Developers documentation

Poznań, Poland, 2006-08-08

Authors:
Maciej Gawinecki
Paweł Kaczmarek

Project site:
http://sourceforge.net/projects/e-travel

Contact:
maciej.gawinecki@ibspan.waw.pl

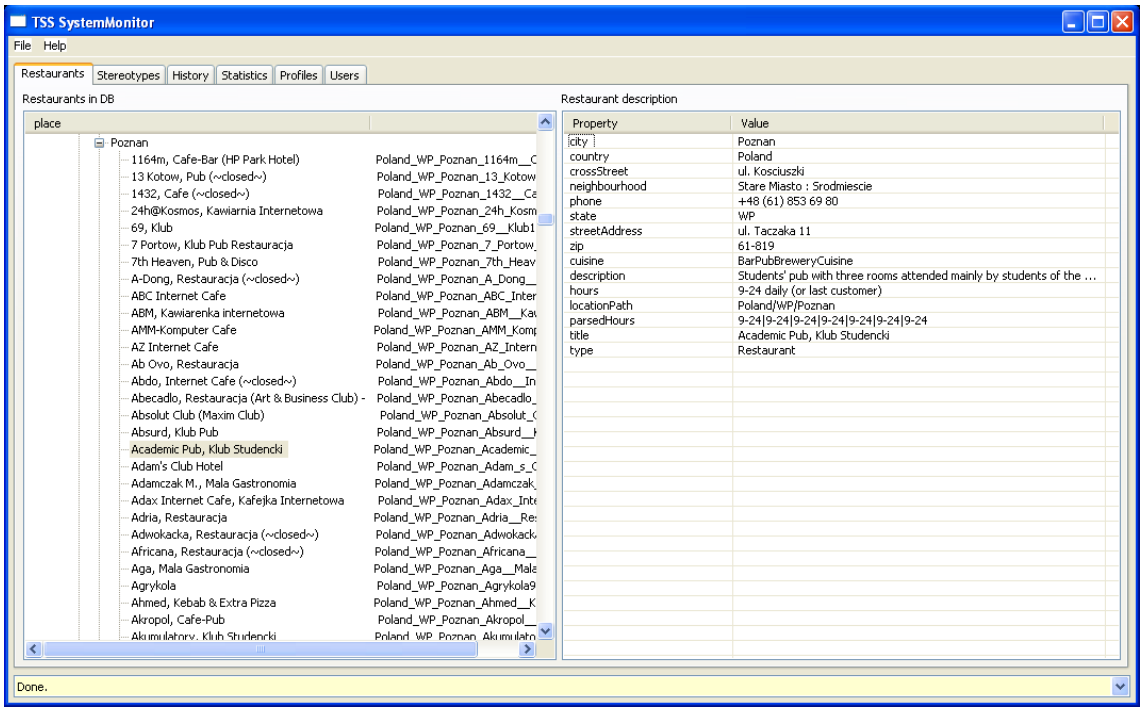
Developer platform (suggested)

GUI Tool Kit (for System Monitor tool):	SWT: The Standard Widget Toolkit 3.2 http://www.eclipse.org/swt/
Diagrams design:	yEd Graph Editor 2.3.1_02 http://www.yworks.com/en/products_yed_about.htm JUDE Community 3.0.1 http://jude.change-vision.com/jude-web/product/community.html
Ontology design:	Protege 3.2 beta http://protege.stanford.edu/ with OntologyBeanGenerator plugin http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator
Programming IDE:	Eclipse 3.2 http://www.eclipse.org/ with Visual Editor http://www.eclipse.org/vep/WebContent/main.php
XML editor:	Altova XML Spy 2006 Home Edition http://www.altova.com/products/xmlspy/xml_editor.html
Text editor:	EditPlus v2.12 http://www.editplus.com/

Additional Tools

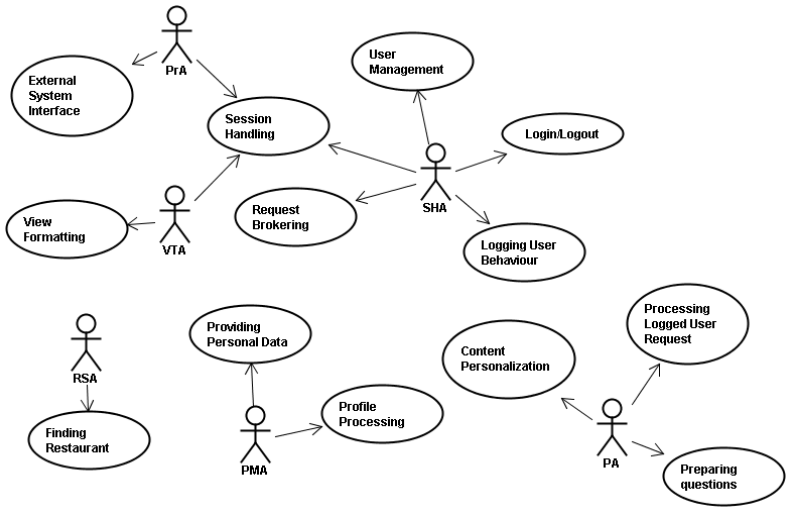
System Monitoring

Helps to track what is going on with:



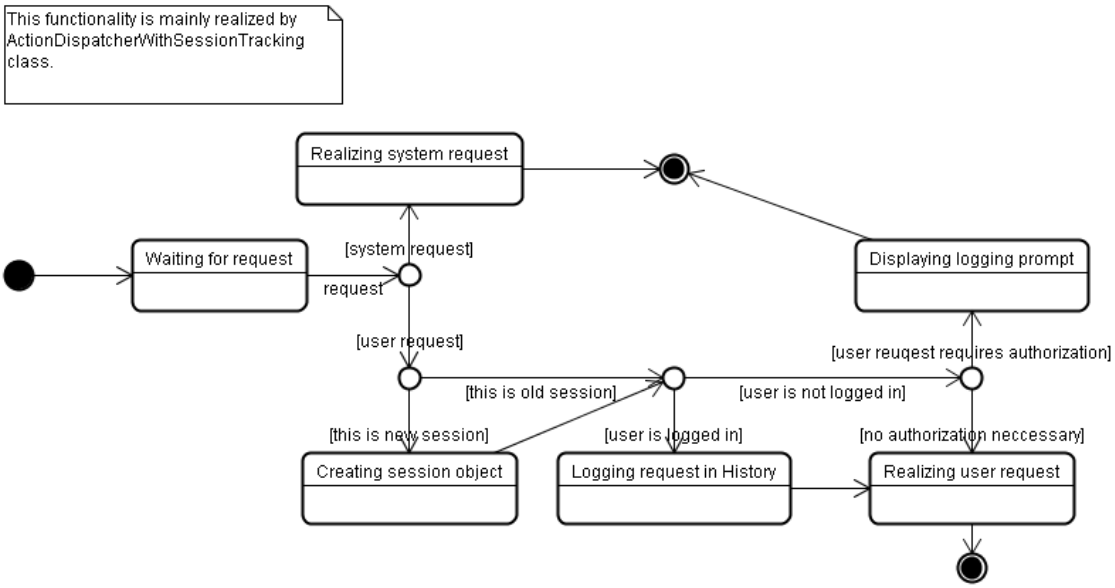
Can be launched be calling *bin\monitor-tss.bat* script.

Use Cases

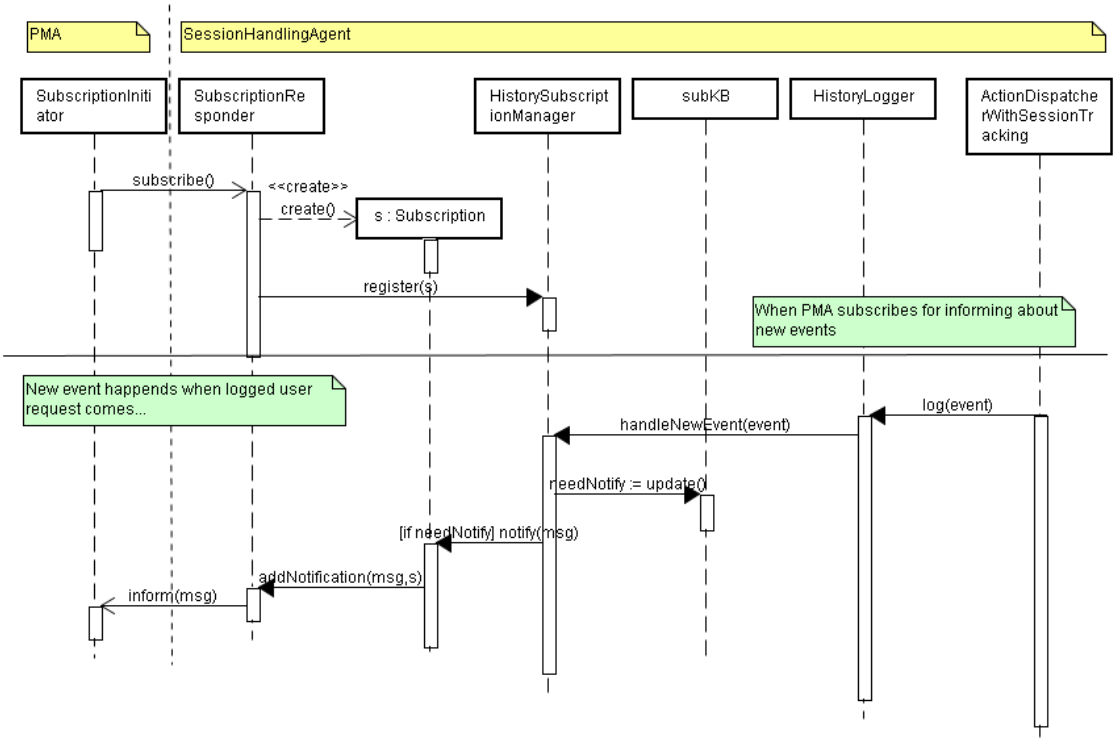


Issues

Session Tracking



Subscribing-for and informing-about new events



Exception handling and logging

We have variety of entities interested in results of requested operations:
- a user is willing to know whether system realized her request or not
- agent A need to know whether requested agent B realized the reqest or failed
- agent and its behaviours should now about errors thrown by used components
- a developer is willing to know why and when and in what context the exception was thrown

Therefore we use the following strategy at handling with exceptions and logging them

1. Exceptions are catched at the components level and, usually, wrapped into exception of higher abstraction.
2. No logging is made inside of components, with an exception of static classes inilized staticly.
3. Exceptions thrown by components has names as specific as possibile, describing error in their well-known functionality or error referenced to input data. They should not describe details of internal implementation.
4. Exceptions are finally catched at the level of behaviours and agents main code and they are not further re-thrown. Behaviours and agents have to handle with them in one or more of the following way:
   (a) log this exception (due to a developer)
   (b) if this exception occured during realization of other agent request, respond with FAILURE ACL message, containing as a Result InternalSystemError with the reason message.
   (c) If above handling fails, log this handling failure
   (d) if it is ProxyAgent, which met the exception or received FAILURE message, then inform the user about the problem.

Dependencies

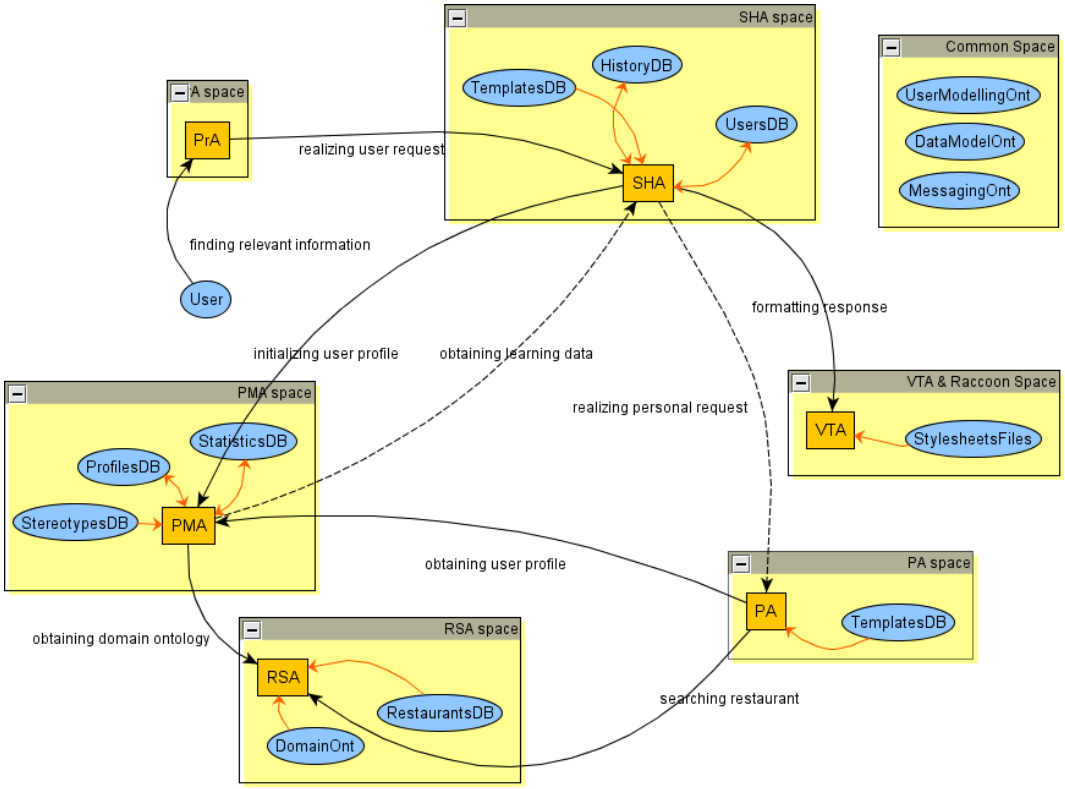
Data-roles assignment

Functional and datastore dependencies

Black lines tells the Agent A at beginning of arrows requires functionality of the

Agent B standaing at the end. When the line is continuoes, existing of Agent B registered in DF is obligatory for registration and start of work of Agent A. Dotted lines stands for optional existance, which allows Agent A to enhance it functionality. This allowed to dismiss problem for circular depenedency between agents.

Orange lines tells about read/write access of Agent from/to data models.



Semantic processing

Scheme ontologies (T-Box)

These ontologies are stored in memory-base models.

Domain ontology
file:ontology/restaurant.owl
file:ontology/location.owl
file:ontology/money.owl

Based on *Chefmoz dinning guide* [http://chefmoz.com] ontology models domain of a restaurant and referenced concepts (location and money)

User modelling ontology
file:ontology/user-modelling.owl

Define classes for profiles, stereotypes, events and user behaviours.

Data model ontology
file:ontologies/data-model.owl

Define structures (see *LayoutStructure* class) and links (see *Link* class) between them for models constructured by *SessionHandlingAgent* and *PersonalAgent*. *S*tructures is used later as source for transformation by the *ViewTransformingAgent*a and links are used to controll data flow of user scenario.

Messaging ontology
file:ontology/messageging.owl

FIPA SL0 compatible ontology used by system agents to communicate. It is used as a source for JADE ontology classes, generated with use of OntologyBeanGenerator.

Database ontologies (A-Box)

These ontologies introduce individuals and, due to their vast capacity, most of are stored in database. For the first run of the system they are loaded from initial files into database.

Polish Restaurants database
file:db/restaurants.owl

Describes 8700 Polish restaurants, translated into OWL from RDF file provided by *Chefmoz dinning guide* [http://chefmoz.com].

Stereotypes
file:db/stereotypes.owl

List of predefined stereotypes used for initialization of user profile.

Statistics
file:db/statistics.owl

Empty file for statistics

Users
file:db/users.owl

Empty file for accounts of registered users.

Profiles
file:db/profiles.owl

Empty file for user profiles.

History
file:db/history.owl

Empty file for logged events (actions performed by the logged user).

Templates
file:db/templates.owl

Templates describing layout of displayed screens.

Converting data between different ontological representation

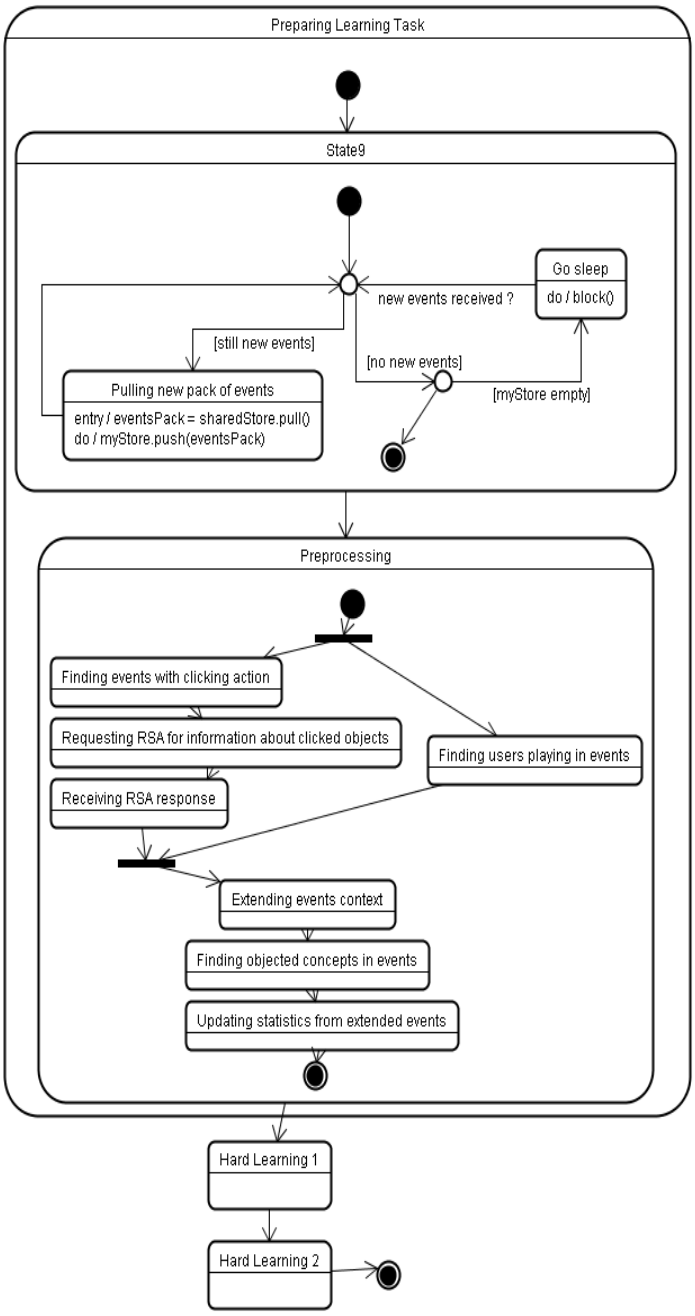
Jastor -> Thing -> Resource -> OntModel -> Jena

OWLData (messaging ontology) -> hasDataModel -> OntModel RDF/XM serialization

ibspan.tss.core.semantic.Memory

```
public void add(OntModel m);
public void add(Thing t);
public void add(Resource r);
public void add(OWLData o);
public OntModel getModel();
public OWLData buildOWLData();
```

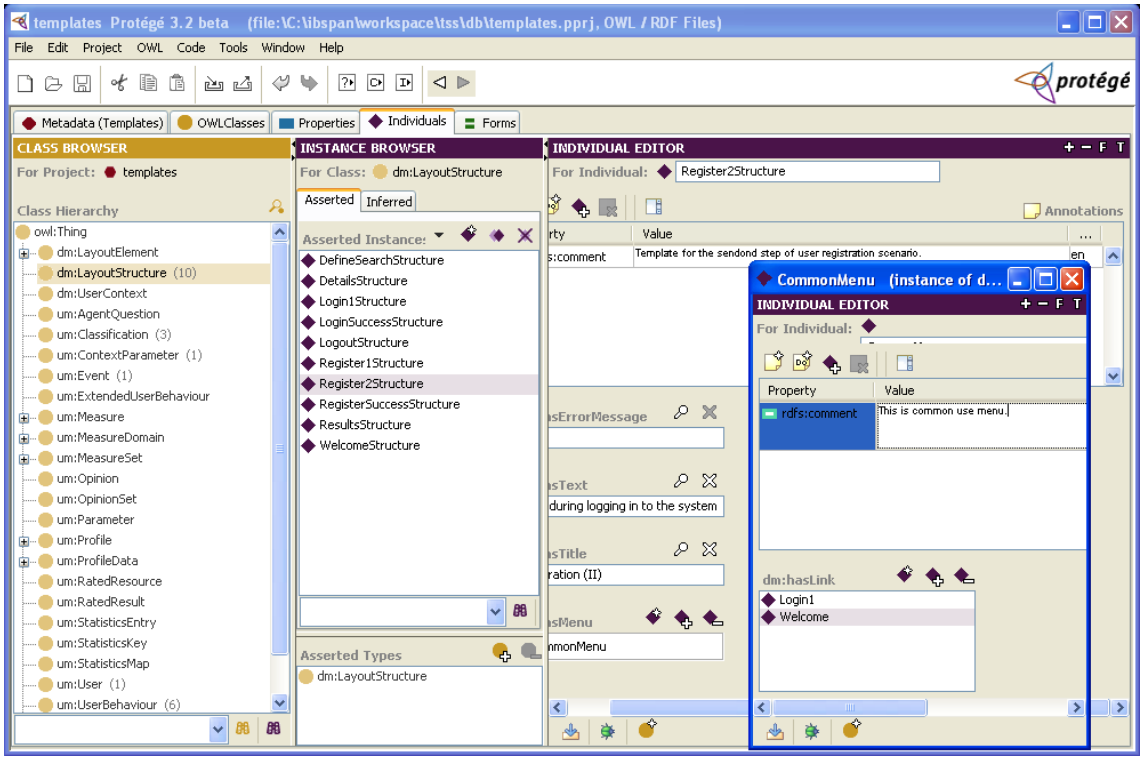
Learning profiles



Implementing single step for user scenario

Preparing view and structure

1. **Write template of a web page** by defining instance of *LayoutStructure* class in *db/templatates.owl* file. The example below shows screen of constructing *Register1Structure* in Protege ontology editor.



2. **Refresh JavaBeans for new template**, by calling standard "*onto*" ant target from command line:

```
ant onto
```

3. **Register template** inside of constructor for `SHAResponder` class.

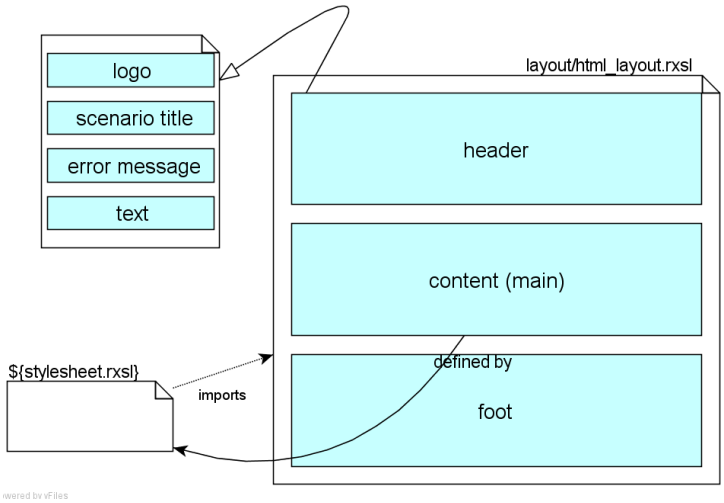
```
r = LayoutStructure.Register1Structure;
templates.register( ViewParams.GET_REGISTER_1_SCREEN_ACTION,
  DataModelFactory.getLayoutStructure(r, mTemplatesDB) );
```

`ViewParams` is interface for storing common used constant names for fields and variables used in user GUI.

4. **Write RxSLT stylesheet**, which will be used by Raccoon server to transform prepared template into media-specific document, e.g. HTML. Each stylesheet must:

   - import *layout/*_layout.rxsl* stylesheet, where * stands for media type, e.g. HTML. This stylesheet is responsible for providing standard processing of fields described in the template, you created. It also provides a few useful functions and variables.
   - Implement „main" xsl:template, to define additional content as e.g. forms.

This is described on the following figure.



Below we present except from *templates/templates/html_register_1.rxsl* file.

```
<xsl:include href="../layout/html_layout.rxsl"/>

<xsl:template name="main">

<form action="{$interface-host}:{$interface-port}"
  method="get">

<xsl:call-template name="form">
   <xsl:with-param name="action-name"
     select="'check-register-1-screen'"/>
</xsl:call-template>

<table align="center">
<tr>
  <td align="right">your name: </td>
  <td><input type="text" name="user-name" /></td>
</tr>
...
```

Function *form* is used to paste session-specific data into document. These data must be preserved during interaction between user and the system.

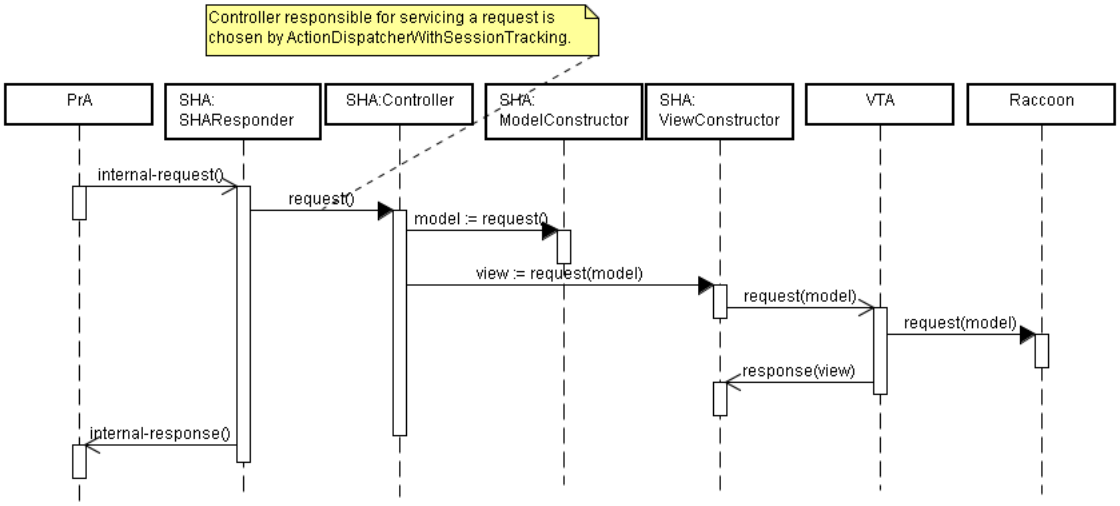
5. **Add the stylesheet entry** to Raccoon configuration file: *etc/raccoon-config.py*.

```
STYLESHEET = {( 'html-media', 'get-welcome-screen') :
                 'templates/html_standard_screen.rxsl',
               ( 'html-media', 'get-register-1-screen') :
                 'templates/html_register_1_screen.rxsl'
               }
```

Preparing control over data flow

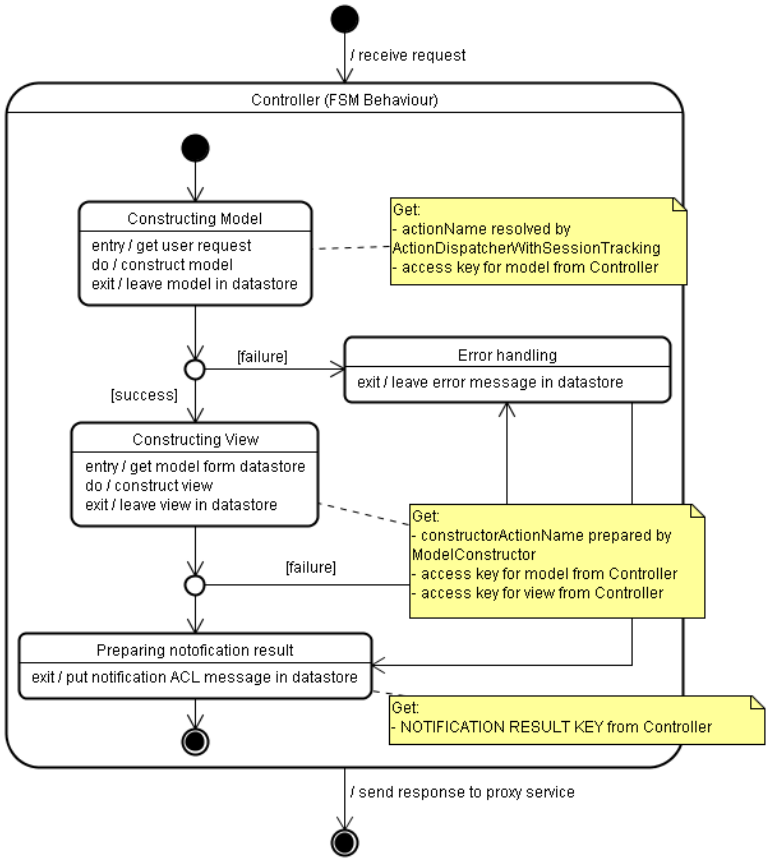
Data flow was described in much detail in referenced documentation. Here we describe things that were mainly changed, with giving the focus on Model-View-Controller (MVC) architecture. The sequence diagram below shows typical realization of user request in the system, when

ProxyAgent (PrA) receives the request.

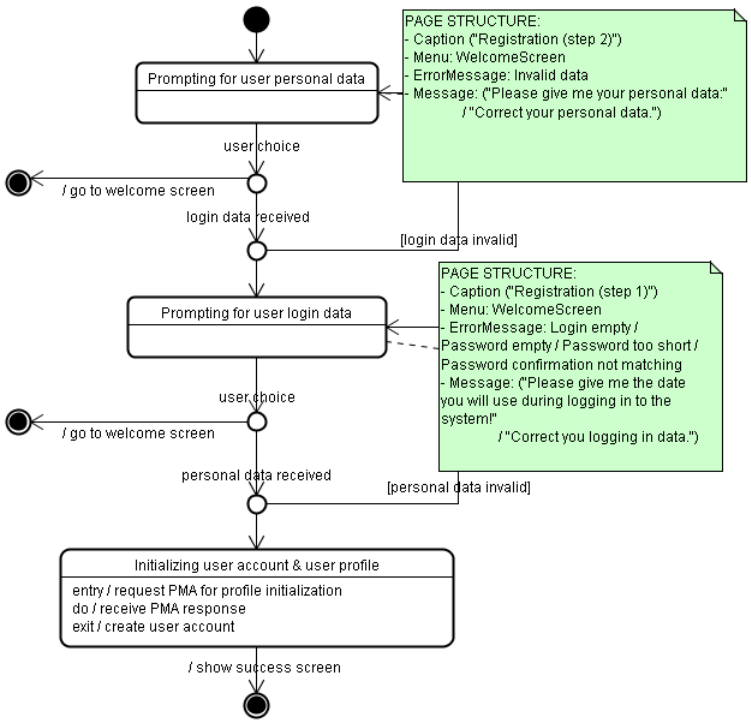


Our main interest will be realization of MVC inside of *SessionHandlingAgent* (SHA), because the rest of process is usually the same for all case and do not need intervention of a developer.

MVC architecture is realized by three classes (or classes extending them): `Controller`, `ModelConstructor` and `ViewConstructor`, which will be described below in a form of an example. The statechart diagram below presents how theses classes behaves during realization of a request.



We will remind how whole registration scenario looks, mainly from the user perspective:



We have prepared template for describing registration window, now we would like to prepare implementation for checking registration data usered entered and choosing appriopriate response for her (choices made after *„Prompting for user login data"* state). These is how we proceed:

1. **Create instance of `Controller`** inside of constructor `SHAResponder` class.

```
Controller c = new Controller( myAgent, getDataStore(),
  ACTION_KEY, ACTION_NAME_KEY, REQUEST_KEY,
  RESULT_NOTIFICATION_KEY );
```

2. **Write class extending `ModelConstructor`.** This class is responsible for preparing instance of `DataModel` class, containing copy of registrated template and, optionally, some additional ontological data, e.g. results of restaurants' search. `DataModel` should also contain `constructionActionName` set up, which is used as suggestion for `ViewConstructor` (and further for *ViewTransformingAgent*), for choosing appriopriate transformation. A developer is provided with set a useful functions to access user request data, update state of the session or access its data.

```
protected SHA_process_user_request getAction()
protected UserRequest getUserRequest()
protected String getActionName()
protected Session getSession(String sessionID)
protected LayoutStructure getTemplateCopy(String actionName)
```

Of course whole request does not have to be completely realized by SHA and can be forwarded to other agent, e.g. to *PersonalAgent* (PA).

The result of operation should be returned with one of the provided functions:

```
protected void putModel(DataModel model)
protected void putError(InternalError error)
```

Full implementation of our example can be found in *CheckRegistration1ModelConstructor.java* source file.

3. **Re-use extension of `ViewConstructor`.** Usually `SimpleViewConstructor` (utilizing VTA and Raccoon service) will be enough for your purposes. However you may write your own extension, but please rember, that model constructed by `ModelConstructor` can be access by the following method

```
protected DataModel getModel()
```

and the result of view constructing process should be put in local *DataStore* with one of functions:

```
protected void putView(DataView view)
protected void putError(InternalError error)
```

4. **Register ModelConstructor and ViewConstructor in Controller**:

```
c.registerModelConstructor(mc);
c.registerViewConstructor(vc);
```

Please remember, that `ModelConstructor` and `ViewConstructor` use its own space in *DataStore* to exchange operation results, and therefore they cannot be shared among different instances of `Controller` class.

5. **Bind `Controller` with the specific action request** inside of constructor of `SHAResponder` class.

```
selector.registerHandler( false,
  ViewParams.CHECK_REGISTER_1_ACTION, c );
```

The following screen shows the result of providing invalid data by the user.

Acknowledgements

The authors would like to thank you the following person for their contribution Wiktor Moderau (for dig-art), Mateusz Dominiak (for support for individuals in Jastor), Adam Souzis and Michał Olczak (for support in configuration of Raccoon in Python), Mateusz Kruszyk (for stereotypes design), Michał Szymczak (for design of restaurant ontology and referenced domains), Minor Gordon (for vital design ideas), dr Maria Ganzha for mathematical considerations and diagrams validation and many thans to Marcin Paprzycki for his enforcing power to do it. Also many thanks for people from very active jena-dev mailgroup and also for authors of Raccoon, JADE and Jastor for their support.