

UNIVERSITA' DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica



Realizzazione di un Web Editor per Asmeta

Relatore:

Prof. Angelo Michele GARGANTINI

Tesi di Laurea Triennale

Favio Andree QUINTEROS TERRAZAS

Matricola n.1035982

ANNO ACCADEMICO 2020/2021

A tutte le persone a me care.

Sommario

La velocissima diffusione di Internet ha portato ad una notevole sviluppo di quelli che sono i servizi forniti da esso. Basti pensare ad esempio al concetto di *nuvola informatica*, o più propriamente in inglese il *cloud computing* per l'archiviazione di dati, in cui fino a due decenni fa era impensabile con le infrastrutture di allora, avere tanta comodità nel poter memorizzare qualche gigabyte di dati su un computer remoto. Oggi, invece, grazie al potenziamento della rete Internet, dovuta soprattutto alla diffusione sempre maggiore del servizio in fibra ottica, e anche all'abbassamento del rapporto euro/gigabyte, servizi come Google Drive oppure iCloud, sono ormai alla portata di tutti.

Più o meno per lo stesso motivo, anche la computazione a lato server è diventata molto diffusa. Esempi di questo tipo possono essere Word, Matlab e anche servizi di conversione di file. Questo è la ragione principale per la quale è stato dedicato interesse nel realizzare un Web Editor per Asmeta, nell'aver quindi la possibilità di poter scrivere del codice da qualunque computer, e poter almeno vedere se quello che si è scritto è sintatticamente corretto, senza la necessità di dover installare nessun software.

Sommario

Introduzione.....	7
1.1 Cosa è una Abstract State Machine in generale	7
1.2 Informazioni sull'elaborato	8
1.2.1 Inquadramento generale.....	8
1.2.2 Struttura dell'elaborato	9
Nozioni su Asmeta ed Xtext.....	10
2.1 Il framework Asmeta	10
2.2 Il linguaggio AsmetaL.....	11
2.3 Il framework Xtext.....	13
Fasi dell'implementazione Web	15
3.1 Semantica, sintassi e configurazione.....	15
3.1.1 Installazione di Xtext su Eclipse e generazione progetto	16
3.1.2 Il file .xtext	19
3.1.3 Il file .mwe2.....	20
3.1.4 Risultato intermedio.....	21
3.2 Progetto Web	22
3.2.1 Scelte progettuali	22
3.2.2 Input del testo tramite JavaScript	24
3.2.3 Comunicazione JSP-Servlet	25
3.2.4 Implementazione bottone di esecuzione	26
3.2.5 Il Controller ed il parsing	27
3.2.6 Definizione dello stile in CSS.....	33

Capitolo 1

Introduzione

Nel primo approccio, molte persone trovano difficoltà nel comprendere esattamente cosa è una Abstract State Machine (ASM). Questo può essere principalmente dovuto al metodo di apprendimento tradizionale rivolto in generale alla “computer science”. La sfida principale delle ASM sono quindi quelle di superare le tradizionali assunzioni sui modelli di calcolo. Molti sono infatti i loro sostenitori, secondo cui un primo approccio riguardo alle nozioni di algoritmo sarebbe più facile e naturale con le idee base degli ASM.

1.1 Cosa è una Abstract State Machine in generale

A prima vista, una Abstract State Machine sembra solo un insieme di istruzioni con assegnamenti e condizioni. In particolare, esistono, come vedremo, molte estensioni della versione base di ASM. Allora cosa rendono le ASM così particolari? Cosa fa delle ASM il modello di calcolo più potente ed universale rispetto ai modelli standard, così come scritto da Yuri Gurevich, suo inventore, nel 1985?

L’idea principale degli ASM è il modo sistematico di rappresentare la correlazione tra i simboli nella rappresentazione sintattica di un programma, agli elementi del mondo reale di uno stato. Uno stato in ASM può quindi includere qualsiasi oggetto o funzione nel mondo reale. In particolare, lo stato non assume nessuna rappresentazione simbolica a livello di bit rispetto ai componenti. Quindi sta qui la differenza rispetto ai modelli

computazionali tradizionali, come ad esempio la Macchina di Turing (in cui lo stato è strutturato in una collezione di simboli) ed il suo relativo Teorema, che hanno il difetto di soffermarsi più sulla trasformazione dei simboli, e non su ciò che essi rappresentano.

Ora è naturale chiedersi come tutto questo sia attuabile. Sono tanti gli algoritmi che possono essere definiti in ASM, ma che non possono essere implementati, anzi, non sono proprio destinati a tale fine. Piuttosto descrivono delle procedure del mondo reale. Ci possono essere ad esempio algoritmi che descrivono il funzionamento di un ATM con un relativo prelievo di denaro dal conto corrente.

Da questo punto di vista le ASM possono essere viste come la teoria del pseudocodice, in cui vengono descritte le dinamiche per i sistemi discreti.

Per concludere questa breve introduzione, è necessario quindi accennare quello che è la prospettiva delle ASM, che sarebbe quella di apportare fondamenti di informatica alla scienza, attraverso la capacità di poter dare una particolare nozione di algoritmo, in cui quelli “implementabili” sono solo una sottoclasse [1].

1.2 Informazioni sull’elaborato

Nei seguenti paragrafi è presente una descrizione generale dell’elaborato e successivamente una descrizione più dettagliata passo a passo.

1.2.1 Inquadramento generale

L’elaborato, prima di presentare quelli che sono gli aspetti più tecnici, cerca di descrivere in linea generale quello che è il contesto. Verrà quindi descritta la parte teorica su Asmeta ed il linguaggio AsmetaL, per poi affrontare quello che ci è di interesse, cioè il Web Editor. Qui ovviamente ci addentreremo negli aspetti più tecnici, descrivendo prima l’ambiente di sviluppo di contorno, quindi Eclipse ed Xtext, per poi andare più in profondità descrivendo lo sviluppo Web attraverso, ad esempio, le JSP e Servlet. In particolare, su quest’ultima, saranno descritte anche le difficoltà personali avute durante

lo sviluppo e le relative soluzioni. Per finire le conclusioni cercheranno di descrivere il progetto con un'analisi auto-critica.

1.2.2 Struttura dell'elaborato

La tesi è strutturata nel seguente modo:

Nel secondo capitolo è dedicato ad Asmeta ed Xtext. Viene spiegato prima di tutto cosa è appunto Asmeta, con i suoi relativi strumenti, per arrivare ad alcuni esempi nell'Asmeta language. Successivamente viene spiegato Xtext, quindi cosa è, e a cosa esso ci serve.

Nel terzo capitolo, che è quello più corposo, viene affrontato il core del progetto Web Editor. Innanzitutto, questo presenta due macroaree:

La prima comincia con l'illustrare come integrare il framework Xtext sull'ambiente di sviluppo Eclipse, specificando come generare un nuovo progetto. Successivamente si va a definire la semantica del testo sull'editor, attraverso la modifica del file .xtext. Infine vedremo come modificare le configurazioni attraverso il file .mwe2.

La seconda parte riguarda invece la parte web. Si comincia con l'illustrare quali sono i file generati automaticamente alla generazione del progetto e come interagiscono con il file principale dell'editor. Successivamente si discute su come modificare i file presenti per poter chiamare una funzione Java, per poter fare il parsing del testo in input. Si illustra quindi cosa sono le JavaServer Pages e le Servlet, necessarie per poter eseguire un metodo Java già esistente. Inoltre, viene spiegato come prendere il testo da tastiera attraverso una funzione in JavaScript. Per finire viene descritto la modifica al file .css per riuscire a modificare lo "stile" della pagina web, e quindi renderla più "gradevole" all'utente.

Nel quarto ed ultimo capitolo sono presenti le conclusioni, con la presentazione dei risultati finali. Vengono definiti quelli che sono i limiti delle scelte effettuate e quindi viene effettuata una auto-critica.

Capitolo 2

Nozioni su Asmeta ed Xtext

In questo capitolo viene spiegato cosa sono Asmeta ed Xtext per capire, per quanto riguarda quest'ultimo, a cosa ci servirà successivamente nel Web Editor.

2.1 Il framework Asmeta

Asmeta è un framework, quindi una struttura utile per la realizzazione di software, per il metodo formale delle Abstract State Machines (ASMs). Questo è basato sul metamodello per ASMs [2]. Per definizione cerca quindi di avvicinare il modo di comprendere umano, la formulazione dei problemi nel mondo reale e lo sviluppo delle soluzioni algoritmiche. In questo caso, però, il metamodello può anche essere inteso come una rappresentazione astratta dei concetti ASMs, in modo da avere uno scambio di formati standard per poter integrare anche i suoi strumenti.

Strumenti di cui Asmeta è composto, e si caratterizzano per le loro attività di validazione e verifica. Questi sono:

- Asmee: un editor per ASMs scritto nel linguaggio AsmetaL
- AsmetaLc: un compilatore/parser per i modelli AsmetaL
- AsmetaS: un simulatore
- AsmetaV: un validatore basato sui scenari
- AsmetaA: un animatore di esecuzione
- AsmetaSMV: un controllore di modelli basato su NuSMV

- AsmetaMA: un advisor di modelli
- AsmetaVis: un visualizzatore grafico per i modelli AsmetaL
- AsmetaRefProver: un dimostratore sulla correttezza della raffinatezza
- Asm2SMT: un traduttore dai modelli AsmetaL a Yices logical contexts
- Asm2C++: un generatore di codice [3]

2.2 Il linguaggio AsmetaL

AsmetaL è un linguaggio usato per le ASMs. Esso è una derivazione del metamodello ASMs e la sua notazione testuale è usata dai modellisti per scrivere modelli in ASM nel framework Asmeta. Il linguaggio AsmetaL può essere diviso in circa quattro parti:

- Il linguaggio strutturale, che fornisce i costrutti necessari per descrivere la struttura dei modelli ASM.
- Il linguaggio di definizione, che fornisce le definizioni base come funzioni, domini, regole e assiomi caratterizzanti le specificazioni algebriche.
- Il linguaggio dei termini, che fornisce tutti i tipi di espressioni sintattiche che può essere valutato in uno stato di una ASM.
- Il linguaggio dei comportamenti oppure quello delle regole, che fornisce una notazione per specificare le regole di transizione di un ASM [4].

Può adesso essere interessante vedere un esempio di codice sul modello ASM, che sarà utile anche per vedere il funzionamento del Web Editor, senza quindi entrare troppo nel dettaglio. Il testo di esempio per il fattoriale ed una breve descrizione è riportato nella pagina successiva.

```

asm factorial

import ../STD/StandardLibrary

signature:
  dynamic monitored value: Integer
  dynamic controlled index: Integer
  dynamic controlled factorial: Integer
  dynamic controlled outMess: Any
  derived continue: Integer -> Boolean

definitions:

  function continue($i in Integer) =
    $i>1

  macro rule r_factorial =
    if(continue(index)) then
      seq
        factorial := factorial*index
        index := index-1
      endseq
    endif

  main rule r_Main =
    seq
      if(index=1) then
        if(value>0) then
          par
            index := value
            factorial := 1
            outMess := "Executing the factorial"
          endpar
        else
          outMess := "Insert a value greater than zero"
        endif
      endif
      r_factorial[]
    endseq

default init s0:
  function index = 1

```

Il modello ASM è così strutturato in quattro sezioni: un header (intestazione), un body (corpo), una main rule (regola principale) e una initialization (inizializzazione). Il codice come vediamo comincia con *asm factorial* il quale deve essere uguale al nome del file dove è contenuto, che in questo caso dovrà quindi essere factorial.asm. Per quanto riguarda le quattro sezioni sono così individuate:

Header:

```

import ../STD/StandardLibrary

signature:
  dynamic monitored value: Integer
  dynamic controlled index: Integer
  dynamic controlled factorial: Integer
  dynamic controlled outMess: Any
  derived continue: Integer -> Boolean

```

Body:

```
definitions:

function continue($i in Integer) -
    $i>1

macro rule r_factorial -
    if(continue(index)) then
        seq
            factorial := factorial*index
            index := index-1
        endseq
    endif
```

Main rule:

```
main rule r_Main -
    seq
        if(index=1) then
            if(value>0) then
                par
                    index := value
                    factorial := 1
                    outMess := "Executing the factorial"
                endpar
            else
                outMess := "Insert a value greater than zero"
            endif
        endif
        r_factorial[]
    endseq
```

Initialization:

```
default init s0:
    function index = 1
```

Dopo aver illustrato questo esempio, non è scopo di questo elaborato andare avanti nel dettaglio di questo linguaggio, quindi per ogni approfondimento su qualsiasi argomento si rimanda alla bibliografia, dove questi argomenti vengono definiti in modo migliore [5].

2.3 Il framework Xtext

Xtext è un framework per lo sviluppo di linguaggi di programmazione e linguaggi di uno specifico dominio. Esso fornisce un'infrastruttura per quanto riguarda il controllo sintassi, il parsing, il linker e compilatore, fornendo anche supporto per la modifica di Eclipse e Web Editor [6].

Xtext, come si può ben capire, fornisce tutti i servizi necessari per sviluppare il progetto del Web Editor. Infatti, per gli scopi di questo elaborato, partendo dai requisiti più elementari, è necessario avere una “casella” di testo in una pagina web su cui poter scrivere del testo, e se si scrivono determinate parole chiave per il linguaggio, nel nostro caso Asmeta, evidenziare quella parola, indicando anche se la sintassi, cioè l’insieme delle parole, ha senso per il linguaggio.

È però necessario fare una precisazione: per lo scopo della tesi, non sarà trattata a dettaglio la parte riguardante la definizione del linguaggio Asmeta, essendo già disponibile un plug-in Eclipse a tale scopo. In questo caso si cercherà quindi di adattare questo progetto già esistente all’ambito web.

Capitolo 3

Fasi dell'implementazione Web

Questo capitolo è dedicato alla descrizione più tecnica di tutto il progetto, presentando passo a passo le operazioni di rilievo effettuate per la realizzazione del progetto, commentando ed esponendo i problemi maggiori, alcuni dei quali hanno comunque inciso in modo rilevante le tempistiche. Certe precisazioni sembreranno forse inutili e anche banali, però sono personalmente rilevanti in quanto molti intoppi sono venuti comunque da fasi in cui normalmente non si pensa di avere problemi. Problemi che in alcuni casi sono stati “risolti” se così si può dire, adottando “soluzioni” drastiche, cioè cambiando computer, oppure come a volte viene definito dagli informatici, cambiando il calcolatore elettronico.

3.1 Semantica, sintassi e configurazione

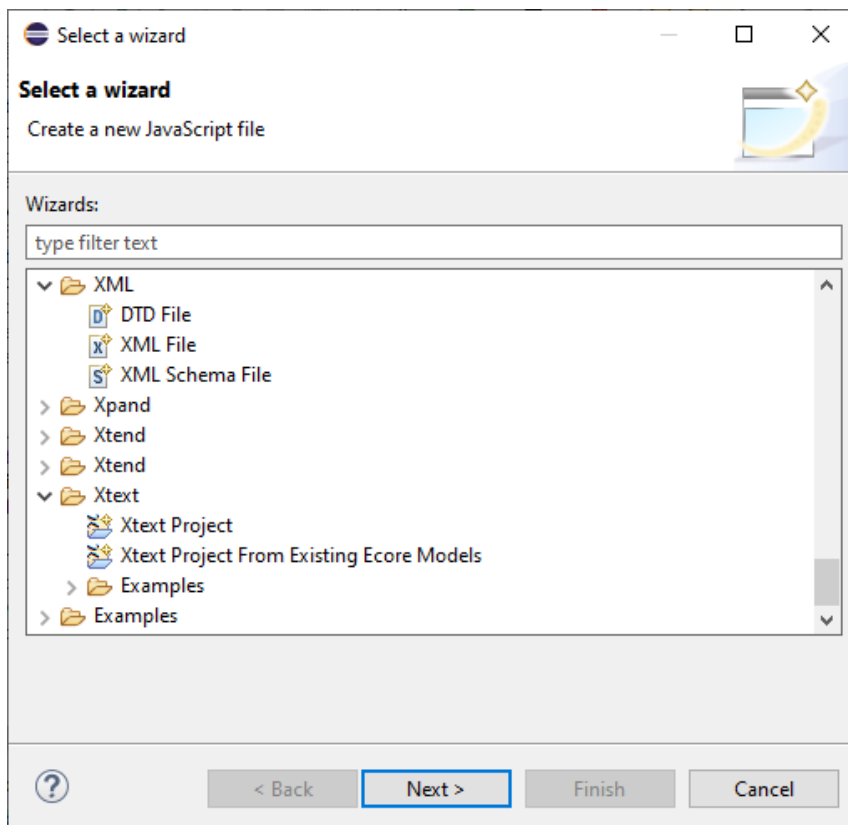
In questo paragrafo è spiegato l'inizio del progetto, per arrivare a come adattare i file del plug-in Eclipse già esistente ed in questo modo riuscire ad avere un editor web che possa evidenziare le parole chiave del linguaggio Asmeta, come ad esempio *asm* oppure *function*.

3.1.1 Installazione di Xtext su Eclipse e generazione progetto

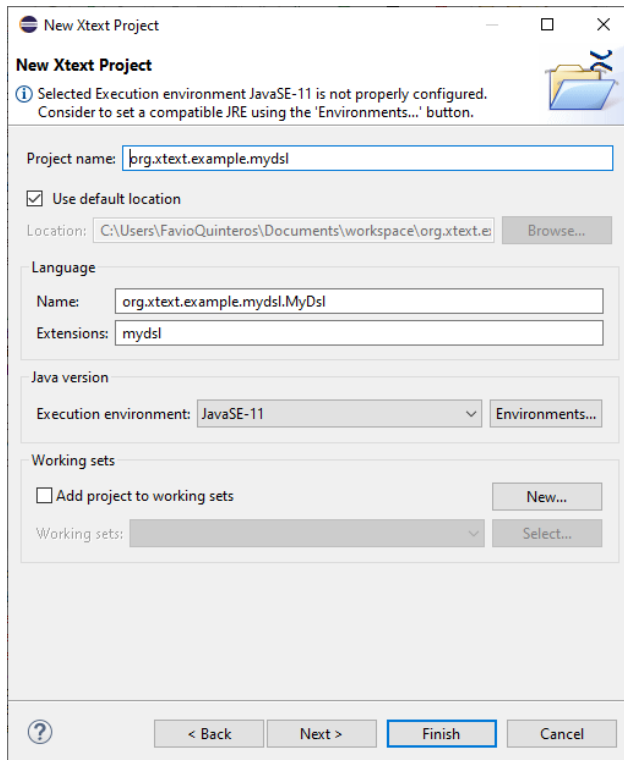
Dopo la breve presentazione su Xtext, viene qui illustrato l'installazione di Xtext su Eclipse, e la creazione del progetto su cui si andrà ad agire. Prerequisito fondamentale è ovviamente avere già Eclipse for Java Developers. Se capita già fin dall'inizio che non si avvia nemmeno Eclipse, le soluzioni di solito possono essere di due tipi: o serve l'installazione di Java sul proprio computer (infatti Eclipse “usa” la Java Virtual Machine), oppure bisognerà modificare in base all'errore sorto, un file di configurazione di Eclipse, che quindi va a modificare le impostazioni di lancio del programma.

Per l'installazione di Xtext, è sufficiente andare nella pagina di download di Xtext, scegliere una versione tra quelle disponibili copiando l'URL che li indirizza. Sarà poi quell'URL a permetterci fare l'installazione, infatti su Eclipse bisognerà andare nella sezione *Help>Install New Software>*incollare URL precedentemente copiato, quindi seguire le indicazioni proposte.

Una volta che si ha a disposizione Xtext su Eclipse, bisogna andare a creare un nuovo progetto, in modo simile a come si crea un progetto Java, quindi *File>New>Other*:



Da qui si seleziona *Xtext Project* > *Next* e dovrebbe apparire la seguente pagina:



New Xtext Project

Selected Execution environment JavaSE-11 is not properly configured.
Consider to set a compatible JRE using the 'Environments...' button.

Project name:

☒ Use default location

Location:

Language

Name:

Extensions:

Java version

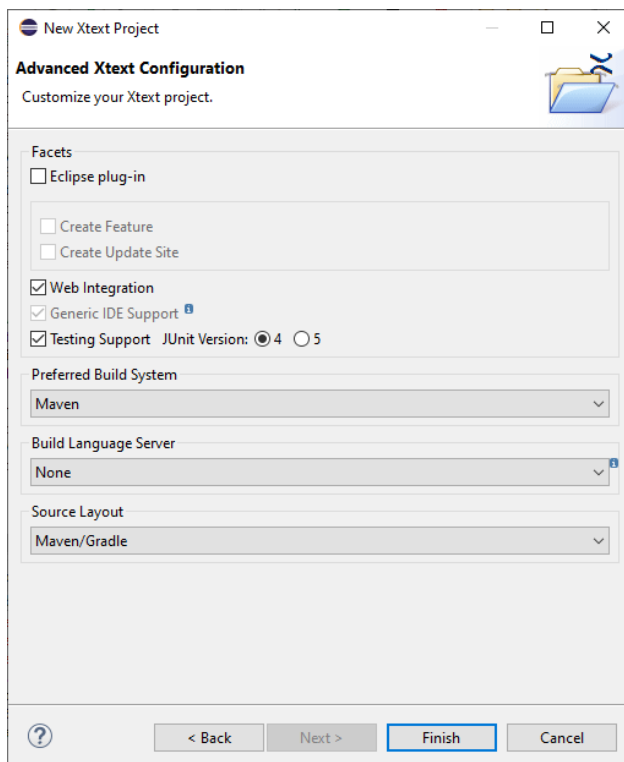
Execution environment:

Working sets

☐ Add project to working sets

Working sets:

Da qui è possibile modificare il nome del progetto e anche il nome dell'estensione, cioè il nome del linguaggio che vogliamo implementare. Una volta modificato, bisogna andare avanti con *Next* ed arrivare alla seguente pagina:



Advanced Xtext Configuration

Customize your Xtext project.

Facets

☐ Eclipse plug-in

☐ Create Feature

☐ Create Update Site

☒ Web Integration

☒ Generic IDE Support

☒ Testing Support JUnit Version: ☒ 4 ☐ 5

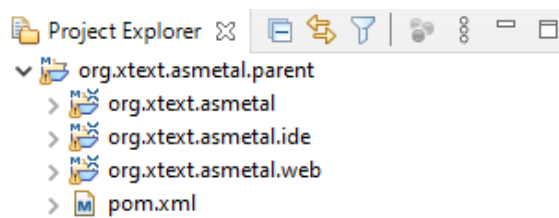
Preferred Build System

Build Language Server

Source Layout

Qui possiamo indicare cosa vogliamo includere nel nostro progetto. Interessandoci solo l'ambito web, è necessario quindi selezionare solamente la casella con *Web Integration*.

Per quanto riguarda il Preferred Build System, è stato scelto Maven il quale permette di semplificare il processo di costruzione. Tramite Maven (che fa parte della Apache Foundation), è possibile compilare il codice sorgente, fare il packaging dei codici compilati in file JAR ed installarli in una repository locale. L'ultimo passo è quindi *Finish* e dovrebbero essere generate le seguenti cartelle:



Prima di continuare con il progetto e su come farlo partire, è doveroso spiegare prima cosa è *localhost:8080*. *Localhost:8080* nei sistemi Windows è fondamentale quando si tratta di dover lanciare un programma lato server, e riuscire in questo modo a fare dei test sul funzionamento. Spiegato in maniera più pratica, *localhost* è più semplicemente un indirizzo locale, quindi ogni volta che sul browser si va su tale indirizzo, il computer “chiama” sé stesso, facendo sia da client che da server allo stesso tempo. Questo permette anche di non dover spendere risorse inutilmente, soprattutto quando non si dispone, ad esempio, di un dominio internet.

Fatta la precisazione, possiamo andare avanti sul progetto, che, per adesso è quello generato automaticamente dal Wizard di Xtext. Infatti, quello già presente è capace di eseguire il classico Hello World. Per visualizzare quindi l'editor già generato automaticamente, è necessario eseguire i seguenti passaggi (che ovviamente valgono per ogni tipo di progetto del genere):

- fare click destro sulla cartella .parent>Run As>Maven install
- fare click destro sulla cartella .web>Maven build.... e su Goals digitare *jetty:run* ed infine cliccare su Run.
- andare su un qualsiasi browser web e digitare localhost:8080

Per comodità e brevità definiremo questi passaggi appena illustrati come *eseguire il progetto*.

Questo dovrebbe essere il risultato:



Si può notare come *Hello* viene evidenziato e non viene rilevato nessun errore di sintassi.

È possibile che durante la fase di Maven install si riporti un errore di “test failure”. Per risolvere tale problema è necessario andare ad agire sul file pom.xml contenuta nella cartella .parent aggiungendo la seguente porzione di codice, che permette di ignorare l’errore prima citato e quindi andare avanti:

```
<configuration>
  <testFailureIgnore>true</testFailureIgnore>
  <useSystemClassLoader>false</useSystemClassLoader>
</configuration>
```

Ora che abbiamo generato la struttura base del nostro progetto possiamo andare a modificarlo per definire il linguaggio Asmeta.

3.1.2 Il file .xtext

Il file xtext è presente nella cartella principale, cioè quella senza l’estensione .ide ne .web. Per essere più precisi è all’interno della cartella *main*. Questo file è importante in quanto definisce la sintassi e semantica del linguaggio. Grazie a questo possiamo quindi definire quali sono le parole chiave che devono essere messe in evidenza e quali sono le regole da seguire per avere una “frase” che per il linguaggio Asmeta abbia senso.

Come è già stato accennato, la struttura del codice .xtext era già esistente da un progetto di un plug-in per Eclipse, quindi a parte l'intestazione contenente il percorso della cartella su cui è contenuto, è stato sufficiente copiare il resto del file.

3.1.3 Il file .mwe2

Partendo da dove questo è allocato come in precedenza, il file .mwe2 ha lo stesso percorso di quello è stato presentato nella sezione antecedente. Quello su cui agisce il file sono le configurazioni per quanto riguarda il progetto in generale. Più esattamente è un generatore che può essere configurato esternamente, in modo dichiarativo. L'utente di questo modo può descrivere la composizione di oggetti in modo arbitrario tramite una sintassi semplice e concisa che permette di dichiarare istanze di oggetti, valori di attributi e riferimenti [7].

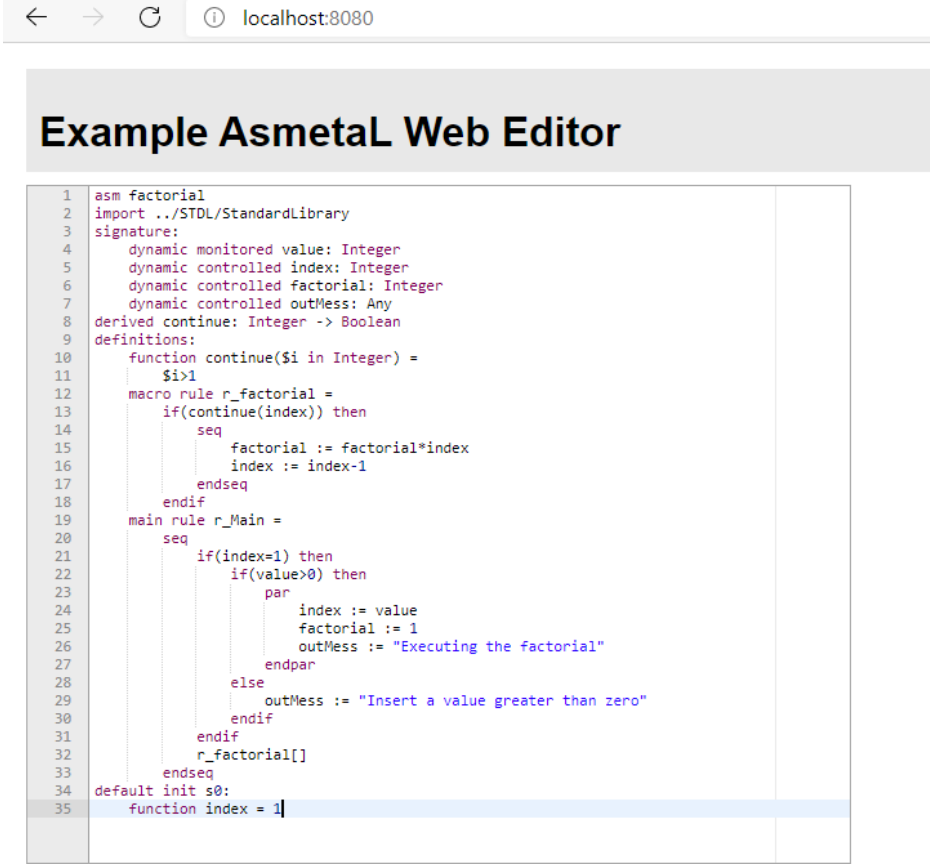
Di seguito viene qui riportato una porzione di codice che risulta fondamentale per questo progetto:

```
Workflow {  
  
    component = XtextGenerator {  
        configuration = {  
            project = StandardProjectConfig {  
                baseName = "org.xtext.asmetal"  
                rootPath = rootPath  
                runtimeTest = {  
                    enabled = true  
                }  
                web = {  
                    enabled = true  
                }  
                mavenLayout = true  
            }  
            code = {  
                encoding = "windows-1252"  
                lineDelimiter = "\r\n"  
                fileHeader = "/*\n * generated by Xtext \${version}\n */"  
            }  
        }  
    }  
}
```

Si può ben capire che da qui è possibile impostare quelli che saranno i componenti da generare. È possibile vedere le configurazioni di percorso e anche quelle di encoding, ma per questo progetto è fondamentale abilitare la parte web, quindi `web={enabled=true}`. Senza questa istruzione, infatti, viene generato un errore nel momento in cui si va a fare un *Maven install*.

3.1.4 Risultato intermedio

Dopo aver fatto le opportune modifiche ai file .xtext e .mwe2, il progetto dovrebbe ora contenere tutto ciò di cui l'editor ha bisogno per poter definire la semantica e la sintassi del linguaggio Asmeta. Infatti, mandando in esecuzione il progetto, viene visualizzato l'editor. Copiando ed incollando un esempio in linguaggio Asmeta nell'editor, otteniamo il seguente risultato:



The screenshot shows a web browser window with the address bar displaying 'localhost:8080'. The main content area is titled 'Example AsmetaL Web Editor'. Below the title is a code editor with a light blue background. The code is written in Asmeta and defines a factorial function. The code is as follows:

```
1 asm factorial
2 import ../STD/StandardLibrary
3 signature:
4   dynamic monitored value: Integer
5   dynamic controlled index: Integer
6   dynamic controlled factorial: Integer
7   dynamic controlled outMess: Any
8   derived continue: Integer -> Boolean
9 definitions:
10  function continue($i in Integer) =
11    $i>1
12  macro rule r_factorial =
13    if(continue(index)) then
14      seq
15        factorial := factorial*index
16        index := index-1
17      endseq
18    endif
19  main rule r_Main =
20    seq
21      if(index=1) then
22        if(value>0) then
23          par
24            index := value
25            factorial := 1
26            outMess := "Executing the factorial"
27          endpar
28        else
29          outMess := "Insert a value greater than zero"
30        endif
31      endif
32      r_factorial[]
33    endseq
34  default init s0:
35    function index = 1
```

Come si può vedere viene riconosciuta la semantica, riconoscendo le parole chiave, inoltre non viene segnalato nessun errore per quanto riguarda la sintassi. Se si scrive qualche istruzione che non ha senso per Asmeta, viene segnalato a fianco vicino ai numeri che segnalano le righe con una X.

3.2 Progetto Web

In questo paragrafo viene presentato quello che è il cuore del progetto, cioè la parte web. Si comincia con la spiegazione del perché si è scelto una determinata tecnologia, per poi andare a spiegare quindi cosa essi sono, spiegando anche quali sono i problemi sorti con le loro relative soluzioni.

3.2.1 Scelte progettuali

Prima della discussione riguardo al progetto, è necessario prima spiegare cosa significa fare il parsing di un codice. Questo termine indica il processo di analisi della sintassi per quanto riguarda la struttura dell'intero codice, quindi, ad esempio, per quanto riguarda Asmeta, la prima riga deve sempre essere *asm* seguita dal nome del file in cui è contenuto il codice. Quindi in parole più semplici, il parser non si limita a fare un controllo sintattico riga per riga, ma fa un controllo per quanto riguarda l'interezza del codice.

Riprendendo l'analisi del progetto, ora che abbiamo un editor web di base, c'è bisogno di implementare alcune funzionalità, come il parsing per l'appunto. Dando un'occhiata nel file system del progetto con estensione .web è stato notato che vi era un file di nome *index.html* che si occupava della visualizzazione dell'editor. Essendo un file con nome *index* si trattava quindi del file che veniva automaticamente chiamato dal server quando veniva composto l'URL *localhost:8080* nel web browser.

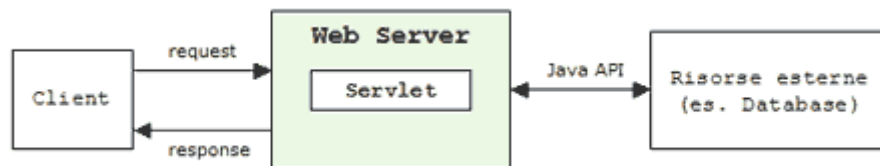
Visto che nel plug.in Eclipse di Asmeta era già presente una funzione Java che faceva da parser al progetto, il mio relatore assieme ai suoi collaboratori hanno suggerito di implementare una JavaServer Pages (JSP). Adesso è quindi necessario cosa è una JSP e quali sono i suoi vantaggi.

Le JSP sono una tecnologia sviluppata da Oracle e provvede a fornire servizi semplificati per la creazione di pagine web dinamiche, indipendentemente dalla piattaforma usata. In parole povere possiamo anche definire le JavaServer Pages come un html arricchito di funzionalità, essendo di base molto simili. Il vero vantaggio delle JSP oltre alla sua indipendenza, è però quella di permettere la separazione tra la parte di presentazione e quella di controllo. Tralasciando il fatto che nel progetto non è presente

una logica di accesso ai dati, quindi ad esempio un data-base, potremmo definire che tramite le JSP noi possiamo rispettare il pattern Model View Controller. Questo in ambito informatico, soprattutto per quanto riguarda l'ingegneria del software, è fondamentale, in quanto per grandi progetti è necessario dividere i compiti tra i vari programmatori.

È stata spiegata l'importanza di separare la presentazione dal controllore, ma non è stato spiegato come questo veramente accada nel progetto. Per questo motivo è necessario introdurre il concetto di Servlet. Spiegato in modo semplice, la Servlet è un codice in Java che risiede sul server in grado di gestire richieste dai client. Questi sono tipicamente collocati all'interno di Application Server, come ad esempio, Tomcat.

Le Servlet comunicano con il client attraverso il protocollo http. Interessante può essere illustrare la comunicazione tra client e Servlet:



Il server al momento di una richiesta (request) da parte del client di una servlet, se è la prima volta, allora istanzia la servlet e avvia un nuovo thread per gestire la comunicazione, mentre nel caso non sia la prima richiesta, allora la non serve ricaricare la Servlet e quindi per ogni nuovo client viene generato un nuovo thread. Successivamente il server invia alla Servlet la richiesta inviata dal client, così che possa elaborare la risposta ed inviarla al server. Infine, il server invia la risposta al client [8].

Ora che dovrebbe essere chiaro in linea generale cosa è una JSP e cosa una Servlet, è possibile definire i ruoli che giocano queste due tecnologie nel progetto. Infatti, come abbiamo visto, le JavaServer Pages sono essenzialmente html che permettono una integrazione di codice Java, quindi si occupano della presentazione, di ciò che l'utente vede. Mentre per le Servlet, sono il codice Java che si occupano dell'elaborazione, quindi la parte di controllo. Quello che verrà fatto successivamente sarà dunque far comunicare queste due tecnologie, separando la View dal Controller, anche se in realtà al momento di compilazione, le JSP vengono trasformate in Servlet.

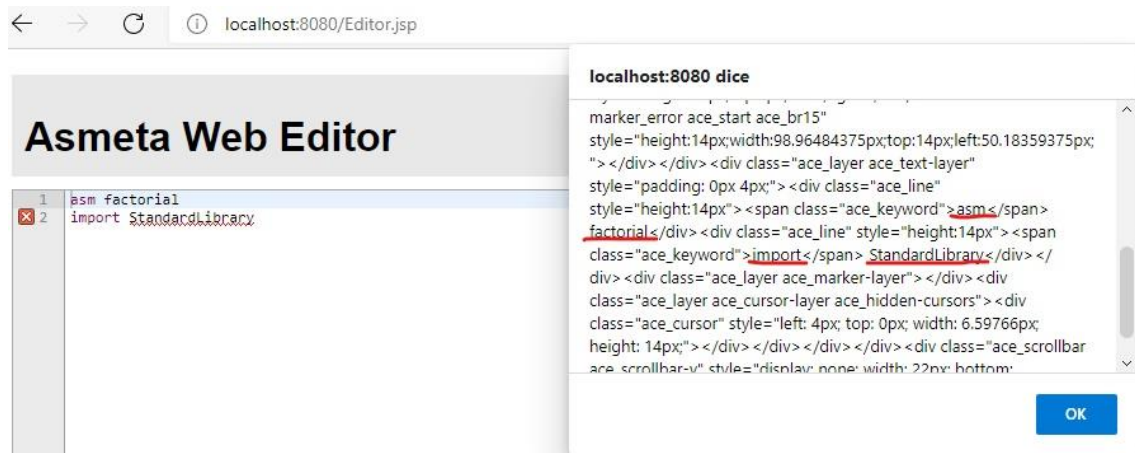
3.2.2 Input del testo tramite JavaScript

JavaScript è un linguaggio di programmazione per lo sviluppo delle web applications. È un linguaggio di scripting lato client usato per rendere interattive le pagine. Essendo lato client la correttezza del codice dipende dal web browser usato, essendo alcuni metodi non presenti in tutti i tipi di browser. È uno dei linguaggi più usati per la programmazione front-end. Prima di continuare, si precisa che, dato che si sta usando la JSP, è stato creato un nuovo file *Editor.jsp* a cui è stato copiato il codice html contenuto in *index.html* che conteneva l'editor web. Per quanto riguarda invece quest'ultimo file, è stato cancellato tutto il codice per sostituirlo con un semplice re-direct all'indirizzo *localhost:8080/Editor.jsp*, di modo che andando su *localhost:8080* si veniva subito reindirizzati a tale indirizzo. Purtroppo, non è stato possibile modificare da *Editor.jsp* a *index.jsp*, in quanto ogni volta che si andava a fare un *Maven install*, si andava a generare un nuovo *index.html* che non permetteva di aprire direttamente *index.jsp*.

Nell'ambito del progetto, JavaScript è stato usato per ricavare l'input testuale che l'utente digita all'interno dell'editor. Il metodo usato è il seguente:

```
var cod = document.getElementById("xtext-editor").innerHTML;
```

Che permette di ricavare per la variabile *cod* il seguente testo:



Il valore di *cod* è quello a destra, che, come si vede, è pieno di codice html, con all'interno però il testo in input (sottolineato). È stato dunque necessario creare una funzione in JavaScript capace di "filtrare" la variabile *cod* in modo da ottenere soltanto il testo, che è quello che serve. Essendo abbastanza corposo il codice non verrà qui riportato.

Altra funzione importante da citare, importante per il progetto, è quella di lettura dei cookies. L'argomento dei cookies verrà spiegato meglio più avanti, però, essendo in JavaScript è giusto comunque citare il metodo capace di recuperare i cookies, quindi informazioni. Il codice è il seguente:

```
function getCookie(name) {  
    const value = `; ${document.cookie}`;  
    const parts = value.split(`; ${name}=`);  
    if (parts.length === 2){  
        var cookiesString = parts.pop().split(';').shift();  
        var counter = 0;  
        var risultato = "";  
        while (counter < cookiesString.length){  
            if(cookiesString.charAt(counter) == "+") risultato = risultato + " ";  
            else risultato = risultato + cookiesString.charAt(counter);  
            counter ++;  
        }  
        return decodeURIComponent(risultato);  
    }  
    else return "";  
}
```

Tramite la prima riga è possibile ricavare in modo “grezzo” il valore dei cookie. Successivamente in base al nome si prende solo il cookie di interesse, per poi finire con un ciclo che converte i caratteri “+” in spazi “ ”, dovuto al fatto che i cookies non supportano certi caratteri, tra cui gli spazi vuoti, quindi è necessario fare un encoding prima di salvarli, per poi “decodificarli” quando li si legge.

3.2.3 Comunicazione JSP-Servlet

Prima è stato illustrato il modo in cui ottenere il testo di input all’editor. Inoltre, l’editor è adesso contenuto dentro il file Editor.jsp. Ora serve inviare il testo ottenuto con JavaScript alla Servlet che si occuperà di elaborarlo, facendo il parsing. A questo scopo sono utili le seguenti istruzioni in JavaScript:

```
document.formname.key.value = tes;  
document.formname.submit();
```

Insieme al seguente codice da implementare nel codice JSP:

```

<!-- parte che collega alla servlet -->
<form name="formname" action="EditorServlet" method="post">
    <input type="hidden" name="key">
</form>

```

Nel quale il valore di *tes* è il testo da passare alla Servlet. Mentre per quanto riguarda la Servlet:

```
String text = request.getParameter("key");
```

In questo modo è possibile ottenere nella Servlet la stringa di testo in input dall'editor, che potrà quindi essere poi elaborata per il parsing.

Riassumendo, quando l'utente digita l'indirizzo *localhost:8080*, il client carica automaticamente il file *index.html*, ma questo contiene un reindirizzamento a *localhost:8080/Editor.jsp*. Questa JavaServer Pages contiene la visualizzazione dell'editor, quello che era in precedenza *index.html*. Su un altro file sulla stessa cartella è contenuta però un file JavaScript che contiene la funzione necessaria leggere il testo in ingresso e mandarlo alla Servlet. Manca quindi un pulsante, da implementare nella JSP che una volta premuto, mandi in esecuzione la funzione JavaScript.

3.2.4 Implementazione bottone di esecuzione

Come è stato accennato prima, a questo punto manca un pulsante che mandi in esecuzione la Servlet una volta che l'utente compone il codice Asmeta nell'editor. A questo scopo è necessario modificare la JSP e implementare oltre al pulsante, anche una console dove poter visualizzare i risultati:

```

<div class="container">
    <div class="header">
        <h1>Asmeta Web Editor</h1> <!-- titolo -->
    </div>
    <!-- parte dell'editor -->
    <div class="content"><div id="xtext-editor" data-editor-xtext-lang="asm">
        <span id="myText"></span></div></div>
    </div>
    <div id="titolo_console"><h1>Console:</h1></div>
    <!-- box della console-->
    <div id="secontainer"><h5><span id="tempo"></span>
        <span id="out" style="font-family:Courier"></span></h5></div>

    <!-- bottone -->
    <div class="run">
        <button onclick="esegui()"></button>
    </div>

```

La prima parte di codice riguarda quello che è l'editor. La parte interessante riguarda l'aggiunta di un secondo riquadro, necessario per contenere i risultati del parsing, fatto attraverso l'istruzione `<div id="secontainer">`. Il bottone invece sarà generato dall'istruzione `<button>`. Per mandare in esecuzione la funzione JavaScript è necessario aggiungere all'interno di `<button>`, `onclick="esegui()"`. In questo modo, quando l'utente premerà il bottone, verrà eseguita la funzione `esegui()`, che quindi farà la lettura del testo all'interno dell'editor, per poi mandarlo alla Servlet. Ottenuto il testo da parte della Servlet manca la parte dell'elaborazione.

3.2.5 Il Controller ed il parsing

Come accennato prima, la Servlet legge i parametri passati attraverso `getParameter()`. Il passaggio del controllo avviene però attraverso il `<form>` contenuto nella JSP, tramite il quale è possibile scegliere quale metodo della Servlet chiamare. Le scelte sono due: o scegliere di invocare il `doGet`, usato principalmente quando la quantità di parametri da inviare non è elevata (passati solitamente tramite URL), oppure `doPost` che invece è usata per quantità di dati più sostanziosi. In questo caso tramite `method="post"` è stato scelto di invocare il metodo `doPost` della Servlet, per non essere troppo vincolati.

Ora che si sa come passare i parametri, possiamo vedere la struttura della Servlet. Nella `form` di chiamata, non è stato definito però come avvenisse il collegamento. Infatti, tramite `action="EditorServlet"` non si sa ancora cosa sia `EditorServlet`. Affinché si possa raggiungere la Servlet, si dovrà aggiungere al codice Java subito dopo tutti gli import, la dicitura:

```
@WebServlet(name = "EditorServlet", urlPatterns = {"/EditorServlet"})
```

In questo modo il collegamento avviene tramite il nome. Per poter usare tale istruzione è necessario, però, aggiungere un'importazione all'inizio tramite: `import javax.servlet.annotation.WebServlet;`

È possibile ora procedere con i passaggi che riguardano l'elaborazione del testo passato come parametro. Importante accennare che quando si crea una classe Java che faccia da Servlet, essa deve estendere `HttpServlet`, essendo, come è stato spiegato. A questo scopo anche qui è necessario aggiungere all'inizio gli import corretti come ad esempio `import`

javax.servlet.http.HttpServletRequest; assieme ad altri che non saranno riportati su questo elaborato. Fondamentale, perché come è stato spiegato, le Servlet comunicano con il client attraverso delle richieste http.

Ottenuto il testo per il controller, è possibile procedere per elaborare tale stringa tramite il metodo già presente. Per importare tale funzionalità è necessario però prima scaricare il pacchetto *AsmetaS.jar* per poi andare nella cartella *.web* principale del progetto, fare tasto destro su di esso > *Build Path* > *Configure Build Path* > *Libraries* > *Classpath* > *Add External JARs*, e selezionare *AsmetaS.jar*. Successivamente, occorre importare nel file Java il metodo, attraverso `import org.asmeta.parser.*; .`

I metodi interessanti per il progetto sono:

- *ASMParser.setUpReadAsm(file)* il quale premette di fare il parsing del file in ingresso, restituendo un oggetto *asmetaa.AsmCollection*.

- *ASMParser.getResultLogger()* il quale permette di capire se ci sono errori. Restituisce un oggetto di tipo *ParseResultLogger*, che avrà una dimensione nulla nel caso in cui non vi siano errori nel parsing,

- *e.getMessage()* in cui *e* è l'eccezione che viene invocata nel caso di errore nel parsing. In questo modo si potrà prendere quello che sono gli errori generati dal parser.

Analisi della porzione di codice riguardante il parsing:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    saveStandardLibrary();
    //
    String parseResult = "";
    String text = request.getParameter("key");
    if (text == null) {
        parseResult = "empty asm";
    } else {
        // estrai il nome della asm
        Matcher m = p.matcher(text);
        if (m.find()) {
            String asmName = m.group(0).substring(4).trim();
            //
            File codice = new File(asmName + ".asm");
            if (codice.exists()) {
                codice.delete();
            }
            // save to a new file the asm
            codice.createNewFile();
            PrintWriter scrivi = new PrintWriter(codice);
            scrivi.print(text);
            scrivi.close();
            // call the parser
            try {
                @SuppressWarnings("unused")
                asmata.AsmCollection asms = ASMParser.setUpReadAsm(codice);
                ParserResultLogger resultLogger = ASMParser.getResultLogger();
                if (resultLogger.errors.size() == 0)
                    parseResult = "No errors found";
            } catch (Exception e) {
                parseResult = e.getMessage();
            }
        } else {
            parseResult = "Must declare asm \"name\"";
        }
    }
}
```

In ordine si può osservare che:

- saveStandardLibrary() è una funzione statica della stessa classe che si occupa di verificare se sia presente nel progetto la StandardLibrary.asm. Se questa è già presente non fa nulla, mentre se manca, allora la recupera, copiandola da una specifica cartella del progetto.
- La stringa *text* contiene ora il valore di testo inserito nell'editor.
- Se *text* è vuota, si segnala subito che il file asm è vuoto, quindi è inutile proseguire con la funzione di parsing.
- Se invece *text* non è vuoto, attraverso *Matcher m* e la successiva condizione *if*, si vuole capire è presente la dicitura *asm "nome"*, quindi se è presente, allora selezionare tale nome che andrà a definire il file .asm.
- Se non viene trovata nessuna corrispondenza, allora si scrive su *parseResult* che si deve dichiarare un *asm "nome"*.
- Riprendendo il caso di una corretta dichiarazione dell'asm, prima di creare il file .asm si controlla se non vi sia uno uguale con lo stesso nome. Se presente, lo si cancella, e quindi si va avanti creando un nuovo file, scrivendoci sopra la stringa *text*.

-Ottenuto il file .asm, si procede col fare il parsing di tale file, attraverso *ASMParser.setUpReadAsm(file)* e *ASMParser.getResultLogger()* che sono stati spiegati in precedenza, in cui nel caso di un parsing senza errori, allora scrive nella stringa del risultato (*parseResult*) che non vi sono errori. In caso contrario, durante l'esecuzione del parsing, verrà sollevata un'eccezione che scriverà in *parseResult* il report di errori.

A questo punto, *parseResult* dovrebbe contenere il risultato del parsing, segnalando se non vi sono errori, se manca asm "nome", e nel caso di errore, segnalare quali essi sono. L'idea in origine è stata quella di, una volta dato il controllo alla Servlet, fare l'elaborazione e successivamente ritornare il controllo alla JSP, mandandogli il valore di *parseResult*. Attraverso l'implementazione della funzione nella Servlet di:

```
request.setAttribute("message",parseResult)
```

e di:

```
request.getRequestDispatcher("/Editor.jsp").forward(request,response)
```

ed anche di:

```
${message}
```

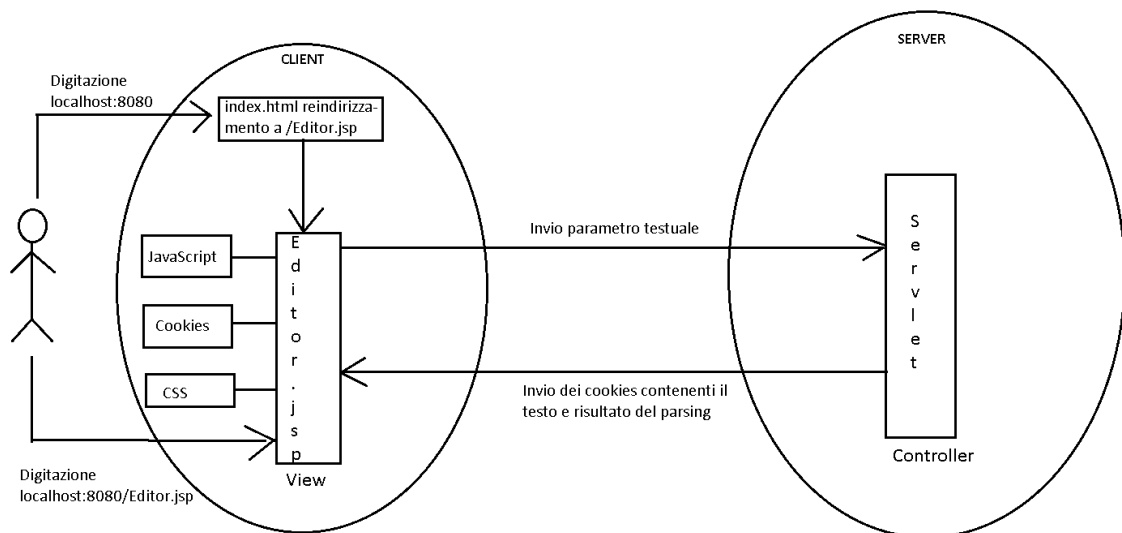
nella JSP

Con queste istruzioni si dovrebbe passare il parametro di *parseResult* alla presentazione, ritornando (con *getServletContext()*) il controllo alla JavaServer Pages. Quello che accade è invece, sì, il passaggio del parametro alla presentazione, quindi viene visualizzato il risultato, però purtroppo, la JSP non riprende la sua esecuzione, quindi l'editor non viene reso più attivo, non più modificabile. Per un Web Editor, il fatto che dopo aver avviato tramite bottone il parsing, l'editor stesso non sia più modificabile, è ovviamente non accettabile per i requisiti intrinseci. Molto probabilmente questo succede perché non c'è un mapping tra JSP e Servlet, vale a dire, in parole semplici, che non vi è una stretta dipendenza tra queste due. Per implementare tale dipendenza, sarebbe necessario introdurre un file web.xml che definisca le relazioni dei file presenti nel progetto. Nel progetto è già presente un file simile nel nome, si tratta del pom.xml, ma internamente è molto diverso, in quanto specifico per l'ambiente Maven. Passando al lato personale, avendo non avendo molta esperienza in ambito web, non avevo fiducia sul fatto di riuscire a modificare le dipendenze, essendo il progetto per quanto riguarda lo "scheletro", dipendente dalle impostazioni di Xtext.

Dopo tante ricerche, è stata trovata una possibile soluzione, cioè quella di usare i cookies. Questi non sono altro che delle informazioni che fisicamente risiedono nel client. Ogni qual volta il server ha bisogno dei cookies, essa invia una richiesta al client, che risponde inviandogli i dati richiesti. Il server a sua volta può inviare al client nuovi cookies. Questo è particolarmente utile in quanto il protocollo http è privo di stato.

Implementati nel progetto, i cookies permetterebbero salvare il testo ed il risultato del parsing sul client, e quindi ogni volta che si carica la pagina, recuperare questa informazione e visualizzarlo. In questo modo, se prima ogni volta che si ricaricava la pagina, il testo sull'editor veniva perso, ora il testo verrebbe salvato rispetto alla ultima esecuzione della Servlet.

Per fissare il concetto di funzionamento del progetto, vi è il seguente schema:



Grazie a questa è possibile vedere meglio il funzionamento in generale del progetto, con gli scambi dei parametri e dove sono collocati i componenti. Non è uno schema dettagliato che copre esattamente tutto il funzionamento, ma semplicemente la meccanica di passaggio del controllo.

Ora che dovrebbe essere chiaro il tutto, occorre definire:

- per quanto riguarda il client, un meccanismo che al caricamento della pagina, vada a recuperare i Cookies, leggendoli e caricandoli nella vista della pagina, sia per quanto riguarda il testo, che il risultato dell'ultimo parsing.
- per quanto riguarda il server, che ogni volta che viene eseguito il codice Java, dopo aver fatto le opportune elaborazioni, mandi al client il testo inviatogli in

precedenza, quindi aggiornandolo, ed anche il testo contenente il risultato del parsing.

Per l'analisi del codice è più interessante prima vedere quello della Servlet:

```
@SuppressWarnings("deprecation")
String cod = URLEncoder.encode(text);
Cookie cookie = new Cookie("cookie",cod);
cookie.setMaxAge(Timeout_cookie);
response.addCookie(cookie);
System.out.println(parseResult);
@SuppressWarnings("deprecation")
String cod2 = URLEncoder.encode(parseResult);
Cookie cookie2 = new Cookie("cookie2",cod2);
cookie2.setMaxAge(Timeout_cookie);
response.addCookie(cookie2);
response.sendRedirect (URLjsp);
```

Alla prima istruzione utile si vede un comando che serve per fare un encoding alla stringa che contiene il testo. Questo è utile farlo perché i cookies non supportano certi caratteri, tra cui anche gli spazi vuoti (" "), quindi è necessario trasformare questi, in una serie di digit special. Successivamente si crea un cookie e gli si attribuisce il valore del testo codificato, si setta il suo tempo massimo di vita, per poi aggiungere alla risposta, che viene quindi inviata al client. Lo stesso avviene anche per la stringa contenente il risultato del parsing. Da specificare che per usare i cookies occorre fare *import javax.servlet.http.Cookie;* all'inizio. Per ultimo vi è il reindirizzamento alla pagina Editor.jsp.

Per quanto riguarda l'elaborazione lato client:

```
window.onload = function(){
    document.getElementById("myText").innerHTML = minmagHtml(getCookie("cookie"));
    document.getElementById("out").innerHTML = minmagHtml(getCookie("cookie2"));
    var today = new Date();
    var giorno = today.getDate();
    var mese = today.getMonth()+1;
    var minuti = today.getMinutes();
    var ora = today.getHours();
    if(giorno<10) giorno="0"+giorno;
    if(mese<10) mese="0"+mese;
    if(minuti<10) minuti="0"+minuti;
    if(ora<10) ora="0"+ora;
    var tempo = "["+giorno+"/"+mese+"/"+today.getFullYear()+" "+ora + ":" +minuti+"]";
    document.getElementById("tempo").innerHTML = tempo;
}
```


Il codice presente nella JavaServer Pages ha il compito di caricare i cookies al momento di caricamento della pagina. Le funzione *getCookie()* ha il compito di caricare i cookie, di decodificarli, quindi di tradurre i caratteri speciali, sostituendoli opportunamente, ad esempio, con gli spazi. Con la funzione *minmagHtml()* si va invece a sostituire i caratteri “<” e “>” rispettivamente con “<” e “>”, questo perché in html, e quindi anche per le JSP, sono caratteri speciali che se mandati in visualizzazione, non vengono interpretati. Attraverso l’assegnazione *document.getElementById(“myText”).innerHTML* è possibile recuperare in fase di presentazione il valore che è stato prima elaborato, quindi il valore dei cookies. Infatti, tramite ** è possibile visualizzare quello che è il testo salvato tramite cookies. Per ultimo, per motivi estetici, vi è la porzione di codice che ricava la data e l’ora, da far visualizzare ad ogni pressione del bottone.

3.2.6 Definizione dello stile in CSS

CSS, acronimo di Cascading Style Sheets, è un linguaggio che determina lo stile, quindi la formattazione delle pagine html. Si occupa quindi, esclusivamente della presentazione delle pagine web. Questo però non può lavorare da solo, ha bisogno ad esempio di un html (nel caso del progetto una JSP), che prima definisca quali saranno gli elementi presenti sulla pagina, elementi le cui caratteristiche verranno meglio definiti dal CSS. Questo file si può trovare in modo predefinito durante la *Maven install* del progetto web Xtext. In questo modo si va a definire il font dei caratteri, il colore dei bordi, i riempimenti ed il posizionamento.

Per quanto riguarda il progetto, sono stati definiti le caratteristiche per tre nuovi componenti. Il codice per quanto riguarda il titolo della console che viene visualizzata a destra dell’editor è il seguente:

```
#titolo_console {  
    display: block;  
    position: fixed;  
    background-color: #e8e8e8;  
    top: 20px;  
    left: 740px;  
    right: 0;  
    height: 60px;  
    padding: 10px;  
}
```

-display:block; definisce il modo in cui verranno rappresentati i paragrafi di testo. In pratica con questa istruzione ogni inizio di paragrafo in html sarà automaticamente mandato a capo.

-position:fixed; definisce che del titolo della console sarà sempre fissa, indipendentemente se uno si muove nella pagina.

-background-color: #e8e8e8; definisce il colore di riempimento dello sfondo che circonda il titolo, con la specificazione, tramite codifica, di quale sarà la colorazione.

-top: 20px; definisce la distanza in pixel tra il bordo più in alto dello schermo e l'oggetto in questione, quindi il titolo.

-left: 740px; definisce la distanza in pixel tra il bordo sinistro dello schermo e l'oggetto in questione, cioè il titolo.

-right: 0px; esattamente come per l'istruzione left, ma prendendo in considerazione il bordo destro.

-height: 60px; definisce la grandezza del titolo, quindi quanto questo sarà ampio in termini di altezza.

-padding: 10px; definisce in termini di pixel, la distanza che il testo contenuto all'interno di questo riquadro avrà dai bordi che lo circondano.

Per il riquadro contenente il risultato del parsing si ha:

```
#secontainer {  
    display: block;  
    position: fixed;  
    top: 110px;  
    bottom: 20px;  
    left: 740px;  
    right: 20px;  
    padding: 4px;  
    border: 1px solid #aaa;  
}
```

Le prime istruzioni sono uguali a quelle di prima. La parte interessante è l'istruzione *border: 1px solid #aaa*, che serve a definire in termini di pixel, quando sarà ampio lo spessore del riquadro che andrà a contenere il risultato del parsing. Con la parola chiave *solid* si va invece a definire il tipo di bordo, che in questo caso sarà di tipo continuo (ad esempio con *dotted* si andrebbe a definire un bordo fatto di puntini). *#aaa* definisce il colore che il bordo va ad assumere.

Per il bottone che manda in esecuzione il parsing si ha:

```
.run {  
  display: block;  
  position: fixed;  
  top: 110px;  
  bottom: 0;  
  left: 680px;  
}
```

Tutti le istruzioni di questo codice sono già state introdotte. L'unica cosa che si può notare è che la posizione del bottone rimarrà invariata nello schermo, anche se si fa scrolling nell'editor.

Per ultimo, la modifica del codice del contenitore di testo dell'editor:

```
.container {  
  display: block;  
  height: 6000px;  
  position: absolute;  
  top: 0;  
  bottom: 0;  
  left: 0;  
  right: 0;  
  margin: 20px;  
}
```

L'istruzione non ancora vista è la *position: absolute*; la quale al contrario di *fixed*, vede che l'oggetto si muove insieme allo scrolling della pagina. L'istruzione *margin* riguarda la distanza in termini di pagina, e non di bordo dello schermo come invece è per istruzioni come *bottom* o *left*. La grossa modifica apportata al progetto è però *height* il quale è stato notevolmente incrementato rispetto al valore che assumeva prima. Questo per supplire al “difetto”, se così si può definire, della funzione JavaScript `document.getElementById().innerHTML`, infatti tale funzione “prende” solamente il testo visibile dall'utente, quindi se nel riquadro dell'editor si va oltre un certo numero di righe di codice, certe righe non sono visibili (a meno di non fare scrolling), e quindi le righe non più a video non vengono considerate, generando inevitabilmente errori nel parsing.

Con questa considerazione viene chiuso quello che è il progetto nella sua parte più tecnica.

Bibliografia

- [1] Wolfgang Reisig. «Abstract State Machines for the Classroom». 2006. URL: https://www.researchgate.net/publication/318521286_Abstract_State_Machines_for_the_Classroom
- [2] URL: <https://asmeta.github.io/index.html>
- [3] URL: <https://asmeta.github.io/download/index.html>
- [4] URL: https://asmeta.github.io/material/AsmetaL_quickguide.html
- [5] URL: https://asmeta.github.io/material/AsmetaL_guide.pdf
- [6] URL: <https://www.eclipse.org/Xtext/>
- [7] URL: https://www.eclipse.org/Xtext/documentation/306_mwe2.html
- [8] URL: <https://www.html.it/articoli/primi-passi-con-le-servlet/>