

# Mục lục

Server .....	3
Servlet .....	3
JSP .....	3
EL .....	3
Tag .....	3
Custom tag .....	4
Tomcat .....	4
Persistence .....	4
Hibernate .....	4
Các khái niệm cơ bản thường gặp .....	5
SessionFactory .....	5
Session .....	5
Transaction .....	5
HibernateUtil .....	5
Transient object / Persistence object .....	5
Entity / Component .....	5
Lazy loading .....	5
Nguyên tắc: KHÔNG DÙNG HIBERNATE .....	6
Tham khảo .....	6
MySQL .....	6
phpMyAdmin .....	6
Select / Update / Delete... .....	6
Stored procedure .....	6
Trigger .....	7
Web service .....	7
Web service .....	7
REST .....	7
Tài nguyên .....	7
Hành động .....	8
Kết quả của hành động .....	8
Định dạng .....	8
Tham khảo .....	8
Jersey .....	8
Ví dụ minh hoạ .....	8
Tham khảo .....	11
Client .....	11
HTML .....	11
JavaScript .....	11
Prototype .....	11
AJAX .....	11

AJAX với Prototype.....	11
Tham khảo.....	11
Data formats .....	12
JSON .....	12
XML .....	12
POJO .....	12
Kết hợp các công nghệ.....	12
Module đăng nhập bằng AJAX.....	12
Bước 0: Chuẩn bị.....	12
Bước 1: Cấu hình module (action).....	14
Bước 2: Viết code cho module (action).....	15
Bước 3: Viết trang JSP: Phần hiển thị ban đầu.....	15
Bước 4: Viết trang JSP: Các thao tác thực hiện bằng JavaScript.....	15
Bước 5: Viết Web service.....	16
Bước 6: Kiểm thử Web service.....	17
Bước 7: Viết trang JSP: Giao tiếp với Web Service.....	19
Bước 7: Viết trang JSP: Xử lý lỗi AJAX.....	21
Bước 8: Kiểm thử toàn bộ chức năng.....	21
Kiểm tra cấu trúc trang: FireBug.....	21
Kiểm tra giao tiếp với WebService: HttpFox.....	21

# Server

## ***Servlet***

## ***JSP***

Trong tất cả các tệp JSP xử lý hành động (actions/\*) cần chứa hai dòng đầu tiên như sau:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<% include file="../../includes/common.jsp" %>
```

Trong đó tệp common.jsp định nghĩa tất cả các thư viện cần thiết cho công việc của một action. Nếu thiếu thư viện nào cần thông báo cho cả nhóm để bổ sung.

## ***EL***

*Expression Language* là ngôn ngữ riêng trong trang JSP, dùng để truy cập giá trị của Java bean cũng như gọi các hàm có sẵn hay tự tạo. VD:

```
${question.answers[i].content} - trả về nội dung của câu trả lời thứ i trong câu hỏi
${ocw:apiUrl('topic.search')} - trả về URL đến API có tên topic.search
```

Mỗi biến trong EL thuộc về một *scope* nhất định, trong đó hay dùng là *sessionScope* (truy cập vào session, vd: `sessionScope.user`, `sessionScope.editToken`) và *param/paramValues* (truy cập nội dung form được submit).

Các hàm EL thường dùng là `fn:escapeXml` và `ocw:apiUrl`, `ocw:restUrl`,...

Trong mọi trường hợp cần tránh sử dụng *scriptlet* (`<%= %>`), thay vào đó hãy dùng EL để tăng tính nhất quán và dễ đọc cũng như dễ thay đổi.

Tham khảo [JavaServer Page 3rd Edition](#) mục 6.3, 7.4.2, Appendix C.

## ***Tag***

JavaServer Pages tag là phần tử có dạng tương tự như mã HTML thực hiện một số hành động nhất định trên server. VD:

```
<ol>
<c:forEach items="${question.answers}" var="answer">
    <li>${answer.content}</li>
</c:forEach>
</ol>
```

Ở đây "c" là prefix gắn với tag library, được định nghĩa bằng directive `<%@ taglib ... %>` ở đầu tệp JSP (hoặc nằm trong tệp `common.jsp` được include) còn "forEach" là tên của tag.

Có 3 dạng tag:

- Chuẩn (standard): các tag được định nghĩa trong đặc tả của JSP, bắt đầu bằng tiền tố `jsp`, vd: `jsp:include`
- JSTL: các tag trong thư viện chuẩn (JSP Standard Tag Library), chủ yếu sử dụng là `c:forEach`, `c:if`, `c:choose` (cùng với `c:when`, `c:otherwise`).
- Tự tạo (custom): các tag được định nghĩa trong dự án, vd: `actionLink`, `articleLink`,...

**Luôn luôn sử dụng tag thay cho scriptlet (`<% %>` hoặc `<%= %>`)** vì tag dễ đọc hơn và để hạn chế code những công việc thuộc về controller ở trong view.

Tham khảo [JavaServer Page 3rd Edition](#) mục 5.6, 7.4.2.

## **Custom tag**

Custom tag là rất quan trọng để tăng tính nhất quán, tiết kiệm công sức và cho phép sự thay đổi được thực hiện dễ dàng hơn trong tương lai. Cần thống nhất một nguyên tắc là **luôn luôn sử dụng custom tag trong những công việc chung cho toàn bộ dự án**. Những ví dụ cho custom tag là: `actionLink`, `actionButton`, `articleLink`, `userLink`, `topics`...

Custom tag được định nghĩa bằng các tag file đặt trong thư mục `/WEB-INF/tags`, sử dụng custom tag bằng tiền tố `tags`: theo sau là tên của tag file.

Tham khảo [JavaServer Page 3rd Edition](#) mục 11.1.

## **Tomcat**

# **Persistence**

## **Hibernate**

Trong khi Java làm việc trên mô hình hướng đối tượng với các quan hệ kế thừa, kết tập, hợp thành... thì các CSDL MySQL chỉ có các bảng liên kết với nhau bằng khoá ngoài. Để tạo ra chiếc cầu nối giữa hai thế giới cần có một Object-relational mapping (ORM) framework.

Hibernate là một ORM framework mạnh một số chức năng của nó có thể liệt kê như:

- Ánh xạ (mapping) mô hình hướng đối tượng và mô hình quan hệ thông qua tệp XML hoặc annotation kể cả các cấu trúc kế thừa, kết tập...

- Lưu trữ (persistence) tự động các đối tượng Java dạng [POJO](#) vào CSDL quan hệ.
- Hiệu suất cao (performance) tận dụng các tính năng lazy-loading, batch updating, cache 2 tầng,...
- Ngôn ngữ truy vấn hướng đối tượng: Hibernate Query Language (HQL) cho phép truy vấn dữ liệu theo phong cách hướng đối tượng và trả về đối tượng hoặc danh sách các đối tượng thay vì result set.

## Các khái niệm cơ bản thường gặp

### ***SessionFactory***

Lớp chứa tất cả thông tin cấu hình cần có để chạy một ứng dụng Hibernate, chịu trách nhiệm tạo ra các Session.

### ***Session***

Lớp biểu diễn một phiên làm việc với Hibernate, mọi thao tác truy vấn, thêm/sửa/xoá đều phải thực hiện thông qua đối tượng lớp này.

### ***Transaction***

Lớp biểu diễn giao dịch (gói các hành động cùng hoàn thành hoặc huỷ bỏ), các thao tác thêm/sửa/xoá cần được thực hiện trong một giao dịch để đảm bảo CSDL chứa dữ liệu cập nhật và hợp lệ.

### ***HibernateUtil***

Lớp công cụ của riêng dự án trong đó quản lý SessionFactory và Session.

- getSessionFactory(): trả về đối tượng SessionFactory.
- getSession(): trả về đối tượng Session của luồng hiện tại hoặc tạo ra Session mới.
- closeSession(): đóng Session của luồng hiện tại nếu có.
- count(hql): đếm kết quả của câu truy vấn hql (Hibernate Query Language).

### ***Transient object / Persistence object***

Transient object là đối tượng Java được tạo mới bên ngoài Hibernate, chưa được lưu trong CSDL và chưa được quản lý bởi Hibernate. Persistence object là đối tượng được quản lý bởi Hibernate.

### ***Entity / Component***

Entity (thực thể) là một đối tượng tồn tại độc lập. Component là một thành phần của đối tượng khác và chỉ có thể được truy cập thông qua một entity.

### ***Lazy loading***

Hibernate có thể trì hoãn việc truy xuất CSDL bằng cách tạo ra các *proxy* thế chỗ cho đối tượng Java

thật. Khi người sử dụng truy cập vào các thuộc tính của proxy Hibernate mới tạo ra lời gọi đến CSDL để lấy về thông tin cần thiết do đó tiết kiệm tối đa chi phí.

Lazy loading chỉ hoạt động nếu Session chứa đối tượng vẫn còn mở. Đó là lí do trong các lớp DAO không có câu lệnh `session.close()`, công việc này được controller (action, api, rest) thực hiện sau khi mọi thao tác với dữ liệu hoàn tất.

## **Nguyên tắc: KHÔNG DÙNG HIBERNATE**

Mọi truy vấn đến CSDL cần được thực hiện qua các lớp Data Access Object, code của các action, api... không nên gọi trực tiếp chức năng của Hibernate để tránh phụ thuộc không cần thiết.

## **Tham khảo**

[Java persistence with Hibernate](#) chap 1,2.

## **MySQL**

MySQL là hệ quản trị CSDL miễn phí được dùng phổ biến hiện nay. MySQL ngày càng được hoàn thiện, ngoài các chức năng cơ bản giờ nó còn hỗ trợ stored procedure/function và trigger.

Việc truy cập đến MySQL được Hibernate thực hiện tự động, code của dự án không nên gọi trực tiếp đến các dịch vụ MySQL (kể cả trong DAO), điều này giúp mã nguồn có thể tương thích cả với các hệ quản trị CSDL khác nữa.

## **phpMyAdmin**

Là phần mềm quản trị CSDL MySQL viết trên PHP được đóng gói kèm với gói phần mềm WAMP/LAMP (Windows/Linux - Apache - MySQL - PHP). phpMyAdmin không hỗ trợ lời gọi stored procedure.

## **Select / Update / Delete...**

Các thao tác cơ bản tuân theo chuẩn ngôn ngữ SQL và được thực hiện tự động hoàn toàn bởi Hibernate.

## **Stored procedure**

Là các đoạn mã được lưu trữ bên trong CSDL, đóng gói các thao tác xử lý phức tạp không thể thực hiện bằng một câu lệnh SQL đơn giản. Lời gọi stored procedure có dạng `call sp_name(param1, param2, ...)`. Để gọi thử stored procedure trong quá trình phát triển dự án cần sử dụng giao diện dòng lệnh như sau:

```
> mysql -uroot -proot (thay root bằng username và password thật)
mysql> call sp_name(param1, param2, ...);
```

## Trigger

Là các đoạn mã được gọi khi một sự kiện nhất định xảy ra đối với CSDL: trước/sau khi thêm/cập nhật/xoá dòng trong bảng. Trigger có thể được sử dụng để quản lý phiên bản dữ liệu bằng cách sao chép dữ liệu ra một bảng khác trước khi thay đổi.

Tham khảo

- [Stored procedure](#)
- [Trigger](#)

## Web service

### ***Web service***

Web service (Web API) là một hệ thống phần mềm cung cấp tương tác máy-máy qua mạng, nó có thể được sử dụng bởi một thành phần khác của hệ thống (vd JavaScript thông qua AJAX) hoặc bởi một hệ thống khác bên ngoài.

Lợi ích chính của việc áp dụng một kiến trúc web service thống nhất là:

- Có thể phát triển web service độc lập với phía client.
- Code dễ viết và dễ đọc hơn
- Dễ dàng công bố API và cho phép các đối tác sử dụng dữ liệu của trang web sau này.

### ***REST***

REST (REpresentational State Transfer) là một phong cách thiết kế web service đơn giản, dễ hiểu, dễ sử dụng, trong đó:

- Đối tượng được biểu diễn bởi URL
- Hành động được biểu diễn bởi HTTP method
- Kết quả của hành động được biểu diễn bởi mã HTTP

(đây chỉ là một cách nhìn, không phải là định nghĩa chính thức/chính xác)

### **Tài nguyên**

REST tập trung vào tài nguyên và các trạng thái của nó chứ không phải hành động vì thế mọi giao dịch với REST trước hết bắt đầu bằng việc xác định tài nguyên. Trong REST mỗi tài nguyên có một URL xác định, vd:

- `/rest/questions`: danh sách tất cả các câu hỏi
- `/rest/questions/1`: câu hỏi có id là 1

- `/rest/questions/topic/1`: danh sách các câu hỏi thuộc chủ đề có id là 1
- `/rest/questions/author/1`: danh sách các câu hỏi do người dùng có id là 1 tạo ra

## Hành động

Sau khi xác định được tài nguyên ta có thể thực hiện những hành động làm thay đổi trạng thái của tài nguyên đó. REST sử dụng 4 hành động có sẵn của giao thức HTTP:

- GET: lấy về tài nguyên
- POST: tạo tài nguyên mới
- PUT: cập nhật trạng thái của tài nguyên
- DELETE: xóa tài nguyên

## Kết quả của hành động

Là mã HTTP và thông tin kèm theo (trạng thái của tài nguyên hoặc thông tin về lỗi), các mã thường gặp:

- HTTP 200 (OK): thành công
- HTTP 204 (No content): thành công nhưng không có dữ liệu trả về
- HTTP 400 (Bad request): tham số đầu vào không hợp lệ
- HTTP 404 (Not found): không tìm thấy tài nguyên được yêu cầu

## Định dạng

REST không yêu cầu một định dạng dữ liệu bắt buộc, để thuận tiện trong dự án thống nhất dùng:

- [JSON](#) khi trả về dữ liệu / lỗi
- Form khi gửi lên thông tin

## Tham khảo

[Restful Java with Jax-RS](#) chap 1.

## Jersey

JAX-RS là một framework hỗ trợ tạo REST Web Service bằng cách sử dụng [annotation](#). JAX-RS chỉ cung cấp các đặc tả và API còn Jersey cài đặt và mở rộng JAX-RS để web service có thể chạy thực sự.

## Ví dụ minh họa

Để minh họa cách hoạt động của Jersey hãy xem xét ví dụ sau trong lớp `oop.controller.rest.AnswerResource`:

```
@Path("/answers") (1)
```



```

public class AnswerResource extends AbstractResource {
    @GET (2)
    @Path("/{id: \\d+}") (3a)
    public ObjectResult<Answer> get(
        @PathParam("id") long id) { (3b)
        Answer answer = AnswerDAO.fetch(id);
        ResourceUtil.assertFound(answer);
        return new ObjectResult<Answer>(answer);
    }

    @PUT (2)
    @Path("/{id: \\d+}") (3a)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED) (4)
    public ObjectResult<Answer> update(
        @PathParam("id") long id, (3b)
        @FormParam("content") String contentStr, (5)
        @FormParam("correct") boolean correct) { (5)
        Answer answer = AnswerDAO.fetch(id);
        ResourceUtil.assertFound(answer);
        ResourceUtil.assertParamNotEmpty("content", contentStr);
        answer.setContent(new Text(contentStr));
        answer.setCorrect(correct);
        return new ObjectResult<Answer>(answer);
    }

    @POST (2)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED) (4)
    public ObjectResult<Answer> create( (6)
        @FormParam("question") long questionId,
        @FormParam("content") String contentStr,
        @DefaultValue("false") @FormParam("correct") boolean correct) {
        ResourceUtil.assertParamNotEmpty("content", contentStr);
        Answer answer = AnswerDAO.create(questionId, contentStr, correct);
        return new ObjectResult<Answer>(answer);
    }

    @DELETE (2)
    @Path("/{id: \\d+}") (3a)
    public void delete(
        @PathParam("id") long id) { (3b)
        AnswerDAO.delete(id);
    }
}

```

- (1) **@Path** cho biết đường dẫn để truy cập đến tài nguyên, mọi lớp tài nguyên đều phải có annotation này ở đầu.
- (2) **@GET**, **@POST**, **@PUT**, **@DELETE** đánh dấu HTTP method mà hàm hỗ trợ, mỗi khi tài nguyên được gọi với 1 trong 4 HTTP method này thì hàm tương ứng sẽ được gọi.
- (3a) **@Path** khi đi với hàm thì nội dung của nó được ghép với **@Path** của lớp thành `"/answers/..."`, "id" là tên đại diện cho phần đường dẫn từ sau `/answers/`, định dạng của nó được mô tả bằng biểu thức chính quy `"\\d+"` (nhiều hơn hay 1 chữ số). Nếu URL phù hợp với định dạng này (chẳng hạn `/answers/1`) thì hàm sẽ được gọi còn nếu không (chẳng hạn `/answers/abc`)

thì hàm không được gọi.

- (3b) **@PathParam** cho biết tham số tiếp theo lấy nội dung từ phần đường dẫn có tên là "id" (được annotate ở phía trên), Jersey sẽ tự động chuyển kiểu dữ liệu từ String thành long.
- (4) **@Consumes(MediaType.APPLICATION\_FORM\_URLENCODED)** chỉ ra rằng hàm này nhận đầu vào là form.
- (5) **@FormParam** ánh xạ giá trị của trường trong form vào tham số của hàm.
- (6) **ObjectResult<T>** đóng gói kết quả đơn lẻ giúp thư viện có thể *tự động* chuyển kết quả thành JSON, khi cần có thể extends lớp này để cung cấp các thông tin bổ sung. **Luôn sử dụng ObjectResult<T>** thay vì trả về đối tượng trực tiếp có thể gây ra lỗi do tương tác phức tạp giữa Jersey và Hibernate.

Để trả về danh sách nhiều đối tượng tham khảo hàm sau trong lớp

`oop.controller.rest.QuestionResource:`

`@GET`

```
public ListResult<BaseQuestion> list( (7)
    @DefaultValue("0") @QueryParam("start") int start, (8)
    @DefaultValue("10") @QueryParam("size") int size) {
    ResourceUtil.assertParamValid(size <= MAX_PAGE_SIZE, "size",
size, "too large");
    List<BaseQuestion> list = BaseQuestionDAO.fetch(start, size);
    String nextUrl = null;
    if (list.size() >= size) {
        nextUrl = "/questions?start=" + (start + size) +
            "&size=" + size; (9)
    }
    return new ListResult<BaseQuestion>(list, nextUrl);
}
```

- (7) **ListResult<T>** đóng gói kết quả kiểu danh sách để có thể *tự động* chuyển thành JSON. **Luôn sử dụng ListResult<T>** thay vì trả về danh sách trực tiếp sẽ gây lỗi vì thư viện không thể chuyển đổi giao diện List thành JSON.
- (8) **@QueryParam** ánh xạ tham số trong query string (vd ?start=0&size=12) thành tham số của hàm. Với các danh sách lớn cần phải phân trang để tránh truyền quá nhiều dữ liệu qua đường truyền.
- (9) **nextUrl**: URL đến trang tiếp theo của kết quả, nếu đã hết kết quả thì nextUrl rỗng. thuộc tính này giúp việc sử dụng web service thuận tiện hơn.

## Ghi chú

- **AbstractResource** là lớp cha của tất cả các lớp cung cấp service. Tất cả các lớp được đặt trong gói `oop.controller.rest`, Jersey được cấu hình để phát hiện tất cả các resource trong gói này.
- **@Produces(MediaType.APPLICATION\_JSON)** (nằm trong class `AbstractResource`) chỉ ra rằng đầu ra của mọi hàm trong các lớp con đều là JSON, các lớp extends từ `AbstractResource` không cần phải lặp lại annotation này.
- Lớp **ResourceUtil** cung cấp các hàm tiện ích để kiểm tra dữ liệu.
- Cần phải đóng gói dữ liệu trả về vào **ObjectResult** và **ListResult** (xem `QuestionResource`) nếu

không thư viện sẽ không chuyển đổi được Java object thành JSON, ngoài ra có thể extends các lớp này để cung cấp thông tin thêm.

## **Tham khảo**

[Restful Java with Jax-RS](#) chap 3,4,5.

# **Client**

## **HTML**

## **JavaScript**

## **Prototype**

[Prototype](#) là một thư viện JavaScript đơn giản nhằm tạo sự thuận tiện cho lập trình viên với những tác vụ như định nghĩa lớp, truy cập và sửa đổi các thành phần HTML, lập trình AJAX...

Các hàm thường dùng khi lập trình với Prototype là:

- `$('#an_id')`: truy cập phần tử HTML có trường id là "an\_id"
- `elem.show()` / `elem.hide()`: hiện/ẩn phần tử elem bằng cách thay đổi thuộc tính CSS "display" thành "auto"/"none".
- `elem.addClass()` / `elem.removeClass()`: thêm/xoá lớp CSS của phần tử

## **AJAX**

AJAX là sự kết hợp của JavaScript và lời gọi không đồng bộ, viết tắt của Asynchronous JavaScript and XML (tuy nhiên định dạng dùng để trao đổi dữ liệu không nhất thiết phải là XML, trong dự án chúng ta dùng [JSON](#)). Lời gọi không đồng bộ cho phép lấy dữ liệu từ server về client mà không cần phải reload trang, khi lời gọi hoàn tất một hàm JavaScript sẽ được gọi và thực hiện những hành động cần thiết.

AJAX tăng sự tương tác của người dùng với ứng dụng, giảm thời gian load trang, giảm tải cho server và đem đến những ứng dụng không thể có với công nghệ trước đó.

## **AJAX với Prototype**

Prototype hỗ trợ việc gọi AJAX theo cách đơn giản, dễ hiểu và tương thích với hầu hết các trình duyệt. Chỉ cần khởi tạo đối tượng `Ajax.Request` và cung cấp cho nó những thông tin cần thiết: địa chỉ, phương thức HTTP, tham số, callback.

## **Tham khảo**

[Introduction to Ajax](#)

# Data formats

## ***JSON***

Đọc [JSON 3-minute lesson](#).

Ví dụ trong dự án:

```
{
  "question": {
    "id": 1,
    "content": {
      "text": "1+1=?"
    },
    "answers": [
      { "id": 1, "content": { "text": "3" }, "correct": false },
      { "id": 2, "content": { "text": "2" }, "correct": true },
      { "id": 3, "content": { "text": "1" }, "correct": false }
    ],
    "level": 5;
  }
}
```

## ***XML***

## ***POJO***

POJO = Plain Old Java Object, là các đối tượng không extends hay implements từ lớp hay giao diện riêng của một framework nào như Hibernate, Servlet... Các ví dụ của POJO trong dự án là các lớp trong gói model, các lớp action.

Lợi ích chủ yếu của POJO là:

- Tách rời (decoupling) các đối tượng thuộc không gian ứng dụng với các thành phần của framework nhờ đó có thể cài đặt business logic trước khi áp dụng framework hoặc thay thế framework sau này.
- Dễ kiểm thử: có thể tách rời business logic ra khỏi các framework để kiểm thử dễ dàng.

# Kết hợp các công nghệ

## ***Module đăng nhập bằng AJAX***

Dưới đây mình trình bày lại các bước đã thực hiện để cài đặt một module bằng AJAX hoàn chỉnh, mọi người có thể tham khảo để làm các action/module khác tương tự.

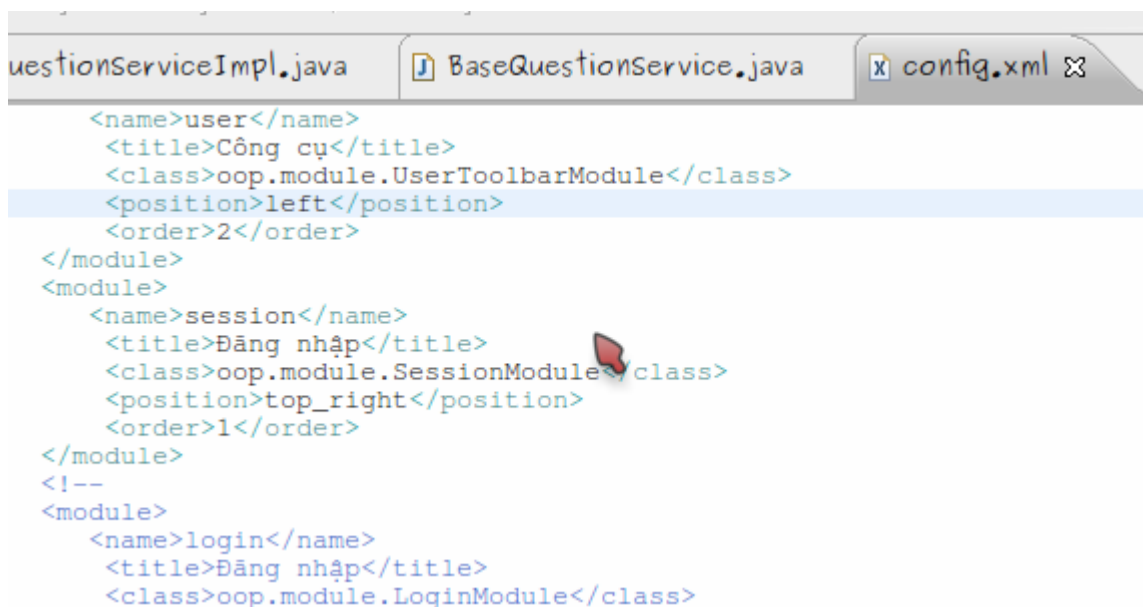
### **Bước 0: Chuẩn bị**

Cài đặt FireBug, JSONView và HttpFox cho FireFox



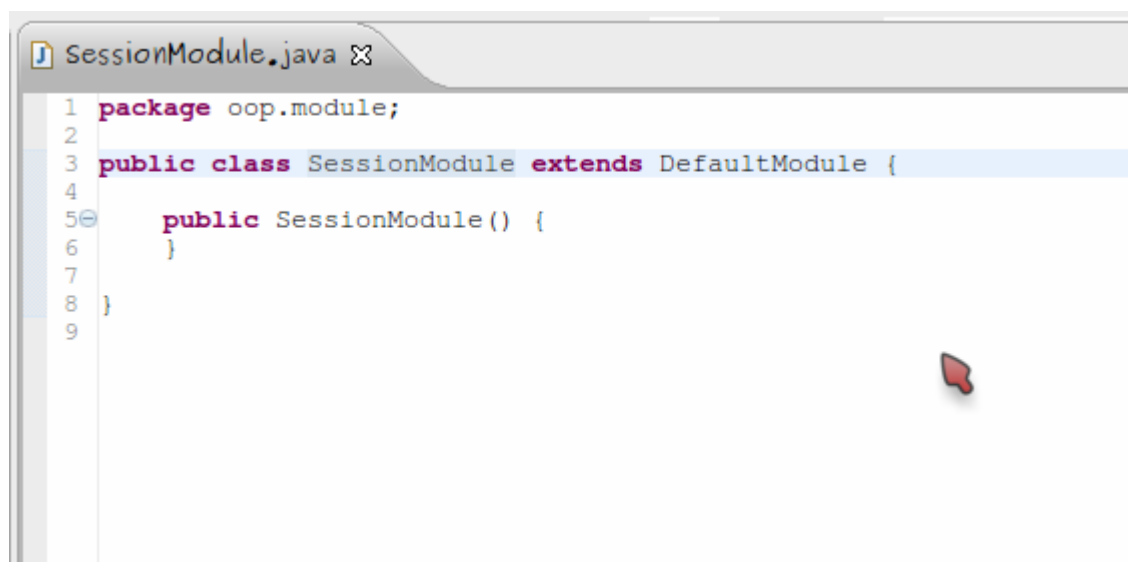
## Bước 1: Cấu hình module (action)

Mở tệp config.xml (nhấn Ctrl-Shift-R và gõ config) thêm cấu hình cho module (action):



```
<!-- user -->
<module>
  <name>user</name>
  <title>Công cụ</title>
  <class>oop.module.UserToolbarModule</class>
  <position>left</position>
  <order>2</order>
</module>
<module>
  <name>session</name>
  <title>Đăng nhập</title>
  <class>oop.module.SessionModule</class>
  <position>top_right</position>
  <order>1</order>
</module>
<!-- login -->
<module>
  <name>login</name>
  <title>Đăng nhập</title>
  <class>oop.module.LoginModule</class>
</module>
```

Tạo lớp Java thực hiện hành động: oop/data/SessionModule.java (tương ứng với thẻ class trong cấu hình).



```
1 package oop.module;
2
3 public class SessionModule extends DefaultModule {
4
5     public SessionModule() {
6     }
7
8 }
9
```

Tạo tệp JSP để trình bày: WebContent/templates/default/modules/session.jsp (tương ứng với tên của module trong cấu hình).

```
*session.jsp x
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <%@ include file="/includes/common.jsp" %>
4
5
6
7
8
9
10
11
12
13
14
15
```

## Bước 2: Viết code cho module (action)

Với action code Java sẽ thực hiện hành động cần làm trên dữ liệu.

Với module code Java sẽ lấy dữ liệu để JSP trình bày (JSP không làm được việc này), module này chưa cần thông tin gì nên để trống.

## Bước 3: Viết trang JSP: Phần hiển thị ban đầu

```
session.jsp x
3 <%@ include file="/includes/common.jsp"%>
4
5 <div class="session">
6 <div class="jsmenu" id="user-toolbar"
7   style="display: ${sessionScope.login ? 'block' : 'none'};"
8 <ul class="level1 horizontal" id="user-toolbar-root">
9   <li class="level1">${sessionScope.user.fullname}
10 <ul class="level2 dropdown">
11   <li><ocw:actionLink name="user.edituser"
12     title="Sửa thông tin người dùng">Sửa thông tin</ocw:actionLink></li>
13   <li><ocw:actionLink name="user.preference">Tuỳ chọn</ocw:actionLink>
14   </li>
15   <li><ocw:actionLink name="user.logout"
16     title="Đăng xuất khỏi hệ thống">Đăng xuất</ocw:actionLink></li>
17 </ul>
18 </li>
19 </ul>
20 </div>
21 <div style="display: ${sessionScope.login ? 'none' : 'block'};"
22   id="guest-toolbar">
23 <ul>
24   <li><ocw:actionLink name="user.signup">Đăng kí</ocw:actionLink></li>
25   <li><ocw:actionLink name="user.login" title="Đăng nhập">
26     Đăng nhập</ocw:actionLink></li>
27 </ul>
28 </div>
29 </div>
30
```

## Bước 4: Viết trang JSP: Các thao tác thực hiện bằng JavaScript

Các thao tác này có thể là ẩn/hiện các trường, hiện hộp hội thoại hoặc textarea...

Thêm code xử lý sự kiện:

```
22     id="guest-toolbar">
23 <ul>
24     <li><ocw:actionLink name="user.signup">Đăng kí</ocw:actionLink></li>
25     <li><ocw:actionLink name="user.login" title="Đăng nhập"
26         onclick="openLoginDialog(); return false;">Đăng nhập</ocw:actionLink></li>
27 </ul>
28 </div>
29 />
```

Viết hàm xử lý sự kiện:

```
42
43 <script type="text/javascript">
44
45     var sessionLoginDialogContent = $('session-loginDialog-content').innerHTML;
46     $('session-loginDialog-content').remove();
47
48     function openLoginDialog() {
49         Dialog.confirm(sessionLoginDialogContent, {
50             width : 300,
51             okLabel : "Đăng nhập",
52             cancelLabel : "Thôi",
53             buttonClass : "session-button",
54             className : "alphacube",
55             id : "session-loginDialog",
56             cancel : function(win) { return true; },
57             ok : session_login
58         });
59         $('session_name').focus();
60     }
61
62     //-->
63 </script>
```

Hàm này sử dụng thư viện [Prototype window](#) để hiện hộp hội thoại. Nội dung của hộp hội thoại lấy từ phần tử «session-loginDialog-content», sau khi lấy được nội dung thì phần tử này được xoá đi để tránh trùng ID.

- cancel: là hàm xử lý sự kiện khi người dùng nhấn nút «Thôi»
- ok: hàm xử lý sự kiện khi người dùng nhấn nút «Đăng nhập»

Eclipse xử lý JavaScript không được tốt lắm nên thường báo lỗi sai, chúng ta không cần quá bận tâm đến những chỗ bị đánh dấu đỏ/vàng, chỉ cần chạy thử được trên trình duyệt là được.

## Bước 5: Viết Web service

Web Service cần được đặt trong package oop.controller.rest, tham khảo thêm phần Web service.

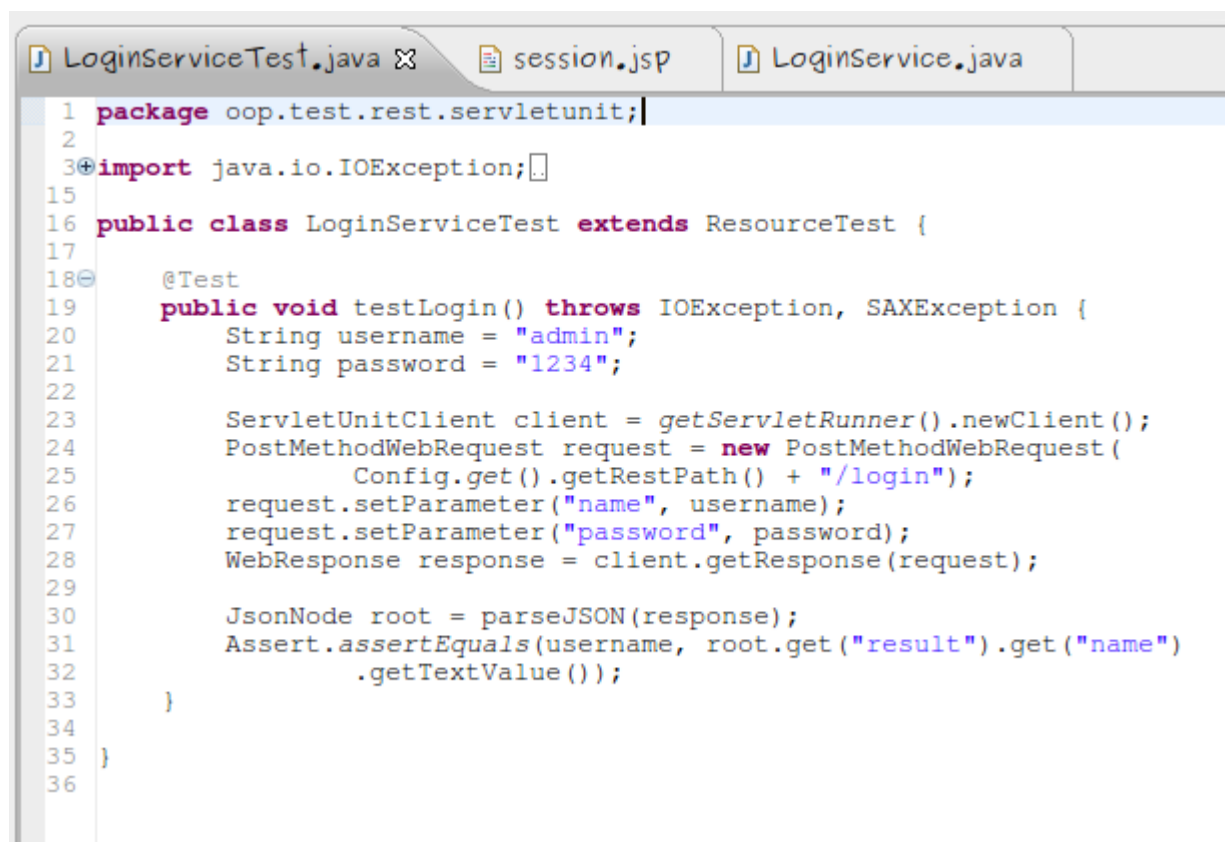


```
session.jsp  LoginService.java ✕
1 package oop.controller.rest;
2
3 import javax.ws.rs.Consumes;
17
18 @Path("/login")
19 public class LoginService extends AbstractResource {
20
21     @POST
22     @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
23     public ObjectResult<User> login(
24         @FormParam("name") String name,
25         @FormParam("password") String password) {
26         User user = UserDao.fetchByUsername(name);
27         if (user == null) {
28             throw new WebApplicationException(Response.status(Status.BAD_REQUEST)
29                 .entity(new ErrorResult("invalid name")).build());
30         }
31         if (user.matchPassword(password)) {
32             SessionUtils.setUser(getSession(), user);
33             return new ObjectResult<User>(user);
34         } else {
35             throw new WebApplicationException(Response.status(Status.BAD_REQUEST)
36                 .entity(new ErrorResult("invalid password")).build());
37         }
38     }
39 }
40 }
41 }
```

## Bước 6: Kiểm thử Web service

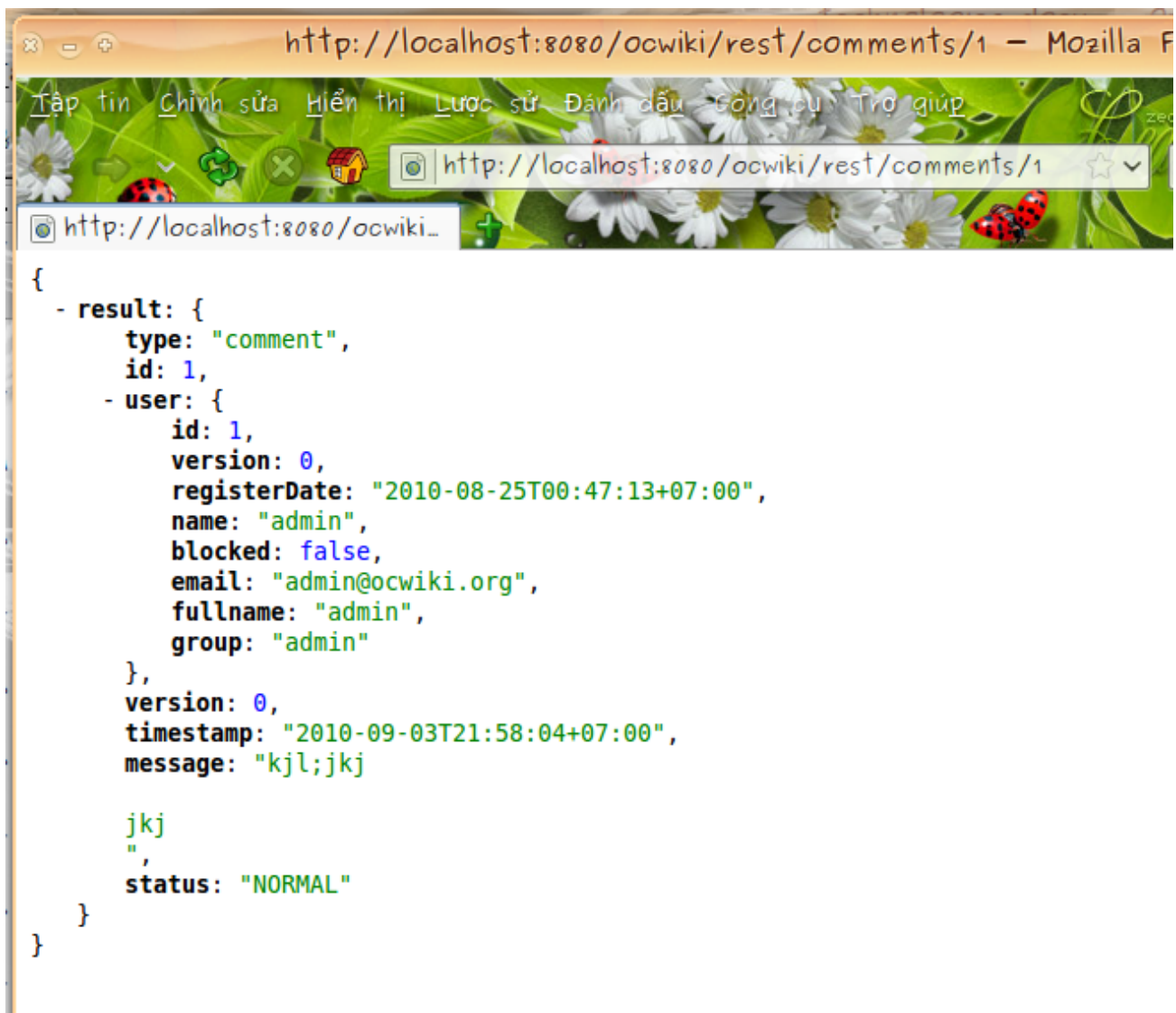
Viết unit test (nếu có thể) hoặc chạy thử trên trình duyệt.

Unit test nên được viết trong thư mục /test/src. Để viết unit test cho Web service hãy đặt lớp trong package oop.test.rest.servletunit và extends từ lớp ResourceTest (sử dụng thư viện [Servlet Unit](#)).



```
1 package oop.test.rest.servletunit;|
2
3+import java.io.IOException;|
15
16 public class LoginServiceTest extends ResourceTest {
17
18     @Test
19     public void testLogin() throws IOException, SAXException {
20         String username = "admin";
21         String password = "1234";
22
23         ServletUnitClient client = getServletRunner().newClient();
24         PostMethodWebRequest request = new PostMethodWebRequest(
25             Config.get().getRestPath() + "/login");
26         request.setParameter("name", username);
27         request.setParameter("password", password);
28         WebResponse response = client.getResponse(request);
29
30         JsonNode root = parseJSON(response);
31         Assert.assertEquals(username, root.get("result").get("name")
32             .getTextValue());
33     }
34 }
35
36
```

Nếu không thể viết unit test, hãy kiểm thử web service của mình trên trình duyệt bằng cách truy cập địa chỉ tương ứng (chỉ thực hiện được với loại hành động GET). Ví dụ chạy thử `oop.controller.rest.CommentResource` trên FF (đã cài JsonView):



## Bước 7: Viết trang JSP: Giao tiếp với Web Service

```

62     function session_login() {
63         if ($F('session_name') == '') {
64             $('#session_nameError').innerHTML = 'Bạn cần nhập tên người dùng';
65             return false;
66         }
67         if ($F('session_password') == '') {
68             $('#session_passwordError').innerHTML = 'Bạn cần nhập mật khẩu';
69             return false;
70         }
71         new Ajax.Request(
72             restPath + '/login',
73             {
74                 method : 'post',
75                 parameters : {
76                     name : $F('session_name'),
77                     password : $F('session_password')
78                 },
79                 requestHeaders : {
80                     Accept : 'application/json'
81                 },
82                 evalJSON : true,
83                 onSuccess : function(transport) {
84                     location.reload(true);
85                 },
86                 onFailure : function(transport) {
87                 }
88             });
89         return false;
90     }
91

```

Tạo lời gọi AJAX đến Web Service bằng đối tượng [Ajax.Request](#) của thư viện Prototype. Trong đó:

- restPath: đường cơ sở của Web Service, được khởi tạo trong index.jsp (ví dụ: <http://localhost:8080/ocwiki/rest>)
- method: phương thức HTTP
- parameters: các tham số (query string với phương thức GET và trường name=value với phương thức POST)
- requestHeaders: thêm «Accept: 'application/json'» để Web Service trả về kết quả dạng JSON
- evalJSON: đặt bằng true để Prototype tự động chuyển kết quả dạng chuỗi JSON thành đối tượng JavaScript
- onSuccess: hàm xử lý sự kiện lời gọi AJAX thành công
- onFailure: hàm xử lý sự kiện lời gọi AJAX thất bại

#### Cập nhật 05/09: Tải lên dữ liệu dạng JSON

- Dùng method: 'post'
- Thêm thuộc tính contentType: 'application/json'
- Thay parameters bằng postBody: Object.toJSON(data) với «data» là đối tượng JS chứa dữ liệu cần gửi

## Bước 7: Viết trang JSP: Xử lí lỗi AJAX

Khi gặp lỗi, web service sẽ gửi về thông báo lỗi thay vì dữ liệu trả về. Có thể tìm ra các thông báo này bằng cách đọc web service hoặc tự tạo ra lỗi và dùng HttpFox để xem thông báo (xem hướng dẫn sử dụng HttpFox ở bước cuối).

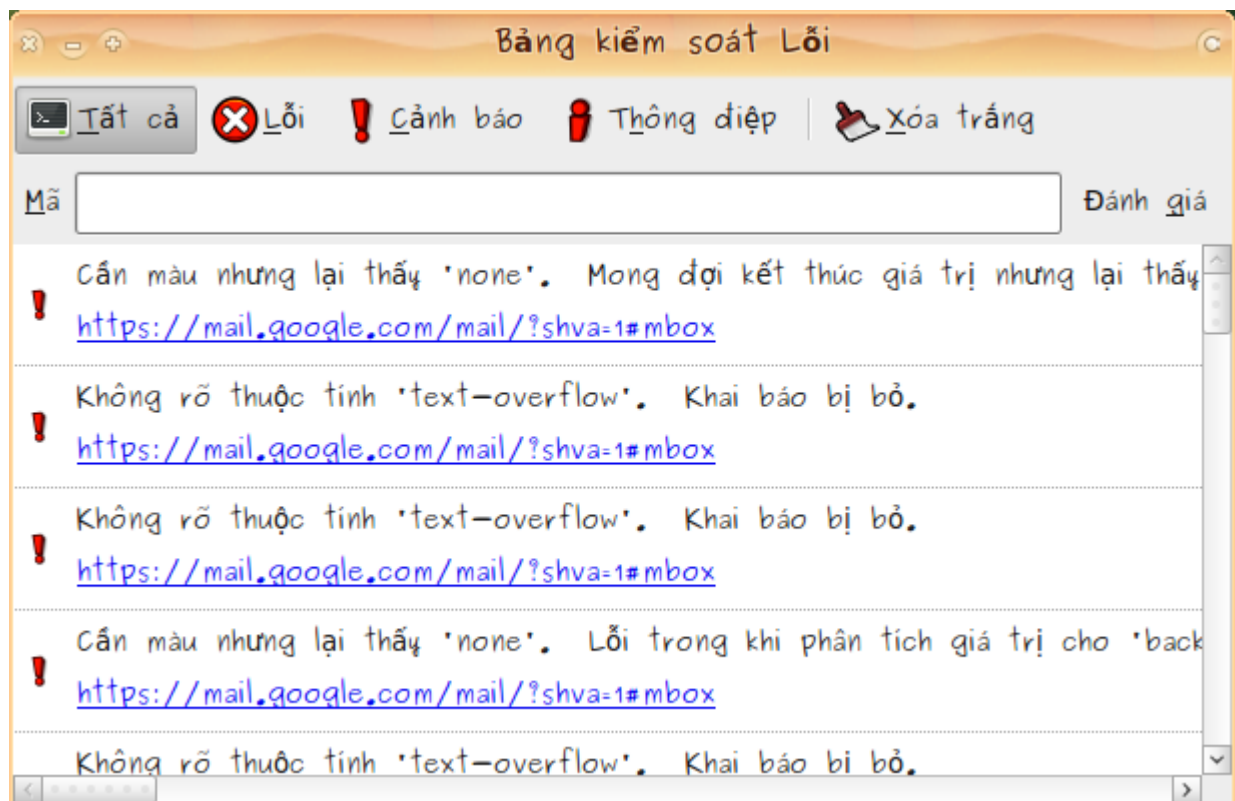
```
onFailure : function(transport) {  
    var code = transport.responseJSON.code;  
    if (code == 'invalid password') {  
        $('#session_passwordError').innerHTML = 'Sai mật khẩu';  
    } else if (code == 'invalid name') {  
        $('#session_nameError').innerHTML = 'Người dùng không tồn tại';  
    }  
}
```

## Bước 8: Kiểm thử toàn bộ chức năng

Chạy thử từ đầu đến cuối, tất cả các trường hợp có thể xảy ra xem chức năng đã hoạt động như mong muốn chưa.

### Kiểm soát lỗi JavaScript: Error console

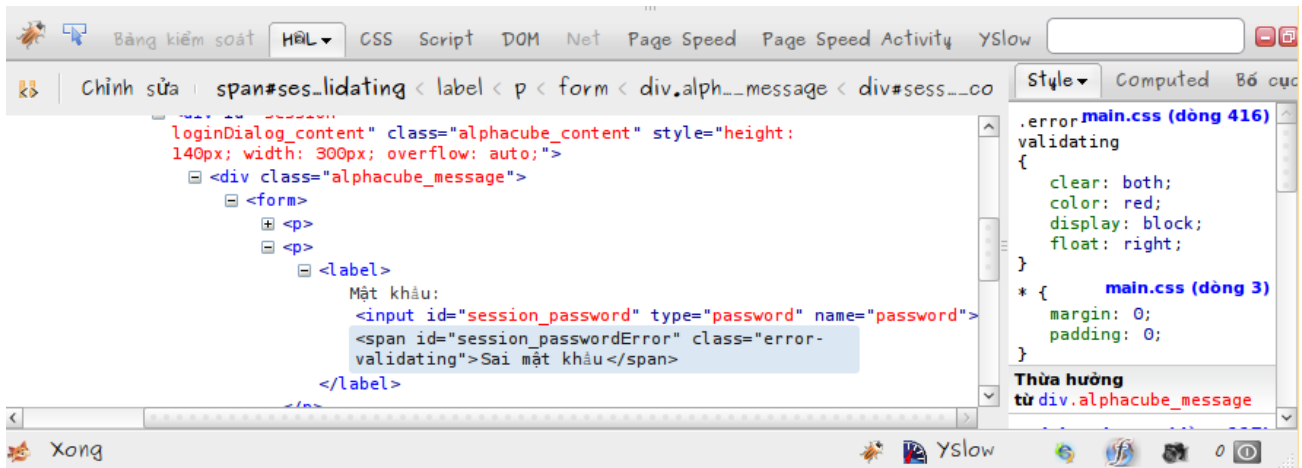
Error console là tính năng có sẵn của FF, có thể mở bằng menu: Tools > Error console hoặc nhấn Ctrl-Shift-J



### Kiểm tra cấu trúc trang: FireBug


FireBug có rất nhiều tính năng hữu ích như:

- Xem, chỉnh sửa mã HTML của phần tử trên trang web
- Xem, chỉnh sửa thuộc tính CSS của phần tử
- Đặt break point, chạy từng dòng code JavaScript
- Xem các thành phần trang đã được nạp
- Và nhiều tính năng khác...



### **Kiểm tra giao tiếp với WebService: HttpFox**

Để chắc chắn về định dạng trả về của dữ liệu/thông báo lỗi có thể kiểm tra bằng HttpFox, cách làm như sau:

- Bật HttpFox bằng cách nhấn biểu tượng  ở góc dưới-phải cửa sổ FF
- Nhấn nút «Start» để bắt đầu theo dõi các thao tác HTTP
- Chạy thử form, nhập giá trị nào đó và gửi
- Nhấn nút «Stop» để dừng theo dõi.
- Xem URL, các tham số, kiểu dữ liệu gửi đi, kiểu dữ liệu mong muốn nhận về (có phải Accept: application/json không?) đã gửi
- Xem response code, dữ liệu trả về

start stop Clear  Autoscroll

start_	Ti_	Sent	Recei_	Method	Result	Type	URL
00:14:28...	0.175	670	182	POST	400	application/json	http://_login

Headers Cookies Query String POST Data Content

Request _	Value	Response_	Value
(Request-Li...	POST /ocwiki/rest/login HTTP/1.1	(status-Lin...	HTTP/1.1 400 Bad Request
Host	localhost:8080	Server	Apache-Coyote/1.1
User-Agent	Mozilla/5.0 (X11; U; Linux i686; vi; rv:1.9...	Content-Ty...	application/json
Accept	application/json	Transfer-E...	chunked
Accept-Lan...	vi-vn,vi;q=0.8,en-us;q=0.5,en;q=0.3	Date	Sat, 04 Sep 2010 05:59:25 GMT
Accept-Enc...	gzip,deflate	Connection	close

Xong Yslow 0

▶ start

⊗ stop

✖ Clear

🔍

☑ Autoscroll

start\_

Ti\_

sent

Recei\_

Method

Result

Type

URL

🔗

00:14:28...

0.175

670

182

POST

400

application/json

http://\_login

<

>

Headers

Cookies

Query string

POST Data

Content

Type:application/x-www-form-urlencoded; charset=UTF-8

Parameter	Value
name	admin
password	lkjsfks

<

>

☒ Pretty

☐ Raw

🦊 Xong

🐛

📊 Yslow

🌐

🔗

📷

0

🔊



▶ start ⊗ stop ✖ Clear  ☑ Autoscroll

start_	Ti_	sent	Recei_	Method	Result	Type	URL
00:14:28...	0.175	670	182	POST	400	application/json	http://_login

HeadersCookiesQuery stringPOST DataContent

Type: application/json  

```
{*code*:invalid password*}
```

☐ Pretty ☒ Raw

XongYslow0