Linee guida sviluppo interfaccia utente KM MFP

Informazioni generali sul device

- Il device è munito di un piccolo touch screen (mono touch) con risoluzione 800X480 px e profondità di colore 24 bit.
- Il browser che visualizzerà le pagine html è della famiglia webkit e riconosce fino ad html 4.
- E' possibile utilizzare librerie js moderne, come ad esempio jquery, twitter-bootstrap...
- Il prototipo in allegato utilizza jquery per chiamate ajax e bootstrap per layout e grafica.

Microframework KMEF(Konica Minolta Essential Framework)

Il KMEF è un framework MVC creato per semplificare lo sviluppo di app su dispositivi konika minolta. E' costituito da un URL dispatcher, un View engine, un Template engine e un data Model.

Di seguito verranno approfondite le parti importanti per l'integrazione con l'interfaccia web.

Url dispatcher

l'url dispatcher si occupa di indirizzare le richieste, contenute nella query string, alle views che le elaboreranno. Inoltre costruisce l'oggetto request definendo delle strutture a dizionario per queryString(GET) e dati in POST.

Contiene un mapping tra il nome della pagina che si sta richiamando e il nome della funzione che verrà richiamata e a cui verrà passato come parametro la Request creata precedentemente.

Esempio:

```
newUrls={
        'root':index,
        'index':index,
        'language':setLanguage,
       'login':Login,
       'logout':Logout}
```

per richiamare la pagina 'index' per esempio, sarà necessario richiamare l'url dispatcher con un parametro: page=index. Per cui se il dispatcher si chiama app.py, avremo la seguente chiamata: app.py?page=index

La parola chiave 'root' serve a richiamare una view di default, per cui se nel parametro page inserisco una pagina non presente nel mapping, richiamerà la pagina index.

Naturalmente una stessa view può essere usata per pagine diverse.

Data Model

Il data model non si riferisce ad un database, ma ai servizi remoti, serviti tramite json o soap.

Il modello serve a creare la richiesta json per un servizio remoto e a recuparare i dati del json strettamente necessari alla applicazione. Le classi del Data Model devono estendere la classe Model.

Esempio:

```
class Login(Model):


    input={
          'user':StringField('Login.User','',[Persistent]),
          'passwd':PasswordField('Password',''),
          }


    output={
           'SessionId':StringField('Login.SessionId','',[Persistent]),
           }


class User(Model):


    input={
          'user':StringField('key@Login.User',''),
          'session':StringField('key@Login.SessionId',''),
          }
```

```
output={
'Name':StringField('User.Name','',[Persistent]),
'UserType':StringField('User.Type','',[Persistent]),
}
```

In questo esempio abbiamo definito due classi che estendono Model: Login e User.

Come si può vedere sono scritte in modo diverso dalle classi python in generale, perché si è usato un approccio più dichiarativo nella definizione della classe.Naturalmente la classe deve essere istanziata in modo classico, per esempio: log=Login().

Ad ogni model deve essere associato un template che costruisce la struttura del json, prendendo le variabili dal dizionario input. La risposta del servizio remoto, invece viene memorizzata nelle variabili deldizionario 'output'. Esempio in pseudocode:

```
log=Login()
log.user.setValue('Gianluca')
log.passwd.setValue('password')
#call service servizio(log)
print(log.SessionId.getValue())
>>> te546733467346376438487388743 89
```

Non è possibile accedere direttamente ai valori delle variabili, ma bisogna richiamarli attraverso getValue() e setValue(), perché ad esse è associato un field e ai valori del field si accede solo attraverso quelle due metodi di get e set.

Nel model User è possibile notare che alcuni field hanno nome key@... Quella key@ è una astrazione del concetto di chiave esterna in un db relazionale e sta ad indicare, nel caso specifico, che il valore di quel campo è il valore ricavato dalla chiamata al model indicato,Login.

Ad esempio, nel caso di 'session':StringField('key@Login.SessionId',''), la variabile session nel model User ha il valore ricavato dal servizio di login che ritornava la sessionId. Come si può notare a tutti i field che fungono da chiavi esterne, in questo caso 'sessionId' nel model Login, viene iniettata una classe che li rende persistenti :

```
 'SessionId':StringField('Login.SessionId','',[Persistent]),
```

Questa classe salva il valore in un file.

La persistenza può essere di due tipi: salvataggio su file (hard persistence) e salvataggio nel cookie (soft persistence), entrambe ottengono lo stesso risultato, anche se quella nei cookie è più performante.

View Engine

....

Template Engine

Il Microframework supporta un semplice template engine per strutturare le pagine html.

I template hanno una struttura ad albero e possono contenere variabili.

Un esempio di pagina creata con i template è la seguente:

```
base={'template':"""
<html>
  <head>
    ${title}
    ${link}
    ${js}
  </head>
  <body>
    ${navbar}
    ${content}
    ${footer}
  </body>
</html>


"""
}


head={
       'extend':base,
       'title':"""<title> We5! Il blog della guida HTML5 </title>""",
       'link':"""<link rel="stylesheet" type="text/css" href="mystyle.css"> """,
        'js':"""<script src="web-link.js"></script> """
     }
```

```
index={
    'extend':head,
    'navbar':"""<div><h1>Navbar</h1> <h2>${nomeUtente}</h2></div>""",
    'content':"""<h2>Content </h2>""",
    'footer':"""<h3>footer</h3> """
    }
Index2={
     'content':"""<h2>Content 2</h2>""",
     'navbar':"""<div><h1>Navbar2</h1> <h2></h2></div>""",
    'footer':"""<h3>footer2</h3> """,
    'extend':head
    }
```

Come si vede dall'esempio, il template "base" fornisce la struttura della pagina html, definendo dei blocchi che saranno sostituiti dall'engine a runtime. Questi blocchi possono essere ulteriormente specializzati in altri template che estendono il template "base", nel nostro esempio "head".

"head", in questo caso, fornisce il contenuto dei soli blocchi presenti all'interno del tag <head>, per cui è necessario definire almeno un altro template che fornisca il contenuto dei blocchi rimanenti.

Nel nostro esempio abbiamo definito il template "index" che estende il template "head" e che definisce i blocchi rimanenti.

Si possono definire un numero arbitrario di template ognuno dei quali può estenderne solo 1. *vedi "index2"*

Il template principale, in questo caso "base", non avrà la parola chiave "extend", ma solo la parola chiave "template" .

Come è possibile vedere dalla definizione del template "index2", non è necessario seguire alcun ordine per la definizione dei blocchi, ma è preferibile comunque usare la parola chiave "extend" in cima alla definizione, per questioni di leggibilità.

I template, a qualunque livello, possono contenere al loro interno delle variabili, in questo caso "index" definisce una variabile "${nomeUtente}" .

Non è possibile definire "cicli" o "if then else" all'interno dei template.

Il template engine non è legato al solo file html, ma può essere utilizzato in qualunque contesto. Ad esempio creazione file json o xml o testo.

Internazionalizzazione i18n

Il microframework possiede un meccanismo per semplificare la scrittura di applicazioni multilingua.
I passi per scrivere applicazioni multilingua sono i seguenti:

1. Individuare all’interno del template tutte le frasi/parole che necessitano di traduzione;
2. Scriverle in un file di testo ed associare ad ognuna di esse una variabile che inizi con “i18n_” ,seguendo questa sintassi:
   italian={
   'i18n_user':'utente',
   'i18n_passwd':'password',
   'i18n_login':'entra',
   'i18n_cancel':'cancella',
   'i18n_btnlogin':'Entra',
   'i18n_btndownload':'Scarica',
   'i18n_btnscan':'Acquisisci',
   'i18n_btnlogout':'Esci',
   'i18n_btnprint':'Stampa'
   }

   english={
   'i18n_user':'user',
   'i18n_passwd':'password',
   'i18n_login':'login',
   'i18n_cancel':'cancel',
   'i18n_btnlogin':'Login',
   'i18n_btndownload':'Download',
   'i18n_btnscan':'Acquire',
   'i18n_btnlogout':'Logout',
   'i18n_btnprint':'Print'
   }

3. Sostituire, all’interno dei template, tutte le occorrenze delle parole/frasi individuate, con le relative variabili. Ad esempio la variabile “i18n_content”, all’interno del template deve essere scritta con questa sintassi: $ {i18n_content}.
4. Esempio:
   index={
   'extend':head,
   'navbar':""" <h1>Navbar</h1> """,
   'content':"""<h2>${i18n_content} </h2>""",
   'footer':"""<h3>footer</h3> """
   }

5. Le variabili possono comparire più volte all’interno del template, ogni occorrenza verrà sostituita dall’engine in fase di elaborazione.

Convenzioni sulle URL

Le pagine all’interno del microframework vengono create dinamicamente e vengono gestite da un URL Dispatcher, che si aspetta il nome della pagina nella querystring, in particolare nella variabile “page”. Ad esempio “app.py?page=index”, dove index è la pagina e app.py è il dispatcher.

Questa sintassi è valida solo per pagine dinamiche. Per le pagine statiche come file js, immagini etc. si procede in modo classico, perchè sono servite dal web server interno e non dal dispatcher.

Gianluca Esposito