**Hướng dẫn tổ chức javascript theo module dựa trên Lazy Load.**

1. **Mục đích**

* Bài viết hướng dẫn cách tổ chức javascript trên client áp dụng mô hình chia để trị như các framework hiện đại đang phổ biến hiện nay (Angular, ReactJs, VueJs, …)
* Ngoài ra, bài viết còn cung cấp các kỹ thuật cần thiết khác để tạo ra một cấu trúc hoàn chỉnh (Lazy load, marshalling, unmarshalling). Từ đó có thể dễ dàng bảo trì, mở rộng ứng dụng web.

1. **Yêu cầu kiến thức cơ bản**

* Nắm vững khái niệm về ngôn ngữ lập trình Java, lập trình **hướng đối tượng** và các thành phần liên quan
* Nắm vững khái niệm về **JAXB**, **marshalling**, **unmarshalling** (tham khảo tại địa chỉ <http://www.kieutrongkhanh.net/2016/10/jaxb-chuyen-oi-xml-schema-hay-dtd-tro.html>).
* Biết cách xây dựng một ứng dụng **JavaEE**, kết nối với hệ quản trị cơ sở dữ liệu sử dụng JDBC.
* Có kiến thức nền tảng về **html**, **JavaScript**, **XML** (tham khảo tại <https://www.w3schools.com> )
* Nắm vững và đã thực hiện toàn bộ nội dung của bài viết liên quan đến XSL (tham khảo lại bài **Style Sheets – Giới thiệu cách sử dụng và cách viết style sheet áp dụng cho XML – Extensible Style Language – XSL (Phần 1)** <http://www.kieutrongkhanh.net/2016/08/style-sheets-gioi-thieu-cach-su-dung-va.html> )
* Nắm được cách áp dụng mô hình MVC bên phía client (tham khảo lại bài **Áp dụng mô hình MVC bên phía client – Nhận file pdf bằng blob** <http://www.kieutrongkhanh.net/2018/06/topic-ap-dung-mo-hinh-mvc-ben-phia.html> )­­­

1. **Công cụ yêu cầu:**

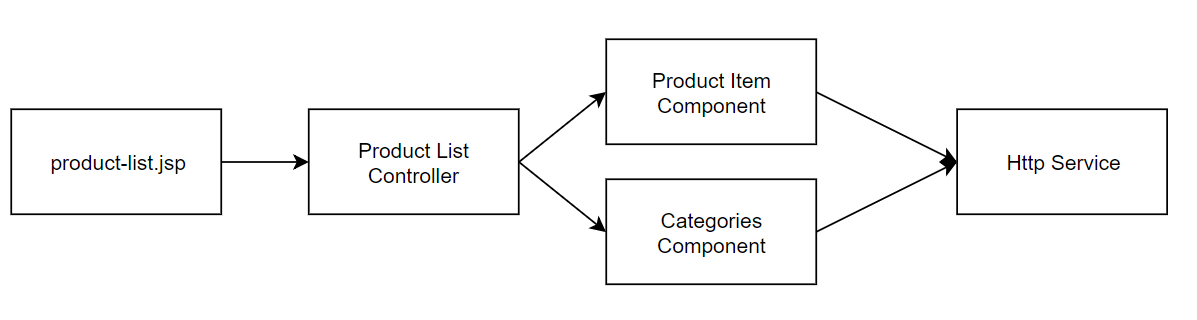
* Netbeans IDE 8.1
* Tomcat server 8.x
* SQL Server Management Studio

1. **Ý tưởng**

* Dựa trên MVC và ý tưởng chia để trị, chúng ta có thể chia nhỏ các logic của ứng dụng ra thành nhiều module độc lập với nhau. Những module đó có thể sử dụng lại ở trong ứng dụng.

1. **Architecture**

* Các thành phần chính:
  + Service:
    - Service là module chứa những function để thực hiện một vài logic nhất định.
    - Rằng buộc: Service này cũng có thể sử dụng service khác để thực hiện xử lý.
  + Component:
    - Một component sẽ có khả năng lưu trữ data, thực hiện việc render một đoạn html của component đó.
    - Rằng buộc: Component này có thể sử dụng service, component khác để thực hiện xử lý.
  + Controller:
    - Một trang JSP sẽ có một controller tương ứng để quy định cách chạy của trang JSP đó.
    - Rằng buộc: Controller này có thể sử dụng service, component để thực hiện việc xử lý. Không có module nào được sử dụng lại controller.
* Ví dụ hiển thị danh sách sản phẩm:
  + Để hiển thị một trang danh sách sản phẩm (product-list.jsp) thì chúng ta cần có một controller tương ứng (ProductListController).
  + Trang danh sách sản phẩm cần hiển thị sản phẩm và các category. Vì thế ta có 2 component tương ứng (Product Item Component, Category Component).
  + Product Item Component và Category Component đều phải lấy dữ liệu từ API rồi render ra html. Vì thế nên cả Product Item Component và Category Component đều sử dụng Http Service để gởi HTTP Request.



Hình

1. **Lazy Load**

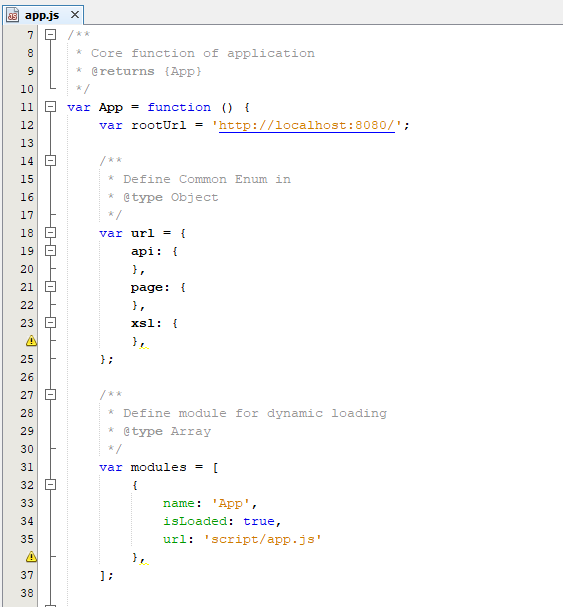
* Khi nhìn vào kiến trúc trên, chúng ta có thể thấy chỉ với một trang JSP đơn giản mà cần tới 4 file javascript. Cũng đồng nghĩa chúng ta phải viết 4 thẻ <script></script> để tải code của 4 file javascript đó. Giả sử khi áp dụng cấu trúc này vào 1 trang JSP phức tạp, chúng ta sẽ phải viết hàng chục thẻ <script></script> của tất cả component, service mà chúng ta sử dụng. Điều này dẫn đến sự hạn chế khi sử dụng chia để trị ở client.
* Vì thế, trong mô hình này chúng ta sẽ hiện thực cơ chế lazy load. Cơ chế này dựa vào khai báo phụ thuộc ở file javascript đã được tải. Từ đó, tự động tải những file javascript bị phụ thuộc về và thực thi.
* Xét ví dụ ở hình 1:
  + Product List Controller phụ thuộc vào Product Item Component và Categories Component.
  + Product Item Component phụ thuộc vào Http Service.
  + Categories Component phụ thuộc vào Http Service.
* Có thể thấy rằng nhờ việc khai báo phụ thuộc như trên thì có thể biết được 1 trang jsp cần những file nào. Vì thế trong file product-list.jsp chỉ cần viết 1 thẻ <script></script> để tải code của Product List Controller thôi.

1. **Hiện thực mô hình**

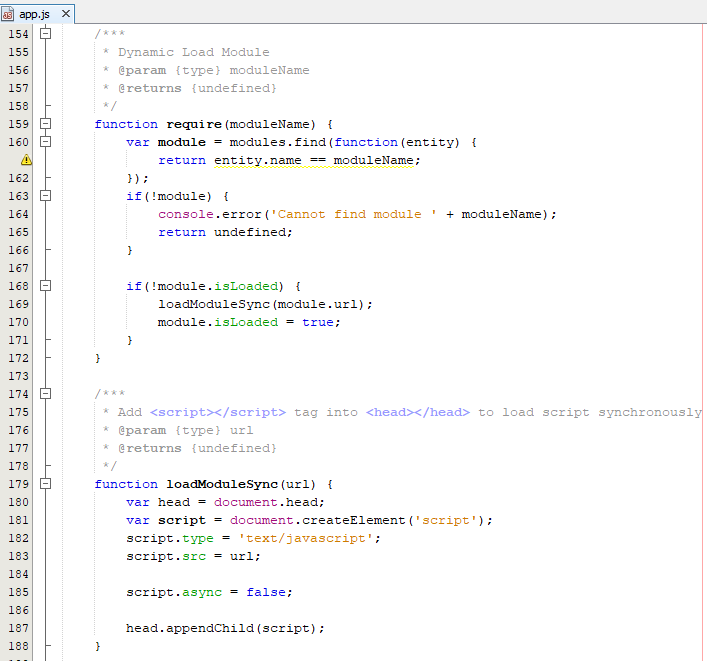
* Mỗi một module sẽ viết trên một file javascript riêng biệt.
* Tạo một object core, object này sẽ chứa khai báo của các module và đường link để tải module đó. Đồng thời object này cũng chứa tất cả function, Enum được sử dụng ở nhiều nơi, xuyên suốt ứng dụng (URL của jsp, URL của xsl, URL của API,…)

1. Hiện thực App (Module core):

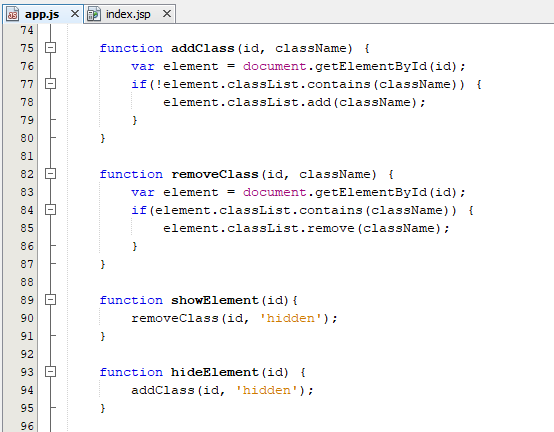
* Khai báo các URL sẽ sử dụng trong ứng dụng và module sẽ sử dụng trong ứng dụng



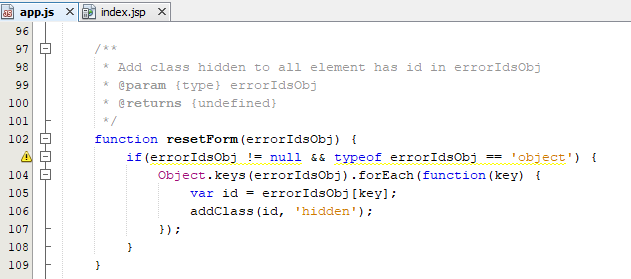
* Một module được khai báo với 3 field:
  + Tên module (name): Dùng để định danh module. Convention: giá trị của field này là tên [Constructor Function](https://www.w3schools.com/js/js_object_constructors.asp) của module đó.
  + Đường dẫn (url): Dùng để xác định đường dẫn để tải module.
  + Cờ đã được tải hay chưa (isLoaded): Dùng để đánh dấu module đã được tải về hay chưa. Ở đây module App luôn được tải về khi load trang JSP nên isLoaded = true.
* Tiếp theo, chúng ta hiện thực hàm tải module khi cần thiết:



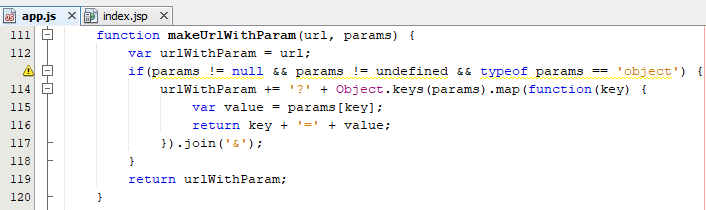
* Trước khi tải module, chúng ta phải kiểm tra module đó đã được tải chưa. Sau khi tải module xong phải đánh dấu module đó đã được tải rồi.
* Để thực hiện tải module, chúng ta gắn thêm 1 thẻ <script src=”{module.url}”></script> thì trình duyệt chạy tới sẽ tự tải module mới về.
* Tiếp theo, chúng ta thêm một số function thường được sử dụng lặp đi lặp lại trong ứng dụng:
  + Ẩn, hiện element



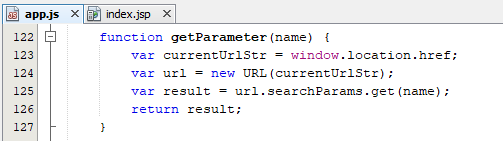
* + Reset form (ẩn đi những element hiển thị lỗi):



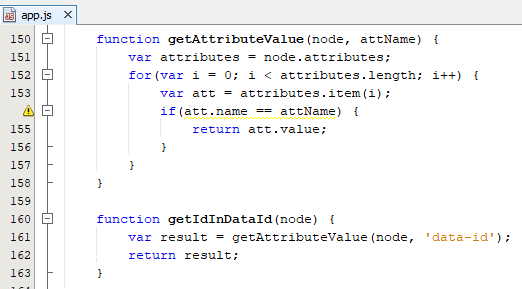
* + Tạo URL với các query parameter



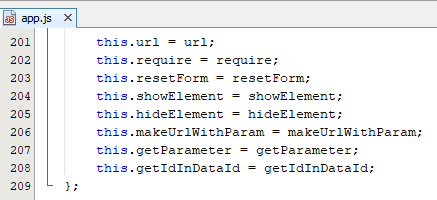
* + Lấy dữ liệu từ query parameter



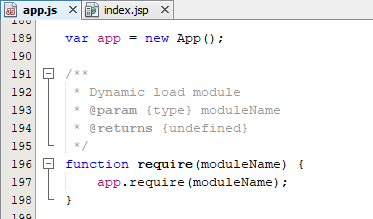
* + Lấy dữ liệu từ attribute của element. Đồng thời, chúng tôi cũng tạo một function để lấy dữ liệu của attribute “data-id”. Bởi vì trong quá trình sử dụng mô hình này, chúng tôi gọi hàm lấy dữ liệu từ attribute “data-id” rất nhiều lần:



* Sau khi hiện thực được những function cơ bản của ứng dụng, chúng ta public các function đó ra bên ngoài class:



* Trong mô hình này, App là 1 object core, cho nên chúng ta tạo một biến toàn cục để lưu object này. Trong object App, hàm require được sử dụng trong mọi module, vì thế ta tạo một hàm toàn cục require() để sử dụng tiện hơn.

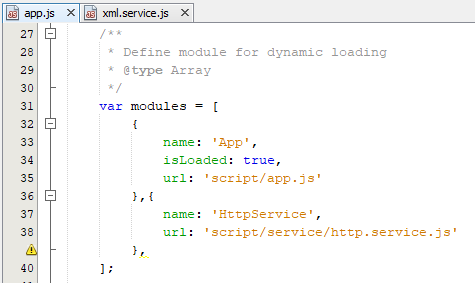


1. Hiện thực HttpService (Module core):

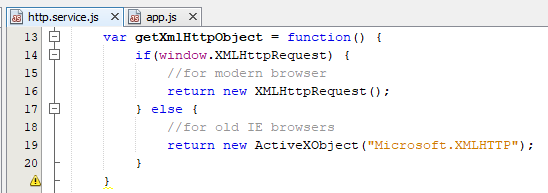
* Mục đích của mô hình này là tách server và client ra thành hai phần riêng biệt. Nên chúng ta lấy dữ liệu chỉ dựa vào API. Vì thế chúng ta cần phải hiện thực module HttpService để gởi request tới RESTful API.
* Trong module này, chúng tôi sử dụng [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) để xử lý việc bất đồng bộ. (có thể tham khảo ở <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>)
* Tạo HttpService trên 1 file mới (http.service.js).



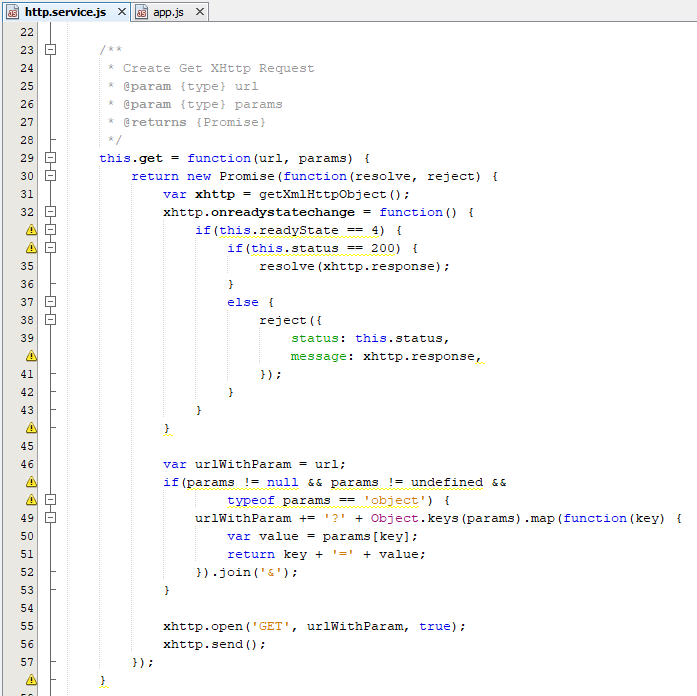
* Đăng ký module vào App core.



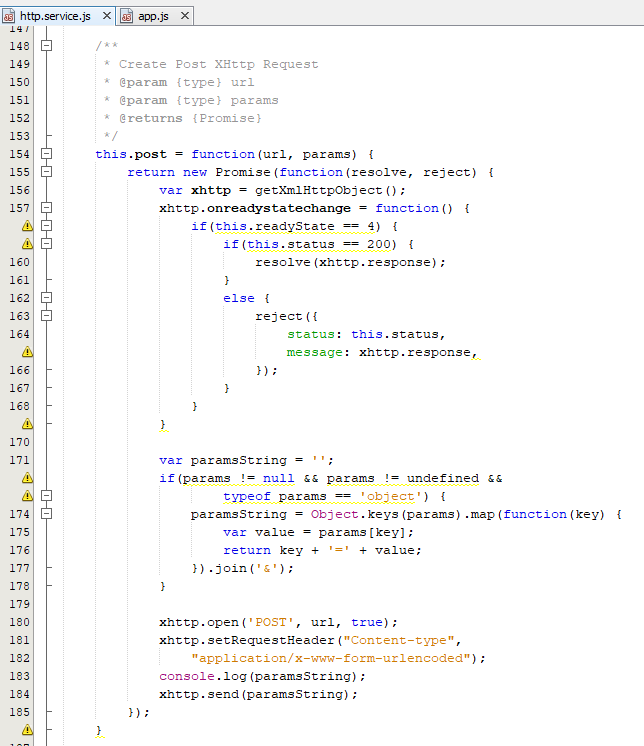
* Hiện thực hàm private khởi tạo XMLHttpObject:



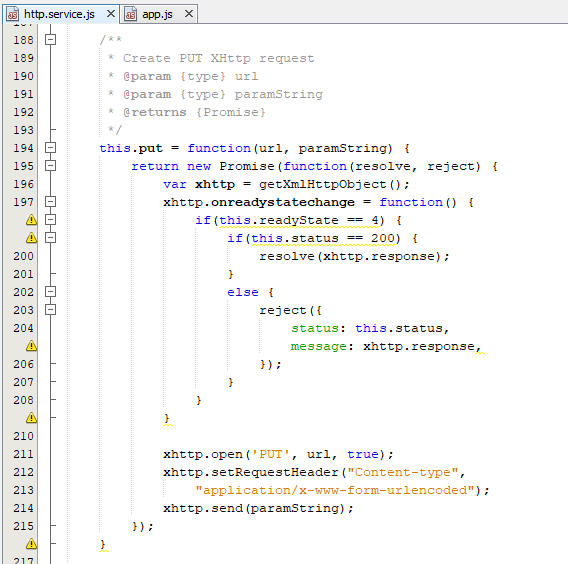
* Tạo hàm public để gởi GET request:



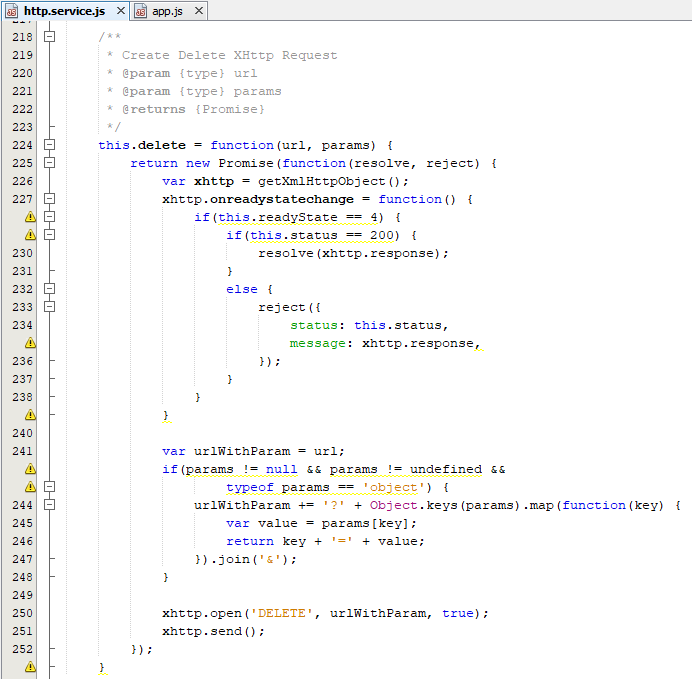
* Tạo hàm public để gởi POST request, sẽ trả về [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise):



* Tạo hàm public để gởi PUT request, sẽ trả về [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise):

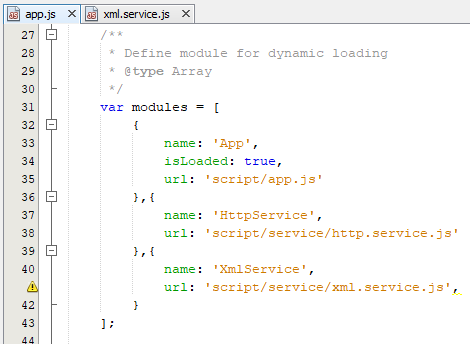


* Tạo hàm public để gởi DELETE request, sẽ trả về [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise):

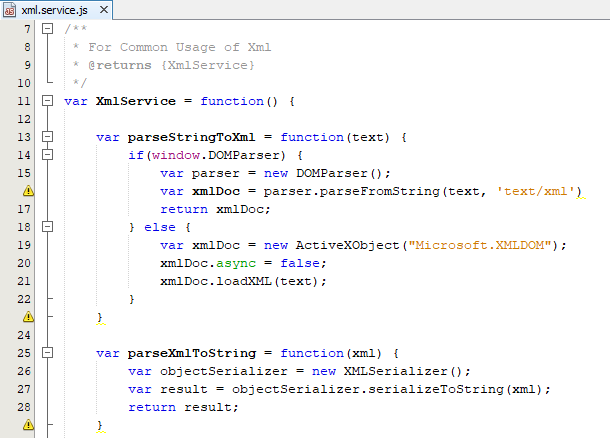


1. Hiện thực XmlService (Module core):

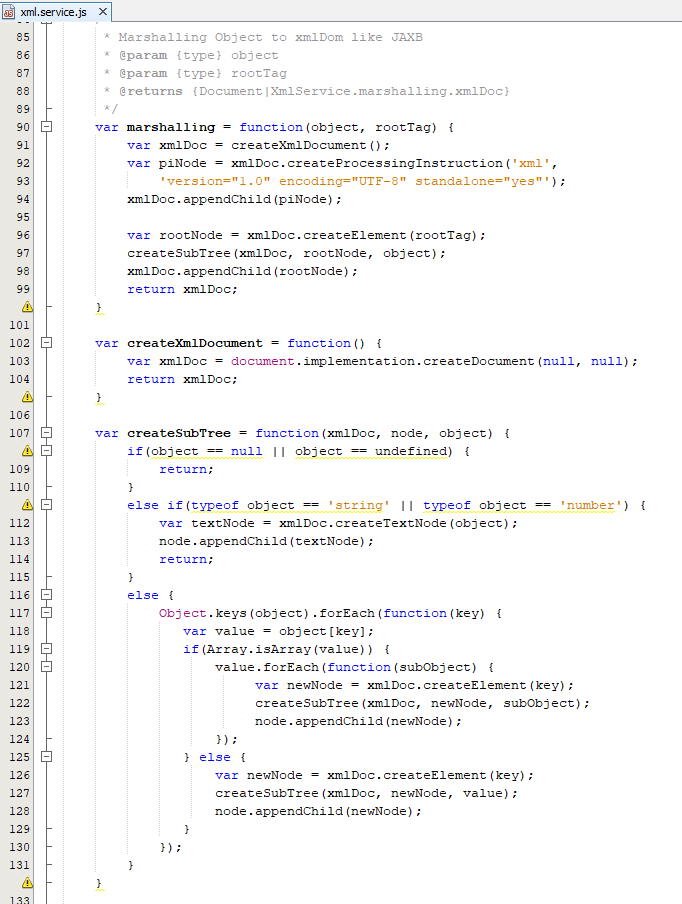
* Mục tiêu của mô hình này là xây dựng một mô hình tương tác với dữ liệu XML. Vì thế chúng ta sẽ hiện thực một module hỗ trợ trong việc xử lý XML.
* Đầu tiên, chúng ta cũng tạo 1 file dành riêng cho module XmlService. Sau đó đăng ký module đó vào hệ thống.



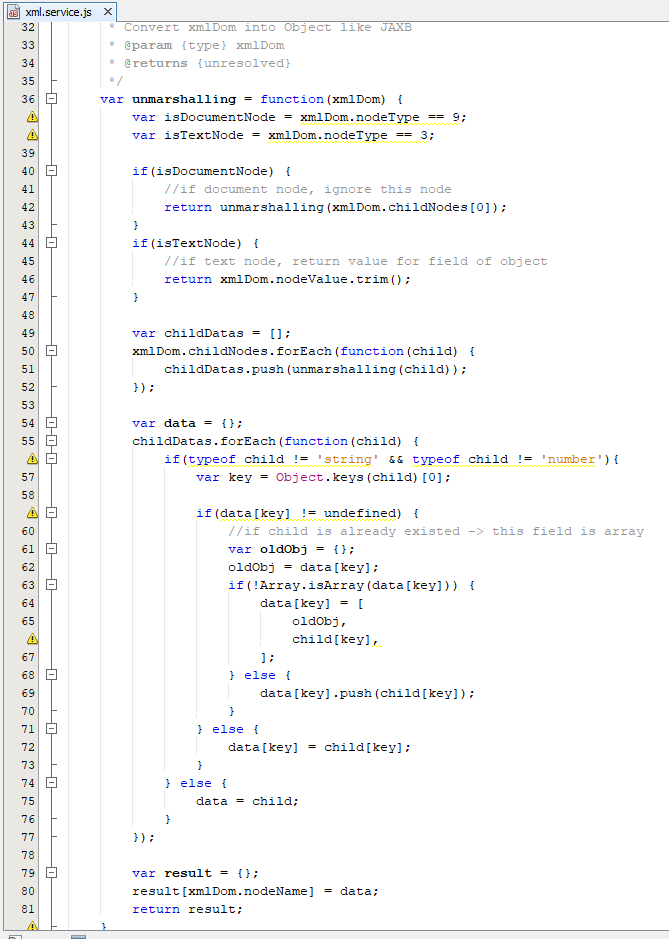
* Thao tác cơ bản nhất khi xử lý dữ liệu XML đó là parse từ String sang XML Dom và ngược lại:



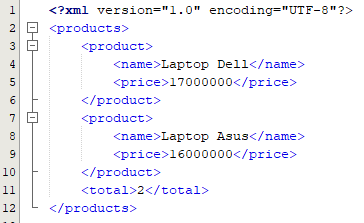
* Để tiện cho việc sử dụng XML, chúng ta hiện thực hiện marshalling và unmarshalling giống như JAXB.
  + Marshalling: Đây là quá trình chuyển đổi từ Object sang XML Dom.



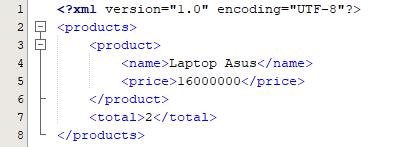
* + Unmarshalling: Đây là quá trình chuyển đổi từ XML Dom sang object.



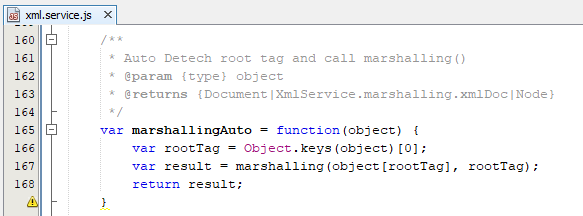
* Ví dụ kết quả của unmarshalling:



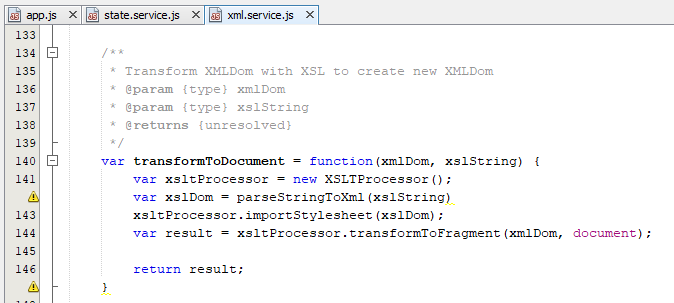
* Ta có thể thấy được kết quả đáng mong đợi, tuy nhiên function này không phân biệt được chỗ nào là object, chỗ nào là array, chỉ phân biệt được khi là array khi có 2 tag sibling giống nhau. Vì thế, mỗi lần unmarshalling array thì chúng ta cần kiểm tra object đó có phải là array không trước khi sử dụng.
* Ví dụ function không nhận ra array (Ở servlet, field product là array, nhưng sau khi unmarshalling thì product là object):

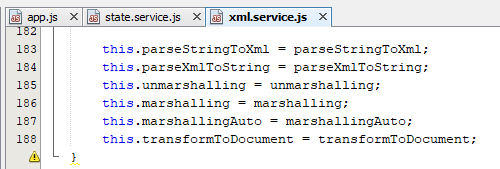
* + Để quá trình marshall và unmarshall được xuyên suốt, tức là khi nhận được XML từ API, biến thành object, rồi từ object chuyển đổi lại thành XML như lúc ban đầu thì chúng ta cần thêm 1 function để hoàn thiện pipeline đó.



* Hiện thực hàm transform XML và XSL thành Html



* Public các function trong module:

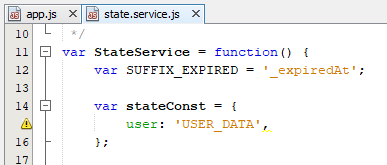


1. Hiện thực StateService (Module core):

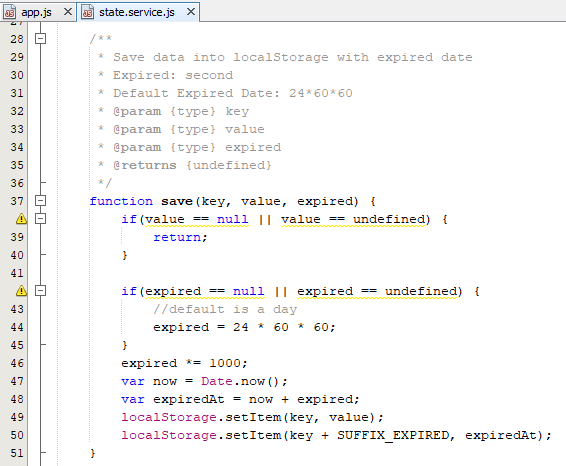
* Module này giúp ứng dụng có thể giao tiếp được với nhau khi chuyển trang, lưu trữ dữ liệu của toàn bộ ứng dụng, hay thực hiện caching ở client.
* Cơ chế: lưu dữ liệu vào localStorage cùng với ngày hết hạn dữ liệu.
* Tạo Module và đăng ký vào hệ thống



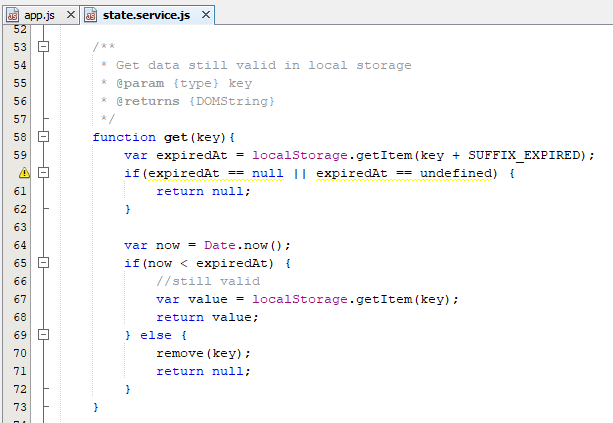
* Tạo một object lưu trữ tên của dữ liệu được lưu trữ:



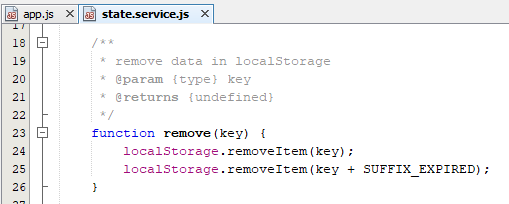
* Tạo function lưu dữ liệu với một khóa được truyền vào với thời hạn mặc định là 1 ngày.



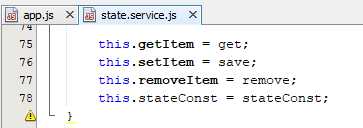
* Hiện thực function lấy dữ liệu của khóa được truyền vào:



* Hiện thực hàm xóa dữ liệu của khóa được truyền vào:



* Public các function của StateService:



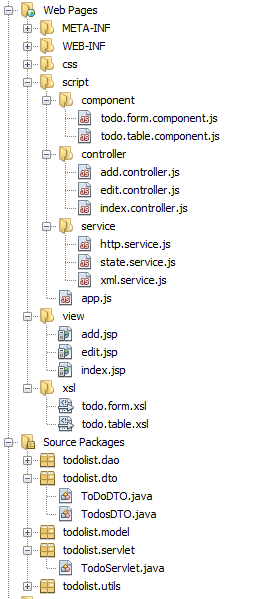
* Vậy là chúng ta vừa tạo ra được cấu trúc cơ bản của mô hình module cho client, sau đây chúng ta áp dụng mô hình này trên để làm một ứng dụng nhỏ.

1. **Áp dụng mô hình module vào ứng dụng To Do List**
2. Mô tả ứng dụng:

* Cho phép xem danh sách các Todo.
* Cho phép thêm, xóa, sửa Todo.

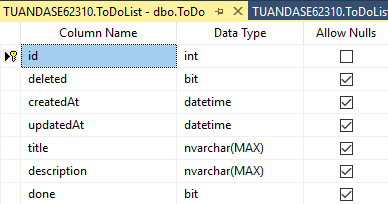
1. Cấu trúc project:

* Để tiện cho việc theo dõi, chúng tôi cấu trúc của các file trong project sẽ dùng trong bài viết:



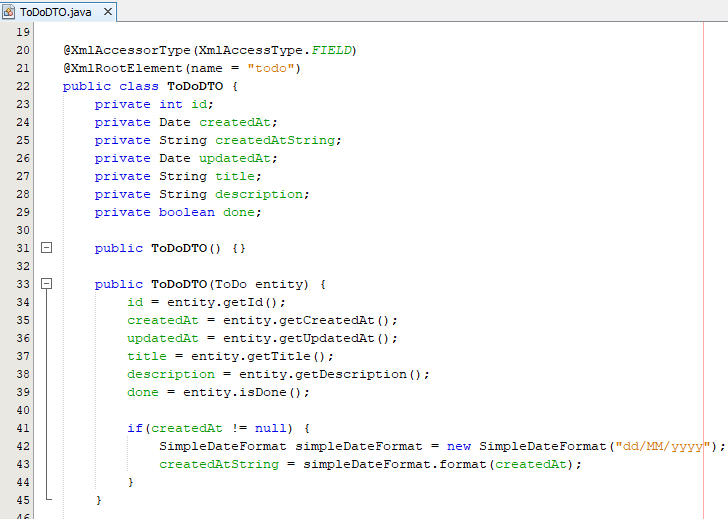
1. Database:

* Project này sẽ sử dụng chỉ một bản trong database để lưu dữ liệu của To Do List.

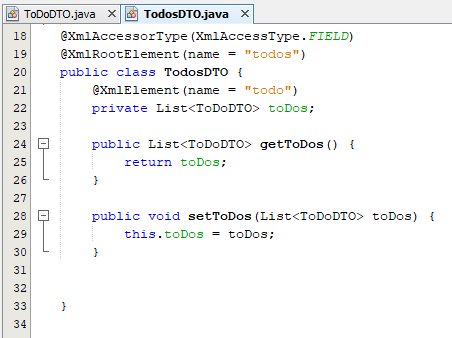


1. Hiện thực phần Back End:

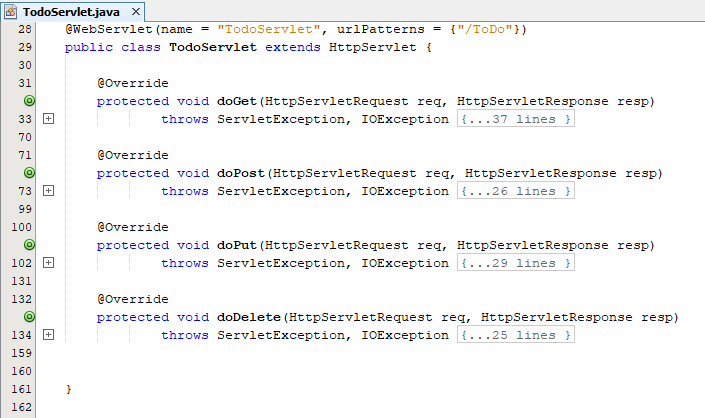
* Bài viết này chỉ chú trọng vào việc hiện thực ở client, cho nên chúng tôi chỉ hướng dẫn bước để tạo API trả về XML và nhận XML để xử lý.
* Trong project này, chúng ta chỉ có 1 resource (ToDo), vì thế chúng ta sẽ tạo 1 restful API để thực hiện các thao tác CRUD.
* Hiện thực DTO để gởi, nhận data:



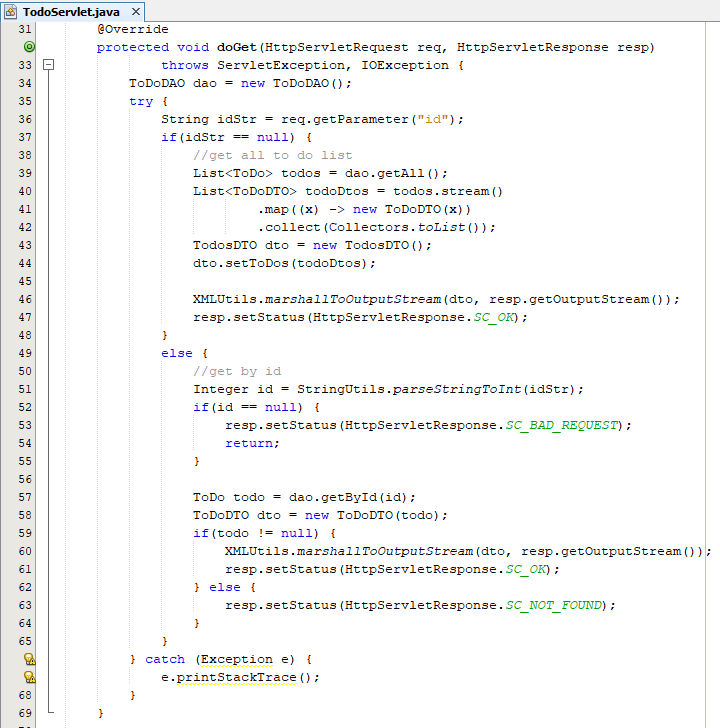
* Chúng ta phải tạo 1 DTO để khai báo cho JAXB biết cách marshall, unmarshall danh sách To do.



* Tiếp theo, chúng ta tạo một servlet đóng vai trò restful API:



* Hiện thực method GET của API để lấy dữ liệu về:



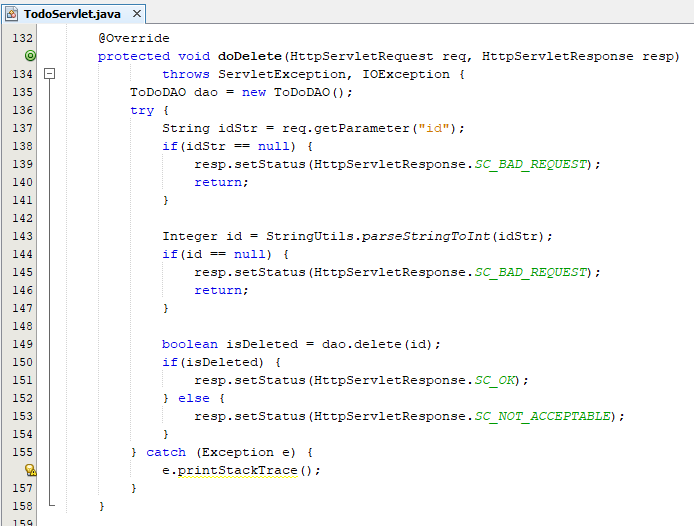
* Hiện thực method POST để thêm mới dữ liệu:



* Hiện thực method PUT để cập nhập data



* Hiện thực method DELETE để xóa data:

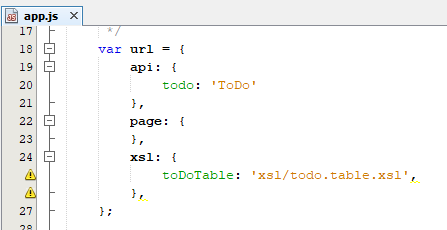


1. Hiện thực phần Client:

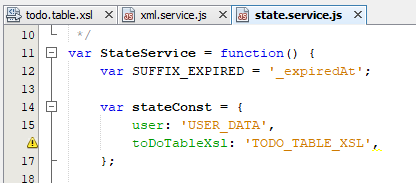
* Hiện thực TodoTableComponent để xử lý, hiển thị danh sách to do.
  + Có nhiều cách để sinh ra HTML element để xuất ra giao diện, tuy nhiên trong bài viết này chúng ta sẽ sử dụng template XSL đế sinh ra HTML từ data XML. Vì vậy, chúng ta tạo 1 file template todo.table.xsl để thực hiện việc render cho component này.



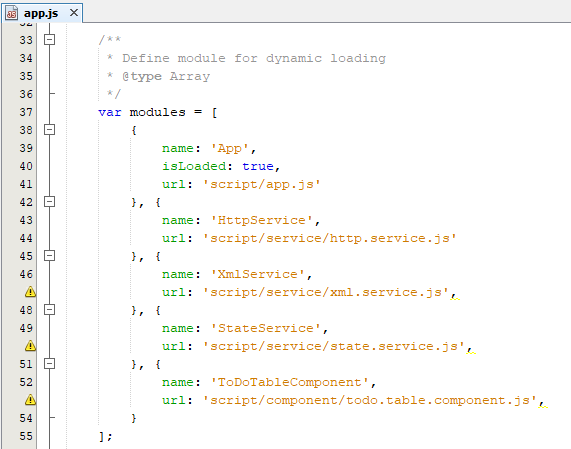
* + Để sử dụng template này, chúng ta khai báo URL để lấy template này về, việc khai báo như vậy sẽ dễ dàng để bảo trì, nâng cấp ứng dụng của chúng ta.



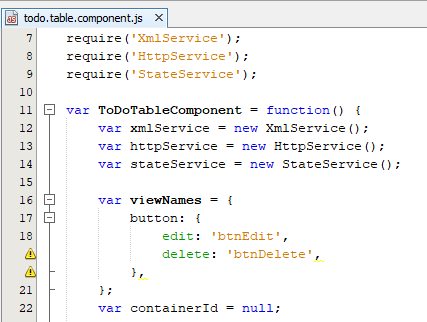
* + Khi sử dụng template này, chúng ta nên cache lại template này để tăng performance của ứng dụng. Do đó, chúng ta thêm 1 khóa cho template này ở trong StateService.



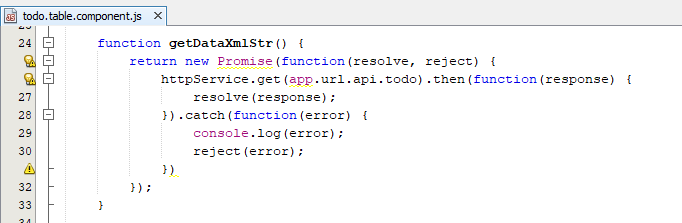
* + Tạo TodoTableComponent, sau đó đăng ký component vào ứng dụng.



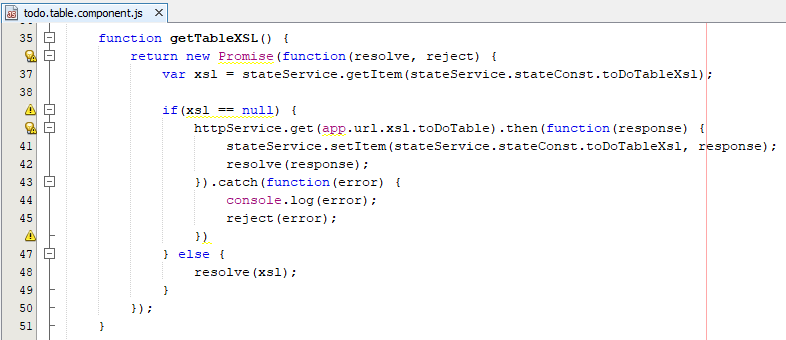
* + Khai báo dependency, các element id, các private variable của component:



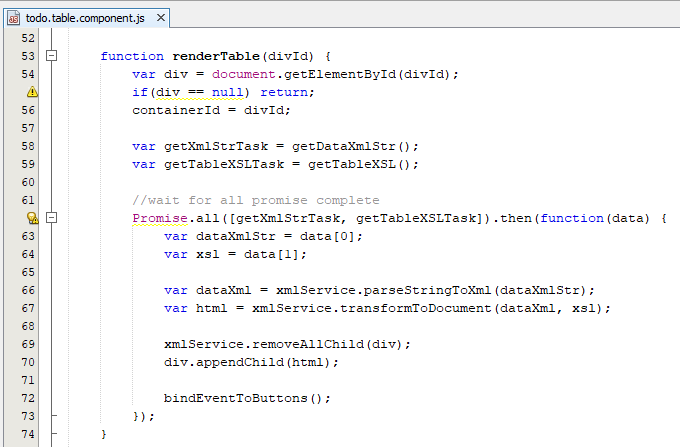
* + Tạo function để lấy dữ liệu Xml về dưới dạng String. Vì function này chạy chạy bất đồng bộ, nên chúng ta sẽ trả về [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).



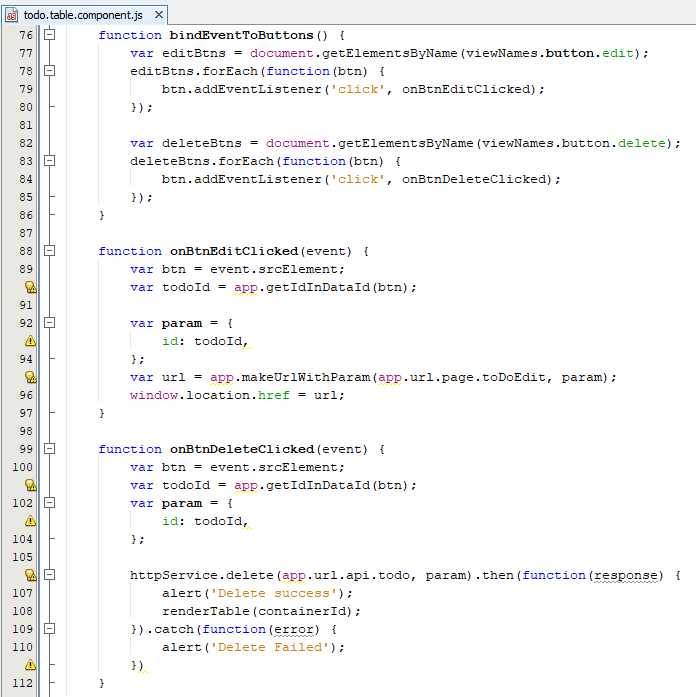
* + Tạo function để lấy xsl. Đây là function bất đồng bộ, nên chúng ta sẽ trả về [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).



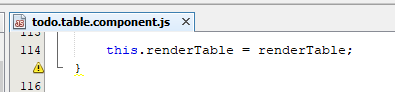
* + Tạo function để render component vào trong Element có Id được truyền vào.



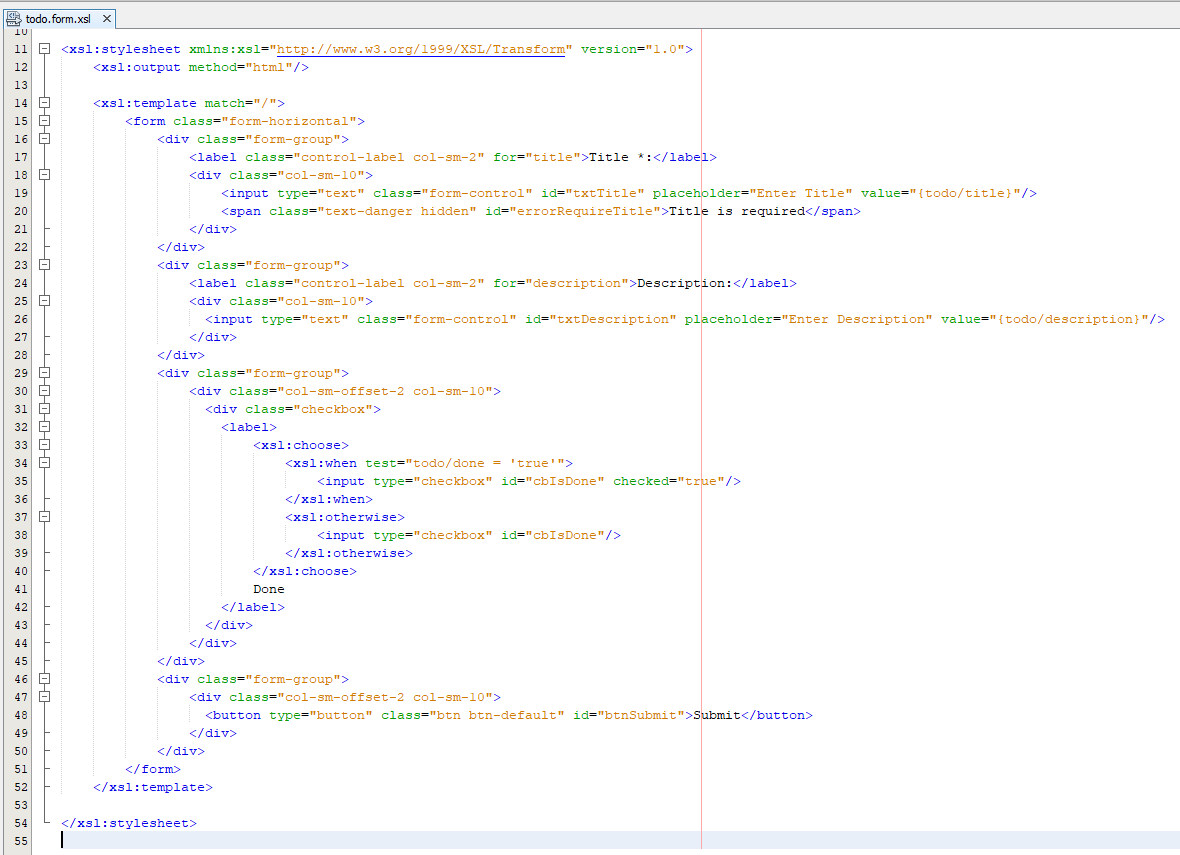
* + Sau khi render, component sẽ thêm các event để xử lý logic của các element đó.



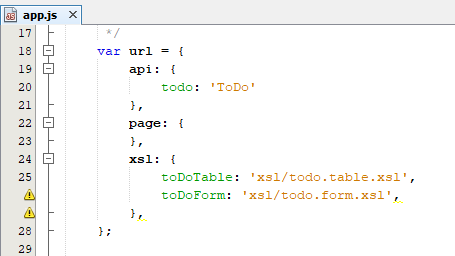
* + Chúng ta public function render của component ra để sử dụng trong controller.



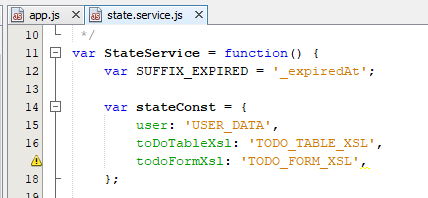
* Hiện thực TodoFormComponent để xử lý logic của form Todo, hiển thị form Todo:
  + Tương tự như TodoTableComponent ở trên, component này cũng cần có 1 file xsl để đảm nhận việc render html. Vì thế chúng ta tạo 1 template todo.form.xsl.



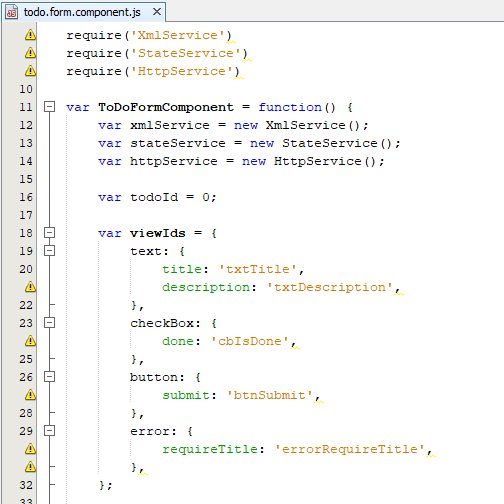
* + Khai báo URL của template:



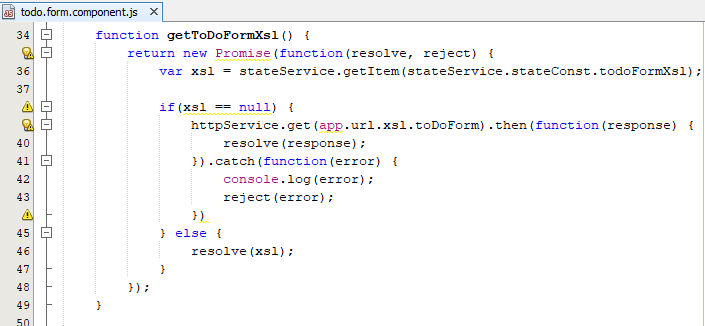
* + Khai báo khóa để cache template lại:



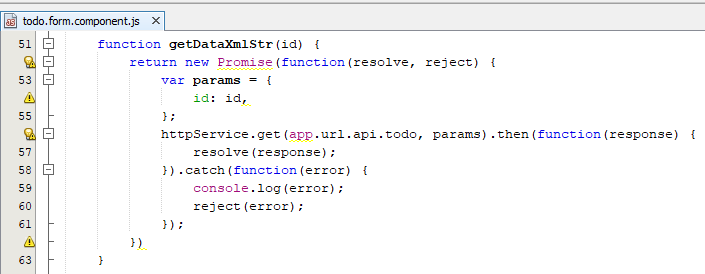
* + Khai báo component, khai báo dependency, private variable của component:



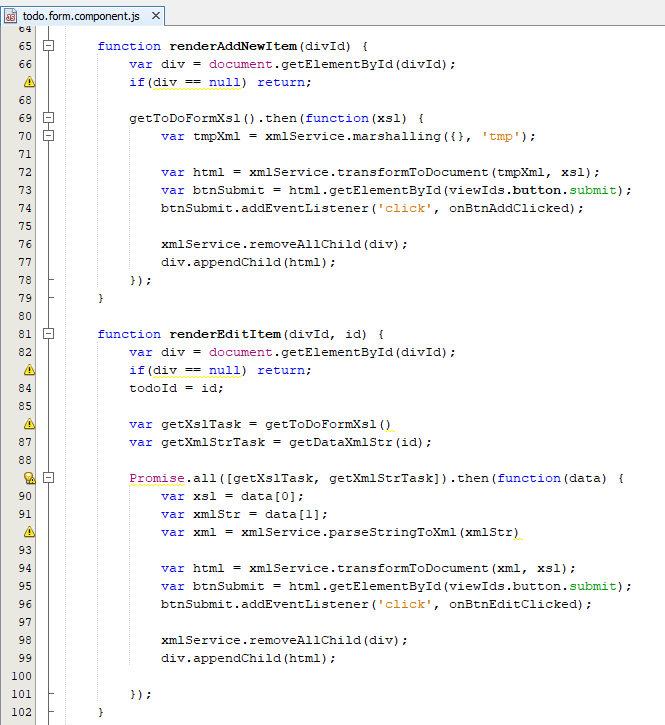
* + Tạo function lấy template data:



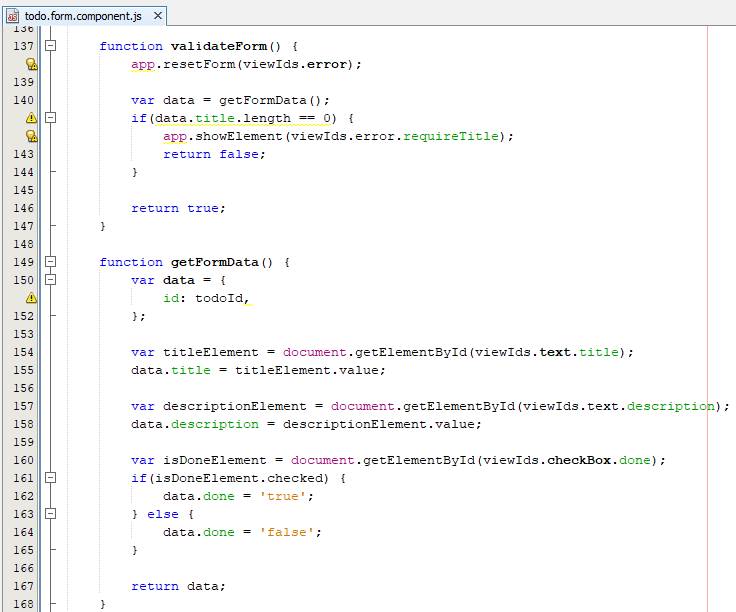
* + Bởi vì component này render form sử dụng chung mục đích cho thêm mới, chỉnh sửa To do. Nên chúng ta tạo hàm để lấy dữ liệu To do từ API để khởi tạo cho giao diện chỉnh sửa.



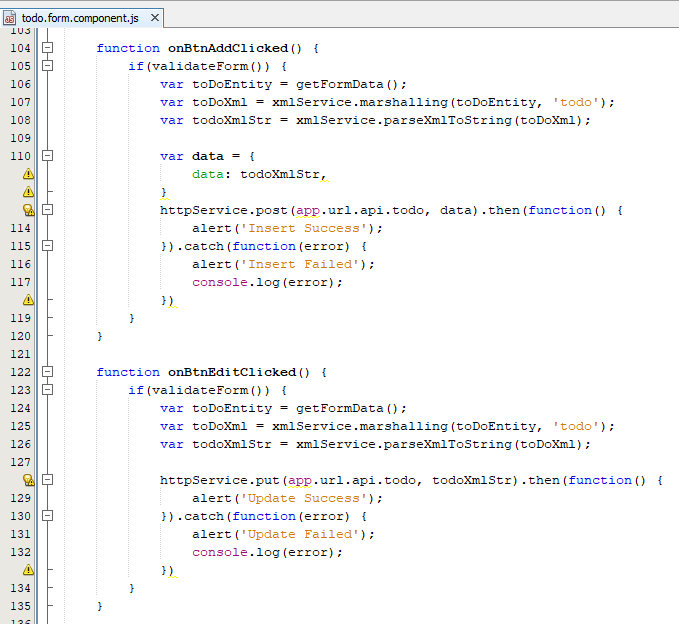
* + Tạo hàm để render tạo mới To do, chỉnh sửa To do:



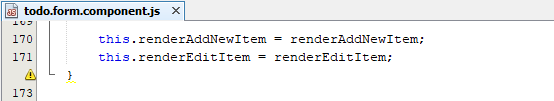
* + Trước khi submit form, chúng ta phải validate data.



* + Tạo function submit data để thêm hoặc chỉnh sửa:



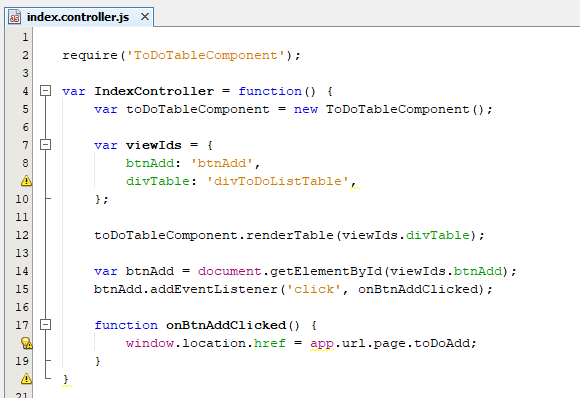
* + Public function render form để sử dụng trong controller:



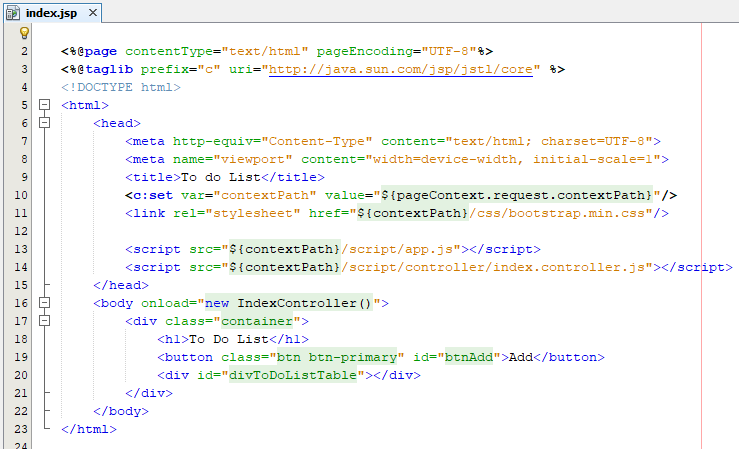
* Hiện thực giao diện và controller:
  + Theo lý thuyết, controller không được sử dụng lại bởi bất cứ module nào hết, vì vậy chúng ta không cần đăng ký vào App.js
  + Giao diện của chúng ta cần 3 file jsp: index.jsp, add.jsp, edit.jsp
  + Chúng ta tạo 3 file jsp và đăng ký vào App.js



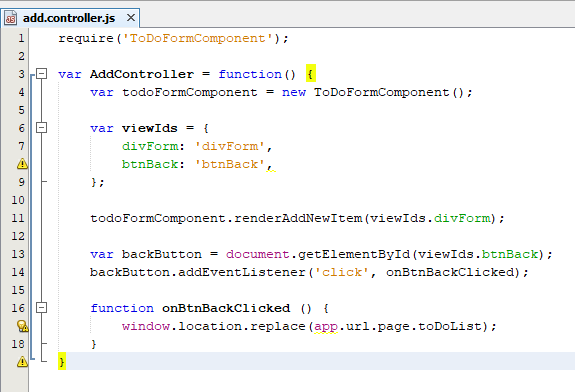
* + Hiện thực trang IndexController để để hiển thị danh sách Todo:



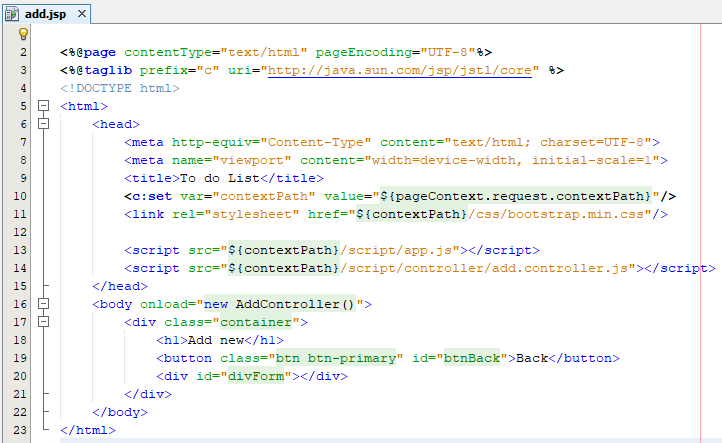
* + Hiện thực trang index.jsp sử dụng IndexController để hiển thị danh sách Todo:



* + Tạo AddController để xử lý logic của trang thêm mới Todo:



* + Tạo trang add.jsp sử dụng AddController để thêm mới Todo



* + Tạo EditController để xử lý logic chỉnh sửa Todo.

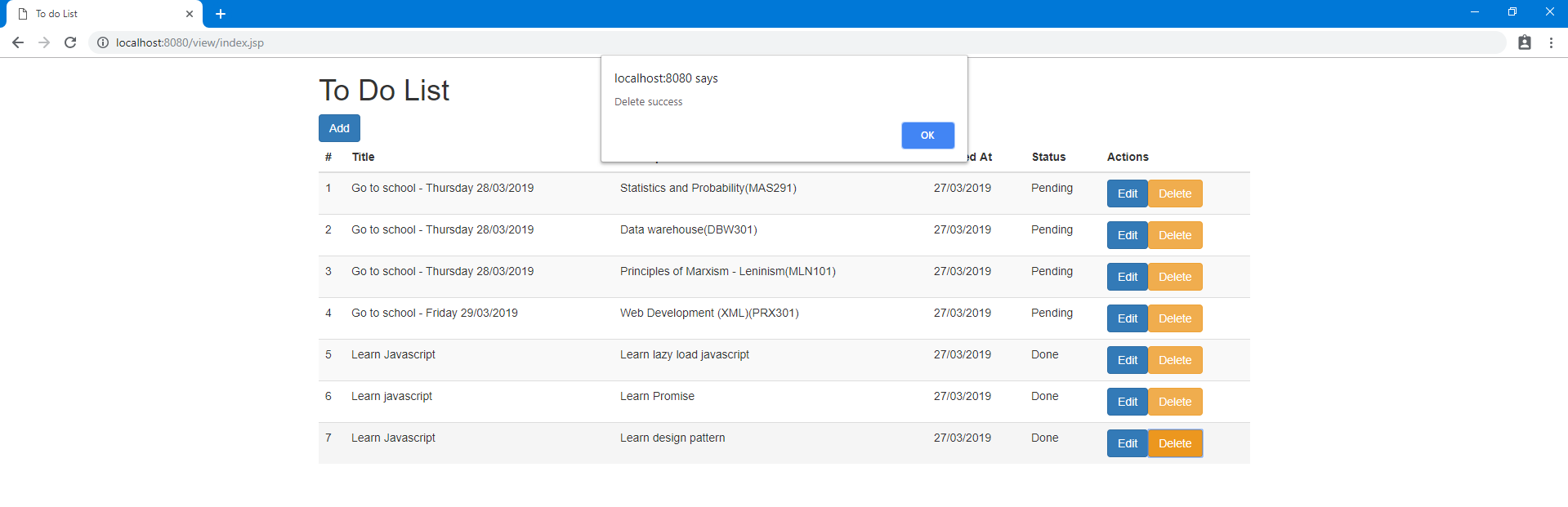


* + Tạo edit.jsp sử dụng EditController để thực hiện chỉnh sửa To do:

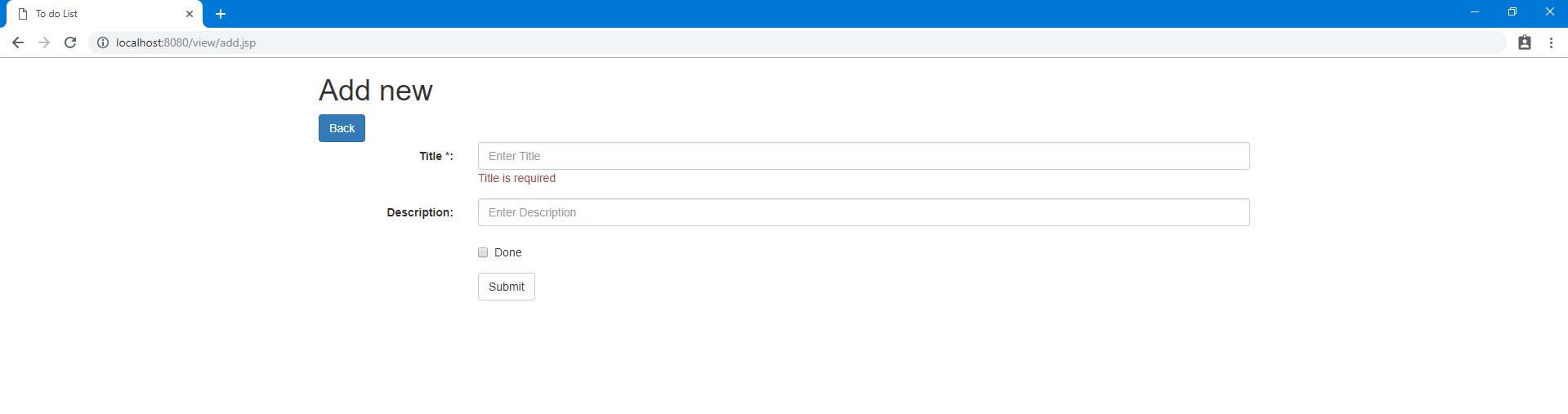


1. Kết quả:

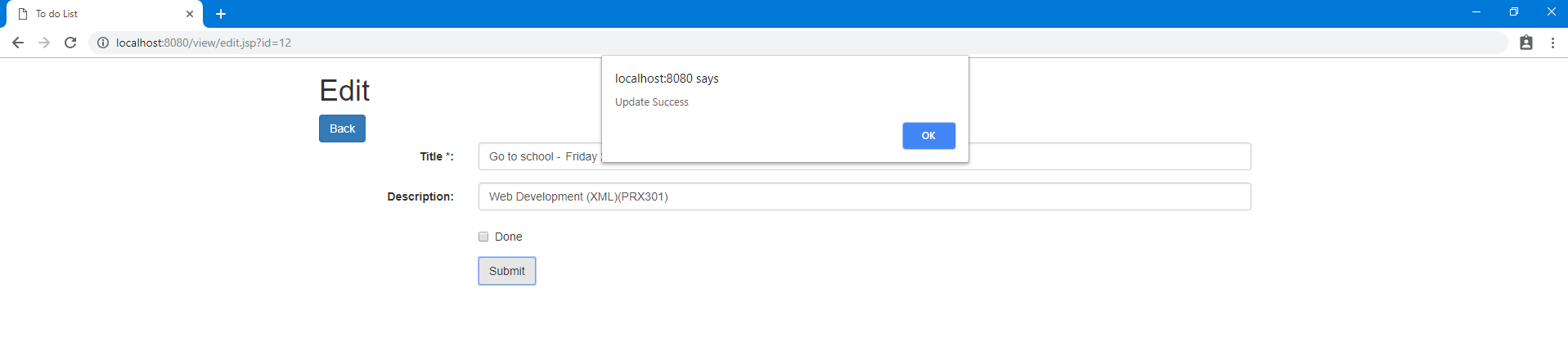
* Trang danh sách To do:



* Trang thêm mới To do:



* Trang Edit To do:



1. **Kết thúc chủ đề.**

* Giả sử trong demo To do List ở trên, chúng ta thêm một tính năng phân quyền người dùng, chúng ta chỉ cần viết thêm 1 module để xử lý việc authorize người dùng. Rồi sau đó chỉ cần sử dụng module đó ở trong các controller nào cần authorize. Do đó, chúng ta có thể thấy, framework này đã làm việc bảo trì hệ thống trở nên dễ dàng hơn.
* Chúc mừng quý vị đã hoàn tất việc tạo ra một mô hình để phát triển ứng dụng ở phía client bằng cách chia nhỏ thành các module và thực hiện load các module khi cần thiết.
* Rất mong sự đóng góp của quý vị về chủ đề này nhằm tạo ra một mô hình vanilla javascript thuận lợi nhất để phát triển ứng dụng web.
* Hẹn gặp quý vị ở các chủ đề khác.