SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

TD8: Servlets, DOM et Virtual DOM

Equipe Enseignante PC3R

01/04/2021

Objectif: Donner le code entier (serveur + client) de deux micro-applications. Le code doit être suffisant pour que les applications puissent être développées et testées entièrement dans une IDE intégrant le déploiement de serveurs (par exemple, Eclipse).

Attention: On cherche seulement ici à démontrer des mécanismes de bases de la programmation Web. On ne se sert pas d'API de haut-niveau, on ne fait pas attention à la sécurité ou à l'efficacité.

1 Approche Ressource: Bibliothèque

1.1 Description

On modélise une application d'une bibliothèque. Du point de vue utilisateur:

- les utilisateurs peuvent s'authentifier.
- l'espace personnel des utilisateurs leur indique les emprunts en cours.
- on peut rechercher des livres par titre ou par auteur, on voit combien d'exemplaires il reste pour chaque livre.
- on peut emprunter ou rendre des livres à distance.

1.2 Serveur

1. Le serveur manipule des entités de trois sortes:
   - des **utilisateurs**, avec leur nom, login et mot de passe (en clair, pour simplifier, mais on se souviendra que c'est une très mauvaise pratique) et **une structure contenant les emprunts en cours** de cet utilisateur.
   - des **livres**, avec titre, auteur, date de publication et **nombre d'exemplaires disponibles**.
   - des **emprunts**, qui font référence à un livre, un utilisateur et qui contiennent une deadline.

   Ecrire trois **classes** correspondant aux trois **entités**, contenant les méthodes permettant de gérer les actions d'emprunter ou de rendre des livres.
2. Pour simplifier, on n'utilisera pas de base de données et d'ORM. On stockera les informations dans le conteneur de servlet (c'est mieux que de les stocker dans un Servlet, car elles ne disparaissent pas quand il est déchargé, mais ça reste une solution non-réaliste, puisqu'elles disparaissent quand le serveur est coupé/relancé).
   Donner la structure du projet Web correspondant.
3. Ecrire trois classes manipulant des **collections** de chacune des trois entités. On implémentera des méthodes basiques d'accès aux éléments des collections (recherche des livres par titre, recherche des utilisateurs par login, recherche des emprunts par identifiant, ...).
4. Ecrire une classe **peuplant** les collections (une pour chaque entité) avec quelques exemples de livres et utilisateurs.
5. Ecrire le **servlet** pour les **livres**. A l'initialisation, le servlet accède au **contexte de servlet** pour vérifier si les collections existent déjà sur le serveur. Si c'est le cas il les récupère, sinon il les crée (à l'aide de la classe de la question précédente) et les inscrit au contexte (elles seront disponibles pour les autres servlets).
   Le servlet des livres permet de récupére (`GET`) un livre (ou une liste de livres) par identifiant, titre ou auteur. Le résultat est sérialisé en JSON (on pourra utiliser *Gson*, par exemple) et attaché à la réponse.

6. Ecrire le **servlet** pour les **Utilisateurs**. Il permet de récupérer les informations (en JSON) d'un utilisateur à partir de son login (`GET`) et de vérifier un couple login/mot de passe (`POST`).
7. Ecrire le **servlet** pour les **Emprunts**. Il permet de récupérer des informations sur un emprunt à partir de son identifiant (`GET`). Il permet aussi de créer un emprunt (`PUT`) (en mettant à jour le nombre d'exemplaires disponibles d'un livre, et en ajoutant l'emprunt à la structure de l'utilisateur concerné) et de rendre un livre (`DELETE`).
   Attention: Un objet Java `Utilisateur` contient une structure d'objets Java `Emprunt` et un objet Java `Emprunt` contient comme attribut son emprunteur (un objet Java `Utilisateur`). Ces circularités sont courantes en Java, mais au moment de la sérialisation (traduction d'un objet en JSON), elles peuvent provoquer un comportement indésirable (boucle infinie).
8. Ecrire un fichier `web.xml` pour lier les Servlets à des URLs simples.

1.3 Client

1. Créer une page `html` vide, qui se contente de charger un script utilisateur.
2. Editer le script utilisateur: en utilisant uniquement des manipulations du DOM (de l'API DOM basique), créer un formulaire d'authentification avec un champ login et un champ mot de passe (dans un véritable projet, il est préférable de charger un sous-fichier `html` directement, ou d'utiliser une interface de haut-niveau).
3. Lier la soumission du formulaire à une requête AJAX qui teste l'authentification (en insérant mot de passe et login dans la *query string*, même s'il ne faut jamais le faire dans une véritable application), qui devient à nouveau disponible si l'authentification échoue, et qui passe à la suite si elle réussit.
   Remarque: On utilisera (au moins) trois technologies différentes pour les requêtes AJAX de ce client: le JS naturel (`XMLHttpRequest`), *JQuery* (`$.ajax`) et l'API *Fetch* (`fetch`).
4. l'authentification réussie débloque un "accueil" contenant le nom de l'utilisateur, la liste des emprunts en cours et deux champs de recherche de livres, par nom et par auteur.
5. l'utilisation d'un des champs de recherche déclenche un appel AJAX qui met à jour la page en incluant la liste des livres correspondant à la recherche (avec le nombre d'exemplaires disponibles).
6. On ajoutera un bouton à chaque résultat de recherche permettant d'emprunter un livre, ce qui doit conduire à un rechargement de la partie "emprunts en cours" de la page. On ajoutera à chaque emprunt en cours un bouton "rendre".

2 Approche Service: Parkings

2.1 Description

On modélise une application d'information sur les parkings d'une municipalité.

- l'application affiche toutes les secondes le nombre de place restantes dans chaque parking d'une ville, en coloriant ce nombre avec un code couleur indiquant le taux de remplissage relatif du parking.
- l'application affiche quel est le meilleur parking vers lequel un utilisateur doit se diriger (sur des critères arbitraires de taux de remplissage, de nombre absolu de places vides ou de dérivée tierce du nombre de places disponibles)

2.2 Serveur

1. Donner le plan du projet, écrire une classe `Parking` qui modélise un parking (nom, capacité max, occupation courante, opérations pour faire rentrer ou sortir une voiture) et une classe `Parkings` qui contient une collection de parkings.
2. Pour modéliser le comportement des capteurs des parkings on crée une classe `ModeleDeCommune` qui implémente `Runnable` et qui constitue en un thread qui fait entrer ou sortir des voitures aléatoirement dans les parkings d'une collection. Ecrire cette classe.
3. Ecrire un **servlet d'initialisation** qui sera appelé à la connexion. Quand il est contacté, ce Servlet regarde dans le contexte si une collection de parking existe déjà, si ce c'est pas le cas, il en crée une (4 ou 5 parkings de tailles différentes) et lance le thread modele (ce qui commence à générer un

flux d'entrées et sorties dans les parkings). Il renvoie un objet `JSON` correspondant à la collection de parkings.

4. Ecrire un **servlet de service de places restantes** qui implémente une fonction qui prend en entrée un identifiant de parking et renvoie le nombre de places restantes dans ce parking. Ecrire ensuite un **servlet de meilleur parlking** qui renvoie un parking correspondant au meilleur endroit pour se garer.
5. Editer le `web.xml` pour lier les servlets à des URLs simples.

2.3 Client

1. Créer une page `HTML` vide qui invoque un script utilisateur qui commence par contacter le servlet d'initialisation pour récupérer la liste des parkings, puis crée une liste correspondant aux noms de chacun des parkings.
2. En utilisant `React`, attacher à chaque élément de la liste un élément `React` qui, chaque seconde, effectue un appel AJAX au serveur pour connaitre le nombre de places restantes dans le parking associé, et colorie ce nombre avec un code couleur exprimant le taux de remplissage du parking. Ajouter un élément qui conseille, en temps réel, le meilleur parking dans lequel se garer.