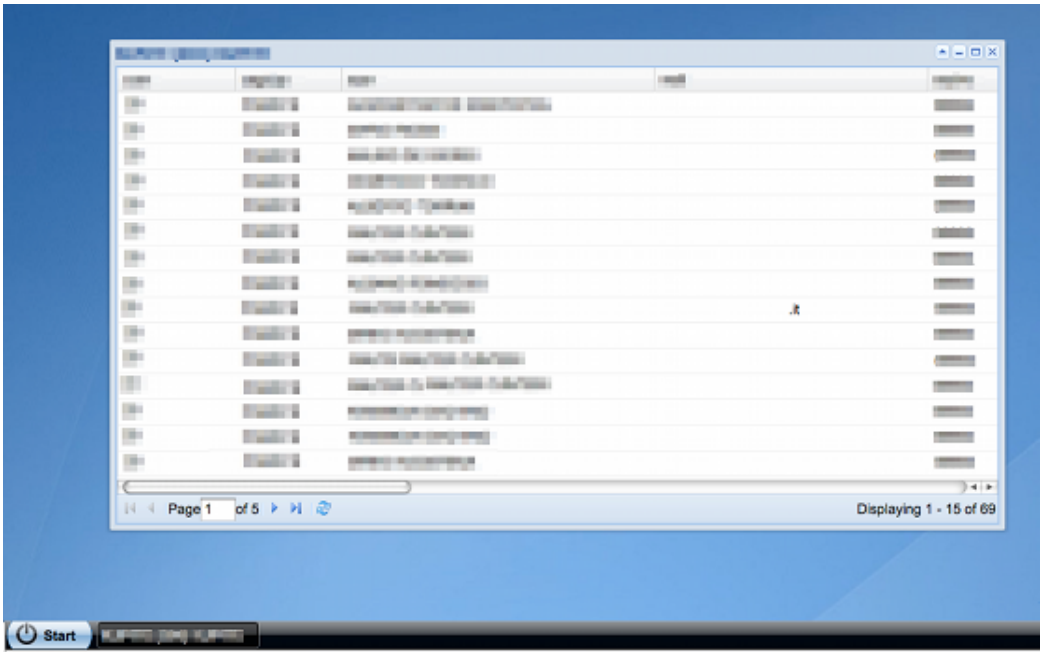

Figura 4.13: Finestra con griglia. In basso possiamo osservate la **`PagingToolBar`**. Le colonne sono ridimensionabili, ed è possibile nasconderle mediante il menu che è visibile aperto in figura 4.14

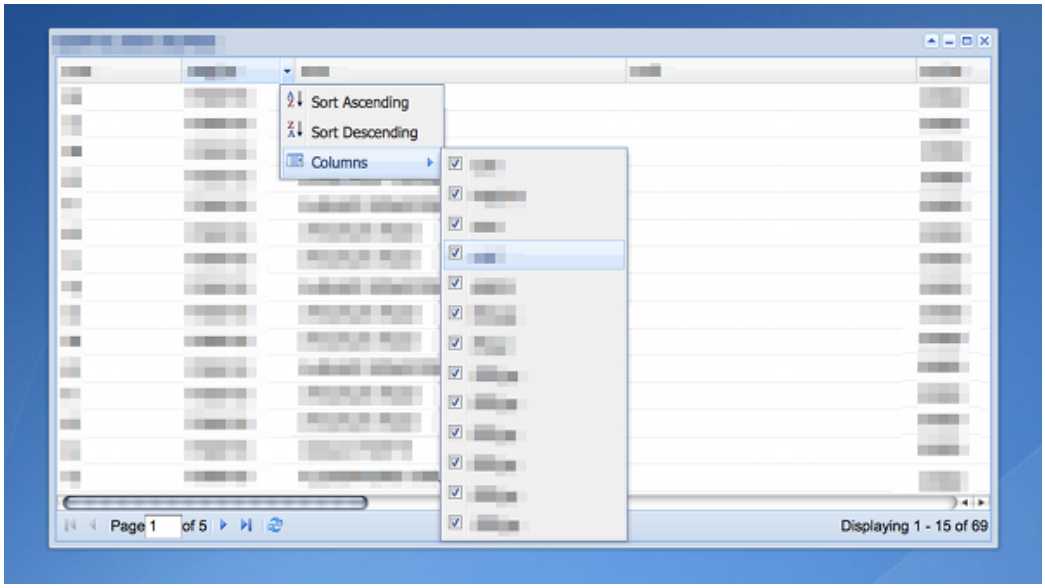

Figura 4.14: Finestra con griglia e menu di colonna aperto. I menu di colonna sono modificabili. Qui il menu permette di ordinare la le righe in modo crescente o decrescente a seconda dei valori della colonna, e di nascondere le colonne. Altre opzioni possono essere aggiunte, ad esempio è anche possibile anche inserire dei filtri e visualizzare solo le colonne che corrispondono al filtro.

Capitolo 5

Valutazione

In questo capitolo si effettuerà una breve valutazione del prototipo sviluppato. Il prototipo è stato valutato solo informalmente, insieme ai responsabili di Wincor Nixdorf.
Le interfacce sviluppate rispecchiano in modo fedele le interfacce preesistenti, salvo alcune modifiche effettuate nella speranza di renderle più usabili, così come era stato richiesto, grazie ai parallelismi tra le due librerie ExtJS e ExtGWT.

Il codice è effettivamente più semplice e comprensibile, e dunque più mantenibile, rispetto al precedente codice JavaScript/ExtJS, grazie principalmente all'uso di Java stesso, al meccanismo di RPC di GWT e internazionalizzazione statica di GWT.
Il linguaggio Java pone una serie di vincoli aggiuntivi rispetto a JavaScript, basti pensare alla forte tipizzazione. Se da un lato questi vincoli costituiscono costrizioni, da un altro rendono il codice più chiaro perché è più semplice capire che cosa si sta manipolando. Il paradigma a oggetti poi è ideale per l'utilizzo di *pattern*, schemi noti e definiti, come ad esempio il *factory* a cui si è accennato nella sezione 4.3.1. Questi schemi, oltre a risolvere specifici problemi, sono autodescrittivi.

Il meccanismo di RPC di GWT (vedi sezione 4.4.1) permette di abbandonare *HTTP request*: client e server possono comunicare sempre via invocazioni di metodi Java.

Infine, il meccanismo di GWT di internazionalizzazione statica è stato utilizzato per spostare al di fuori delle classi Java ogni tipo di stringa, fosse essa un messaggio di errore, una qualche forma di *label*, ad esempio per un pulsante o una finestra, o un indicazione di stile come id per CSS o dimensioni di margini: in questo modo parte delle modifiche all'interfaccia possono essere eseguite solo manipolando coppie *key/value* in file di proprietà, senza bisogno di intervenire all'interno del codice.

Oltre alla chiarezza e del codice e alla sua mantenibilità, si è anche avuto modo di verificare alcune prestazioni del prototipo, in particolare la velocità con cui il web desktop veniva visualizzato sul browser dell'utente finale, e la sua velocità di risposta agli eventi generati dall'utente come il click del mouse. Tale velocità è stata giudicata positivamente: è infatti risultata migliore rispetto a quella del web desktop attualmente in uso, in JavaScript/ExtJS. Presumibilmente, tale maggiore velocità è da attribuirsi alle ottimizzazioni effettuate dal compilatore GWT sul codice JavaScript generato (vedi sezione 2.3.3), ottimizzazioni che non vi sono per i widget ExtJS.

Capitolo 6

Conclusioni e sviluppi futuri

In questa tesi ho illustrato lo sviluppo di un prototipo di web desktop con Google Web Toolkit e la libreria ExtGWT di Sencha Inc., per indagare se tali tecnologie potevano costituire una valida alternativa alla combinazione AJAX/ExtJS con cui è implementato il web desktop che costituisce il *presentation layer* del framework NRGine di Wincor Nixdorf.

Il prototipo sviluppato è stato considerato soddisfacente, e dunque GWT/ExtGWT una valida alternativa alla combinazione AJAX/ExtJS del web desktop attuale.
GWT permette di ottenere codice JavaScript a partire da codice Java, molto più chiaro, comprensibile e mantenibile. È anche più semplice da documentare, e, grazie al compilatore GWT e al *development mode*, più facile da debuggare. Tutti gli strumenti disponibili per lo sviluppo in Java possono essere inoltre utilizzati per programmare in GWT, con un grande numero di vantaggi: meno errori grazie ai controlli sintattici, maggiore velocità grazie all'autocompletamento...
Anche la velocità di *rendering* del web desktop sviluppato è stata valutata positivamente: essa è migliore di quella mostrata dal web desktop precedentemente in uso, indice del fatto che il compilatore GWT è effettivamente in grado di produrre codice performante.

Avere l'occasione di studiare e utilizzare Google Web Toolkit è stata un'esperienza estremamente interessante. Si tratta a mio parere di una proposta innovativa e originale, e trovo che i vantaggi dello sviluppare in Java siano molti più degli svantaggi dell'allontanarsi dal linguaggio che viene effettivamente usato, JavaScript.
Google Web Toolkit ha un'ampia documentazione, e una vasta community alle spalle. È facile da installare e utilizzare, e permette di ottenere notevoli risultati con sforzi limitati, basti pensare alla comodità di utilizzare chiamate RPC per le comunicazioni client/server.
La libreria ExtGWT, per contro, è scarsamente documentata, fa un uso molto frequente di tipi generici, e non è troppo flessibile, ragion per cui il mio giudizio su di essa non è altrettanto positivo. Tuttavia, essa pare essere una delle più valide proposte esistenti per quanto riguarda l'ampliamento di Google Web Toolkit: la collezione di widget che fornisce è veramente ampia, e una nuova versione 3 è stata recentemente annunciata, dunque è lecito presumere che ulteriori miglioramenti siano prossimi.

Il prototipo sviluppato è, ovviamente, solo una bozza di quello che sarà il web desktop finale. I possibili sviluppi futuri sono innumerevoli. A molti si è già accennato durante la descrizione di quanto è stato fatto: ora riporterò i principali.
Le finestre con forms andranno implementate, e altre funzionalità sono necessarie per le finestre con griglie, come la possibilità di filtrare i risultati secondo la corrispondenza con ricerche testuali o espressioni matematiche, come ad esempio il fatto che una data colonna possieda un valore positivo.
A questo proposito, anche il problema della creazione di **`BeanModel`** per i diversi dati da visualizzare dovrà essere affrontato.

In futuro si dovrà eliminare il riferimento alla libreria **`gwt-log`** e utilizzare il logging fornito da GWT.

Nuove interfacce dovranno essere aggiunte, fino a rendere il nuovo web desktop completo quanto il precedente. Tale ampliamento porterà poi certamente alla necessità di ripensare lo schema di creazione delle finestre attuale.

La **`CustomWindowFactory`** potrebbe anche non rivelarsi inadeguata, ma di certo la sua implementazione dovrà essere modificata.
Al momento infatti **`CustomWindowFactory`** è una cosiddetta factory parametrizzata, ovvero utilizza una condizione per scegliere quale tipo di oggetto creare. Questo comportamento, benché ammissibile in un prototipo, sarà inaccettabile quando la factory dovrà trovarsi a scegliere tra più di cinquanta diverse finestre. Si dovrà quindi fare in modo che siano le singole classi a registrarsi presso la factory, così che l'aggiunta di una nuova classe a quest'ultima non renda necessario modificare il codice della factory stessa.

L'uso del modello *model-view-controller* potrà essere esteso e migliorato, anzi sarà opportuno valutare l'eventualità di sostituirlo col modello *model-view-presenter* proposto da GWT, il quale, è stato annunciato, sarà supportato dalla versione 3 di ExtGWT.
L'adozione di questo modello dovrebbe permettere di spostare le chiamate RPC al di fuori delle interfacce grafiche dentro il *presenter*, migliorando la modularità del codice, tutto a vantaggio della mantenibilità.
I messaggi rivolti all'utente potranno essere resi maggiormente informativi e personalizzati, utilizzando per essi invece che l'interfaccia **`com.google.gwt.i18n.client.Constants`** l'interfaccia **`com.google.gwt.i18n.client.Messages`**.

La gestione degli errori andrà migliorata, fornendo un meccanismo comune per tutte le interfacce, ed effettuando un'analisi sistematica delle diverse categorie di errori, e delle azioni da intraprendere per ognuna di questi.
Gli aspetti di sicurezza, gestione delle risorse e bilanciamento di carico, che non sono stati in alcun modo affrontati, dovranno essere valutati.

Oltre a questi aspetti più, diciamo così, "interni", anche una lunga lista di funzionalità è mancante e dovrà essere aggiunta. Il login si limita a verificare le credenziali dell'utente ma nessuna sessione viene realmente creata. Si deve aggiungere la creazione e la gestione della sessione, fare in modo che essa sia conservata se la scheda del browser è chiusa e riaperta, fornire un meccanismo di logout.

Sono previste finestre ulteriori che forniscano informazioni sulla sessione in corso, sull'utente, e così via.
È anche richiesto che le personalizzazioni che un utente apporta al web desktop, come ridimensionare le colonne di un data griglia, o nasconderne alcune, siano ricordate e riproposte identiche quando l'utente chiude e riapre la stessa finestra, o anche tra una sessione e l'altra.

Elenco delle figure

Bibliografia

[All02] Jeremy Allaire. Macromedia Flash MX-A next-generation rich client. http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf, March 2002.

[Gér09] Olivier Gérardin. Why Ext-GWT MVC is broken. http://blog.gerardin.info/archives/40, March 2009.

[GHJV09] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 2009.

[Gooa] Google. Google Web Toolkit, JRE Emulation Reference. http://code.google.com/intl/it/webtoolkit/doc/latest/RefJreEmulation.html.

[Goob] Google. Google Web Toolkit, Product Overview. http://code.google.com/p/google-web-toolkit-doc-1-4/wiki/ProductOverview.

[Gooc] Google. Google Web Toolkit, Showcase of Features. http://gwt.google.com/samples/Showcase/Showcase.html.

[Good] Google. GWT Coding Basics, Compatibility with the Java Language and Libraries. http://code.google.com/intl/it/webtoolkit/doc/latest/DevGuideCodingBasicsCompatibility.html.

[Gooe] Google. GWT Coding Basics, JavaScript Native Interface (JSNI).

http://code.google.com/intl/it/webtoolkit/doc/latest/DevGuideCodingBasicsJSNI.html.

[Goo12] Google. Google Web Toolkit, pagina ufficiale. http://code.google.com/intl/it/webtoolkit/, 2012.

[Got07] Greg Goth. The Google Web Toolkit shines a light on Ajax frameworks. *Software, IEEE*, 24(2):94 –98, 2007.

[Joh09] Bruce Johnson. Reveling in constraints. *Queue*, 7:30:30–30:37, July 2009.

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.

[Mic] Microsoft Developer Network. Model-View-Controller. http://msdn.microsoft.com/en-us/library/ff649643.aspx.

[Ohl11] Ohloh Community. Ext JS. http://www.ohloh.net/p/extjs, 2011.

[Pre05] Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. McGraw-Hill International Edition, 6th edition, 2005.

[Ram10] Chris Ramsdale. Large scale application development and MVP. http://code.google.com/intl/it/webtoolkit/articles/mvp-architecture.html, March 2010.

[Sau12] Fred Sauer. gwt-log: Logging Library for Google Web Toolkit (GWT) with Deferred Binding. http://code.google.com/p/gwt-log/, 2012.

[Sen12a] Sencha. Ext GWT, pagina ufficiale. http://www.sencha.com/products/extgwt/, 2012.

[Sen12b] Sencha. ExtJS, pagina ufficiale. http://www.sencha.com/products/extjs/, 2012.

[Sen12c] Sencha. Pagina ufficiale. http://www.sencha.com/, 2012.

[Sun06] Sun Microsystems. JavaOne, Sun's 2006 Worldwide Java Developer Conference. http://java.sun.com/javaone/sf/2006/, 2006.

[Voi07] Voices That Matter: Google Web Toolkit Conference. http://www.voicesthatmatter.com/gwt2007, 2007.

[Wik] Google Wiki, Google Web Toolkit. http://google.wikia.com/wiki/Google_Web_Toolkit.

[Win12] Wincor Nixdorf. Sito ufficiale. http://www.wincor-nixdorf.com, 2012.