Applications Web dynamiques
Enterprise Java Applications

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

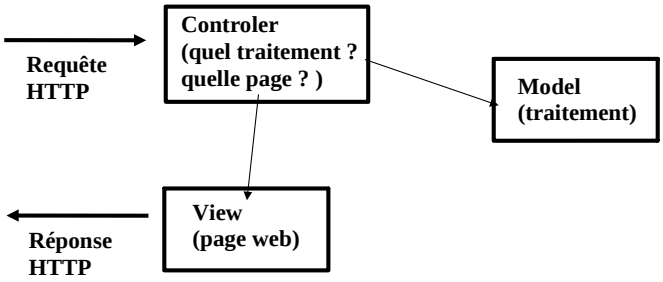
1

L'objectif a été de fournir des technologies logicielles permettant de faciliter le développement des applications Web dynamiques. Nous avons vu précédemment la structuration suivant le modèle MVC, avec le Contrôleur (des servlets), la Vue (des JSP) et le Modèle (pour l'instant des objets Java).

Dans la suite, nous nous intéressons à structurer la partie Modèle (métier).

Rappel : Modèle MVC

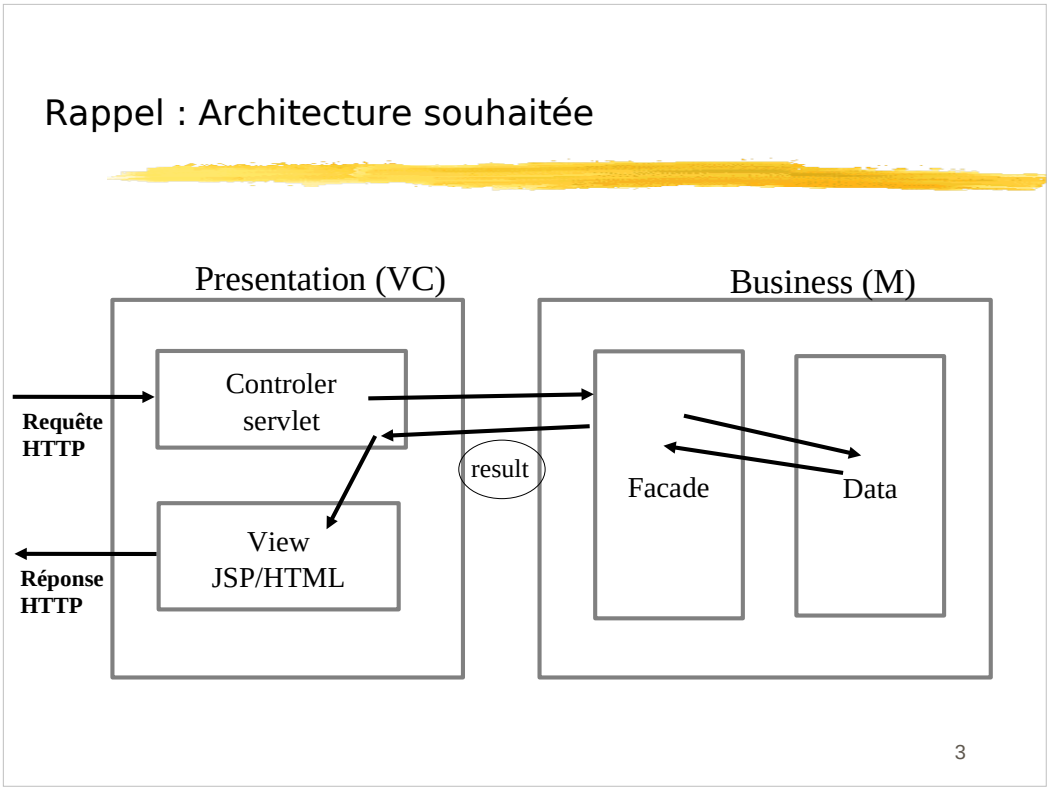
- Model View Controler
- Séparation entre
  - Le Contrôleur : servlet qui aiguille les requêtes
  - La Vue : pages JSP pour l'affichage à l'écran
  - Le Modèle : les classes (beans) qui traitent les données



2

Souvenez vous de ce schéma illustrant le modèle MVC.

Le Contrôleur reçoit la requête HTTP, appelle le Modèle pour faire le traitement correspondant à la requête (et récupère un résultat), puis appelle la Vue pour générer la page Web présentant les résultats.



On avait également vu ce schéma, dans lequel la partie frontale à gauche (qu'on appelle aussi souvent front-end) correspond au servlet container qui héberge les servlets du Contrôleur et de la Vue (les JSP étant compilées en servlet).

La partie à droite (souvent appelée back-end) correspond au Modèle aussi appelé code métier, car cela implante la logique de l'application indépendamment du fait que ce soit une application Web ou pas. Cette partie back-end est divisée entre Facade et Data. La Facade correspond au code de l'application, qui fournit une interface utilisée par le Contrôleur. La partie Data correspond à la gestion des données que l'on conserve. Cette partie Data repose en général sur une base de données.

Préambules

- Auparavant, mon cours reposait sur les EJB (Enterprise Java beans) qui fournit :
  - Des session beans pour structurer la Facade
    - Accessibles avec Java RMI ou des Web Services
  - Des entity beans pour gérer des Data dans une base de données
- Je suis passé à Spring
  - En raison de sa popularité dans les entreprises
  - Il fournit plus ou moins les mêmes concepts
    - Des REST web services pour construire la Facade
    - Des entity beans (on parle de Java Persistence API)

4

Mon cours reposait avant sur des EJB (Enterprise Java Bean).

Les EJB fournissent notamment :

- des session beans pour implanter la Facade. Ce sont des objets Java qui permettent notamment de faciliter la gestion en cluster (réplication sur plusieurs machines pour parallélisation). Ils sont accessibles depuis une servlet en local dans le même container Tomcat , ou à distance avec Java RMI, ou encore sous la forme de web services.

- des entity beans pour implanter les données. Ces sont des objets Java qui sont automatiquement enregistrés/chargés dans/depuis la base de données.

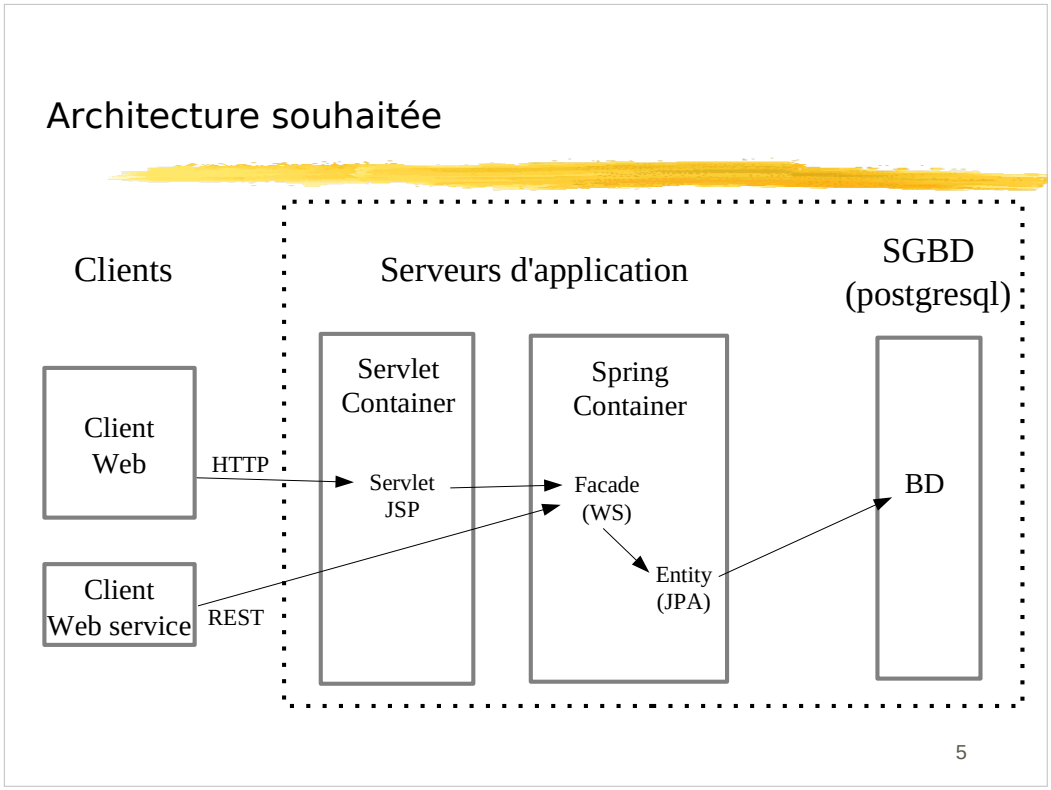
Les EJB restent une technologie très utilisée.

Mais devant la popularité de Spring, je basculé mon cours vers celui-ci.

Spring fournit les mêmes concepts avec quelques variantes technologiques.

La Facade est construite un web service REST (avec Spring que vous avez vu en cours Intergiciels).

Spring fournit la notion d'entity beans dans ses services REST pour la persistence de données dans des BD.



Quand on instancie cette architecture dans un environnement tel que celui proposé par Spring (mais on retrouve les mêmes principes dans d'autres environnements), on obtient l'architecture ci-dessus.

On voit qu'on a toujours les servlets et JSP dans le servlet container (Tomcat), par contre la Facade est implantée sous la forme d'un WS (Web Service) dans un autre container (Spring container, un autre Tomcat). On peut aussi tout exécuter dans le même containter (1 seul Tomcat).

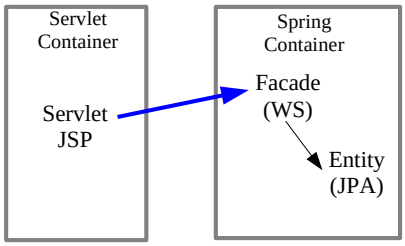
Notons que comme la Facade est implantée comme un WS, elle peut être accédée depuis n'importe quelle application, pas seulement l'application web implantée par le servlet container.

Enfin, la gestion des données repose principalement sur une base de données (ici Postgresql). Mais pour faciliter l'utilisation de la base de données depuis la Facade, Spring fournit une implantation de JPA (Java Persistence API) sous la forme d'Entity. Cela simplifie énormément le boulot : on ne manipule plus la BD avec des requêtes SQL, mais on voit des objets Java stockés dans la BD.

Nous allons voir tous ces aspects un peu plus en détail. La présentation courante se concentre sur la Facade et la présentation suivante se concentrera sur la gestion des données (avec des Entity).

Structuration de la Facade avec Spring

- Voir le cours Intergiciel du semestre précédent
  - La servlet appelle la Facade sous la forme d'un web service REST
  - La Facade peut être appelée depuis n'importe quelle application (pas que la servlet)
  - La Facade est juste une fonction (réplication dans un datacenter)
- Exemple sur l'application bancaire



6

Dans la suite, je m'appuie sur les WS REST qui ont été présenté dans le cours Intergiciel au semestre précédent.

On implante la Facade sous la forme d'un WS Spring.

On utilise resteasy pour implanter la partie client dans la servlet (qui appelle la Facade).

Je m'appuie sur l'application bancaire de gestion de comptes vue dans la présnetation précédente.

Exemple : les données manipulées (Data)

```
public class Compte {
   private int num;
   private String nom;
   private int solde;

   public Compte() {}

   public Compte(int num, String nom, int solde) {
       this.num = num; this.nom = nom; this.solde = solde;
   }

   public String toString() {
       return "Compte [num="+num+", nom="+nom+", solde="+solde+"]";
   }

   // setters and getters
```

7

Les données manipulées ne changent pas. Ce sont des comptes bancaires. On a ici une simple classe Compte.

Exemple : implantation de la Facade (service REST)

```
@RestController
public class Facade {
    private Map<Integer, Compte> comptes = new Hashtable<Integer, Compte>();

    @PostMapping("/addcompte")
    public void addCompte(Compte c) {
        comptes.put(c.getNum(), c);
    }

    @GetMapping("/consultercomptes")
    public Collection<Compte> consulterComptes() {
        return comptes.values();
    }

    @GetMapping("/consultercompte")
    public Compte consulterCompte(@RequestParam("num") int num) throws RuntimeException {
        Compte c = comptes.get(num);
        if (c == null) throw new RuntimeException("Compte introuvable");
        return c;
    }
```

8

Vous avez ici l'implantation de la Facade comme un WS Spring.

Rappels :

- les méthodes sont accessibles avec des requêtes GET (@GetMapping) ou POST (@PostMapping). Ces annotations indiquent comment ces méthodes sont représentées dans l'URL d'appel du WS.

- les objets Java reçus ou retournés sont sérialisés en JSON

- les annotations @RequestParam indiquent comment les paramètres sont représentés dans l'URL d'appel du WS.

A noter que nous avons toujours une Hashtable pour gérer les données. Nous verrons plus tard (dans la présentation suivante) comment gérer les données dans une BD.

Exemple : implantation de la Facade (service REST)

```
    @PostMapping("/debit")
    public void debit(@RequestParam("num") int num, @RequestParam("montant") int montant)
                                                                    throws RuntimeException {
        Compte c = consulterCompte(num);
        if (c.getSolde()<montant) throw new RuntimeException("Solde insuffisant");
        c.setSolde(c.getSolde()-montant);
    }

    @PostMapping("/credit")
    public void credit(@RequestParam("num") int num, @RequestParam("montant") int montant) {
        Compte c = consulterCompte(num);
        c.setSolde(c.getSolde()+montant);
    }

    public Facade() {
        addCompte(new Compte(1, "dan", 2000));
        addCompte(new Compte(2, "alain", 4000));
        addCompte(new Compte(3, "luc", 6000));
    }
}
```

9

Exemple : le controleur (interface cliente RestEasy)

```
@Path("/")
public interface Facade {

    @POST
    @Path("/addcompte")
    @Consumes({ "application/json" })
    public void addCompte(Compte c);

    @GET
    @Path("/consultercomptes")
    @Produces({ "application/json" })
    public Collection<Compte> consulterComptes();

    @GET
    @Path("/consultercompte")
    @Produces({ "application/json" })
    public Compte consulterCompte(@QueryParam("num") int num) throws RuntimeException;

    @POST
    @Path("/debit")
    public void debit(@QueryParam("num") int num, @QueryParam("montant") int montant) throws RuntimeException;

    @POST
    @Path("/credit")
    public void credit(@QueryParam("num") int num, @QueryParam("montant") int montant);

}
```

10

Dans la servlet, nous utilisons ReastEasy pour appeler la Facade.

Nous décrivons donc l'interface du WS en RestEasy.

Exemple : le controleur (servlet)

```
@WebServlet("/Controller")
public class Controller extends HttpServlet {

   final String path = "http://localhost:8080/bank-spring";

   Facade facade;

   public Controller() {
      super();
      ResteasyClient client = new ResteasyClientBuilder().build();
      ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));
      facade = target.proxy(Facade.class);
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                    throws ServletException, IOException {
      try {
         String action=request.getParameter("action");
         if (action.equals("consulter")) {
            int num=Integer.parseInt(request.getParameter("num"));
            request.setAttribute("num", num);
            request.setAttribute("compte", facade.consulterCompte(num));
         }
```

11

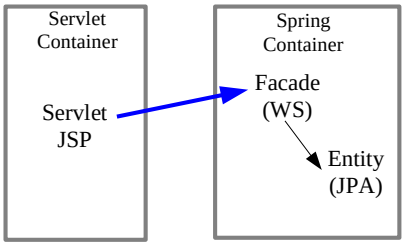
La seule modification dans la servlet concerne la Facade.

Au lieu d'instancier la classe Facade, nous initialisons RestEasy pour obtenir un stub (proxy) qui implante l'interface Facade.

Ainsi, tous les appels de méthode sur la variable "facade" font un appel au WS qui implante la Facade.

Conclusion

- On a une séparation plus claire entre
  - Le front-end : comprend la vue (V) et le controleur (C)
    - Sous la forme de servlet/JSP
  - Le back-end : le modèle
    - Sous la forme d'un WS REST

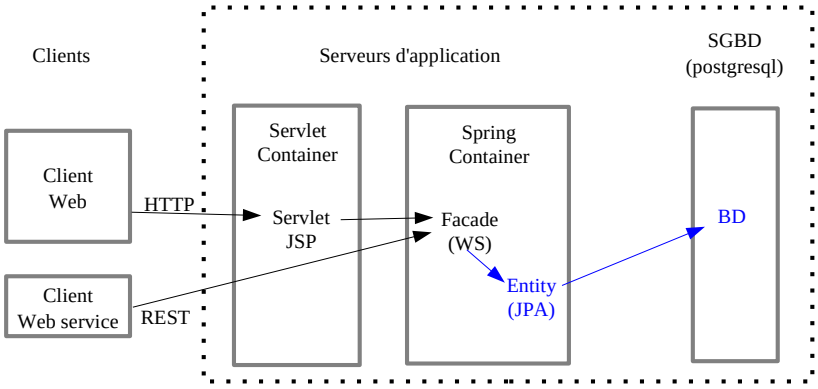


12

De cette façon nous avons une séparation plus claire entre le frond-end et le back-end.

Conclusion

- Suite : on voudrait gérer les données dans une BD
  - Sans avoir à programmer avec JDBC



13

Dans la présentation suivante, nous allons voir comment gérér les données dans une BD.