
Analiza și proiectarea sistemelor software

Curs 10

PLAN CURS

Proiectarea aplicațiilor Web

- Principii
- Modelul proiect pentru aplicații Web
- MVC (Model-view-controller)
- MVC Frameworks
- Proiectare aplicații MVC
- Concept-based URI
- Ruby on Rails MVC Framework
- Spring Web MVC Framework

MODEL – VIEW - CONTROLLER

MVC – stil arhitectural potrivit pentru aplicații centrate pe *utilizator* și *informații*.

Separarea aspectelor esențiale ale unei aplicații interactive

- logica de prezentare (V – view)
- logica de control (C – controller)
- logica business și datele (M – model)

Cuplarea slabă a acestor componente.

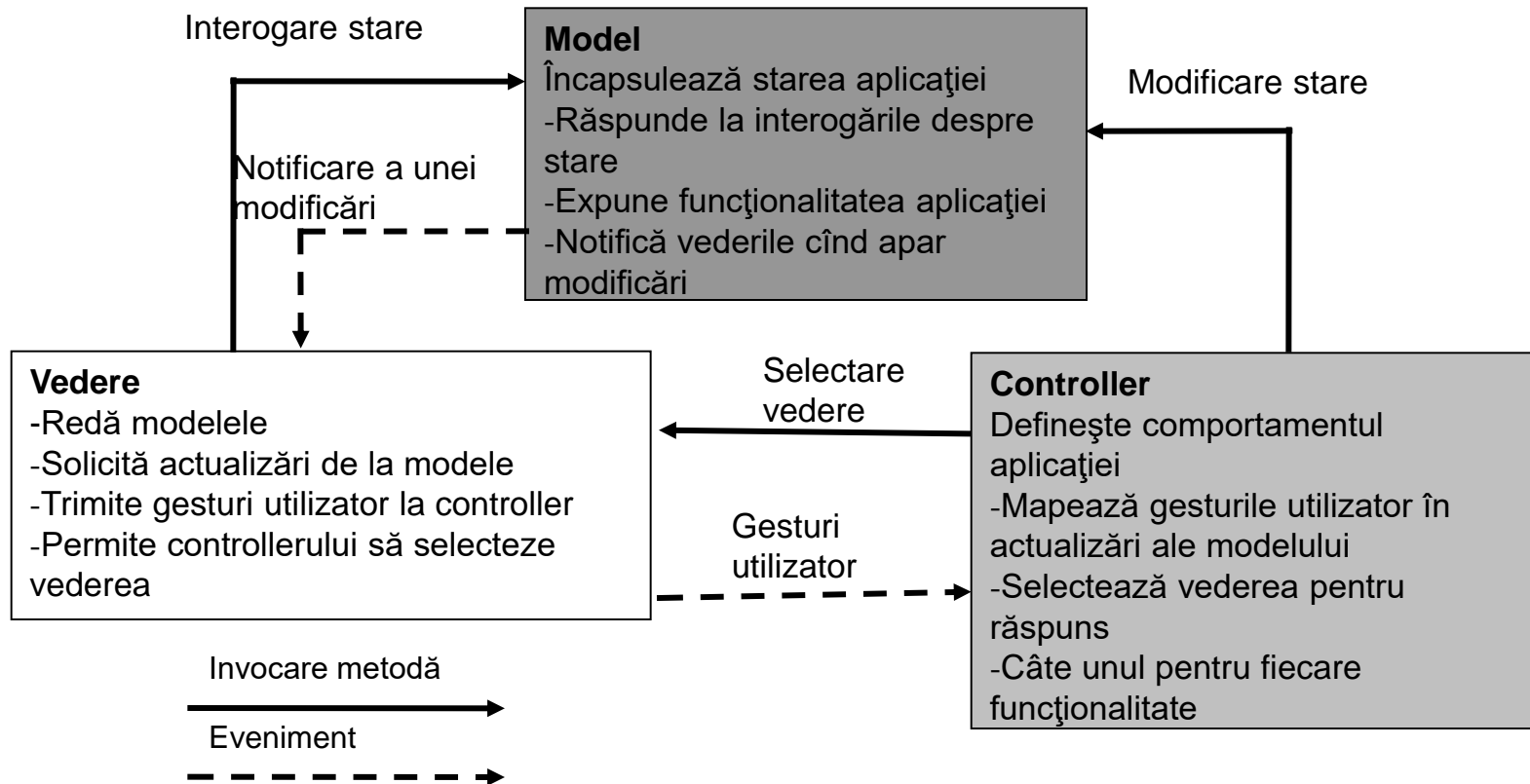
M – *independent* de V și C

COMPONENTELE MVC

ARHITECTURA MVC

- *model* - conținutul și logica de procesare specifice aplicației
 - obiectele conținut
 - accesul la surse externe de date și informații
 - funcționalitatea de procesare specifică aplicației
- *view* – conține funcțiile de interfață și permite
 - prezentarea conținutului și a logicii de procesare
 - accesul la funcționalitatea de procesare solicitată de utilizatorul final.
- *controller*
 - gestionează accesul la model și la view
 - coordonează fluxul de date dintre acestea.

COMPONENTELE MVC



Decuplează vederile (prezentarea) de model (datele).
Stabilește un protocol de tip subscribe / notify între ele.

MVC

ȘABLOANE DE PROIECTARE INCLUSE

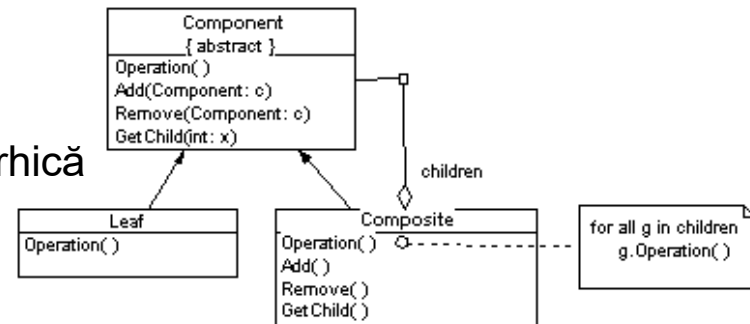
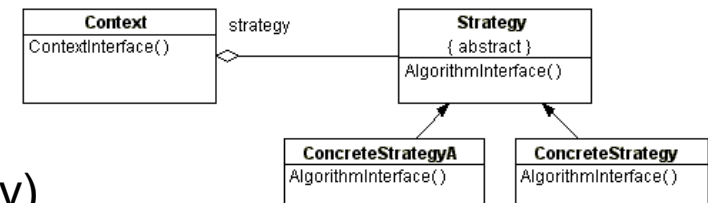
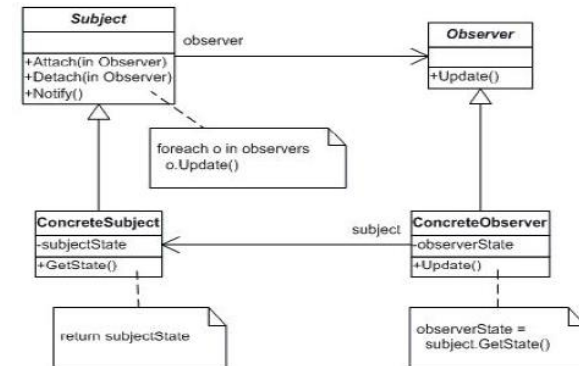
Șabloanele de proiectare pot fi combinate pentru a crea structuri noi:

- specifice unor aplicații
- șabloane noi

Exemplu : MVC (Model View Controller)

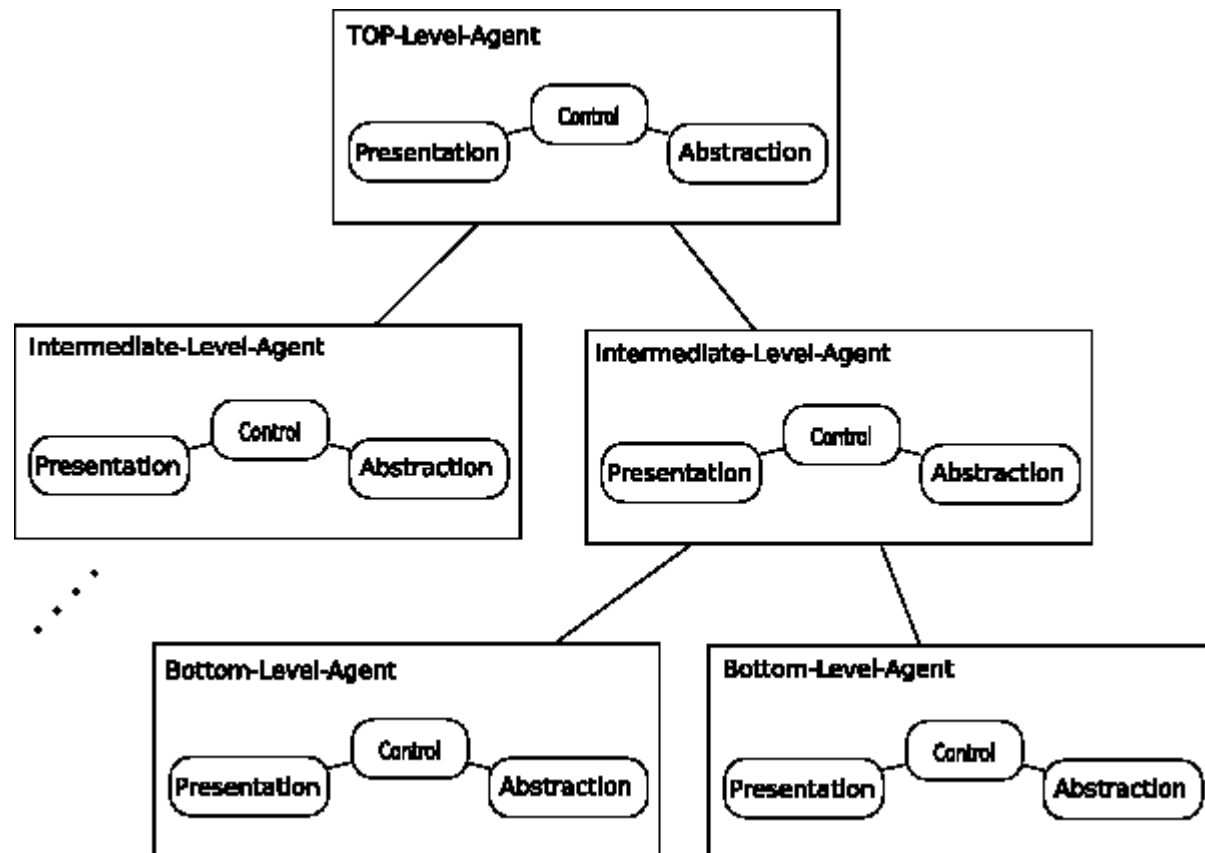
compus din :

- **Observer** – Model (Subject) și View (Observer)
- **Strategy** – View (Context) și Controller (Strategy)
 - vederile sunt configurate cu strategii implementate în controller-e => flexibilitate în modificarea statică/dinamică a comportamentului
- **Composite** – View
 - administrare și actualizare structură internă ierarhică de elemente GUI.



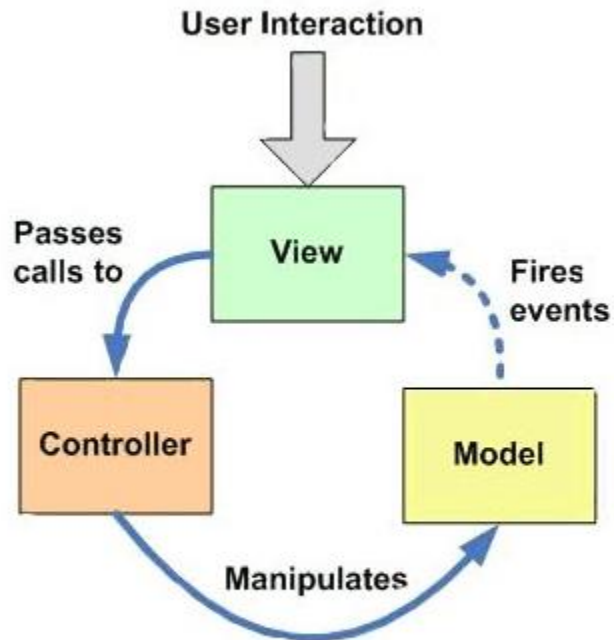
VARIANTE MVC

HMVC – hierarchical MVC (similar cu PAC – presentation-abstraction-controller)

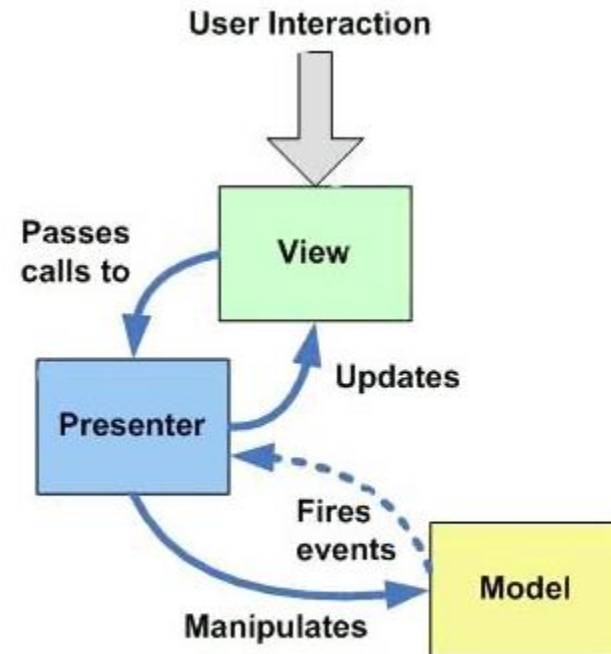


VARIANTE MVC

MVC vs MVP – model-view-presenter (ex. .NET, JFace, Swing, Nette Framework)



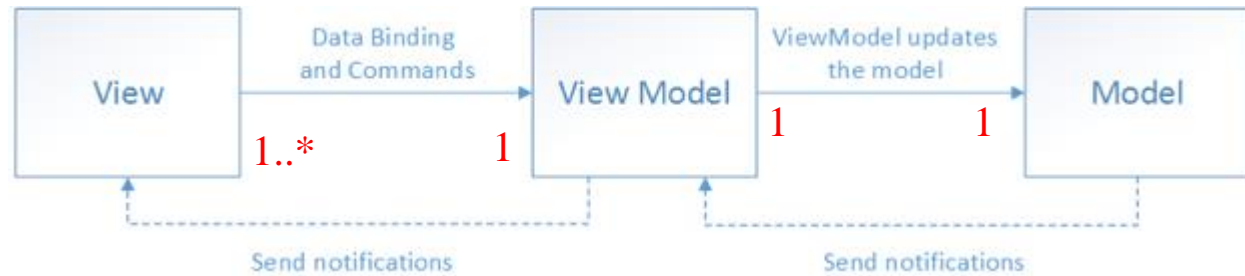
Model-View-Controller



Model-View-Presenter

VARIANTE MVC

MVVM – model view view model*



Separare suplimentară : prezentare (View) de logică prezentare (View Model).

ViewModel :

- View abstractizat, independent de platforma UI
- Expune metode și comenzi pentru manipulare date din Model.
- Legare declarativă View cu date și comenzi din ViewModel
- Sincronizare automată

* Utilizat în framework-uri .NET și JavaScript.

PLAN CURS

Proiectarea aplicațiilor Web

- Principii
- Modelul proiect pentru aplicații Web
- MVC (Model-view-controller)
- MVC Frameworks
- Proiectare aplicații MVC
- Concept-based URI
- Ruby on Rails MVC Framework
- Spring Web MVC Framework

FRAMEWORK-uri (CADRE)

- Infrastructură *schelet* cu ***implementare tipică*** (implementează un *mecanism intern*), conținând ***puncte de extensie*** pentru a permite adaptarea la o problemă de domeniu specifică.
- Punctele de extensie permit integrarea de clase și funcționalitate ***specifice problemei***.
- În proiectare OO un framework este un set de clase (inclusiv interfețe) ***cooperante***.

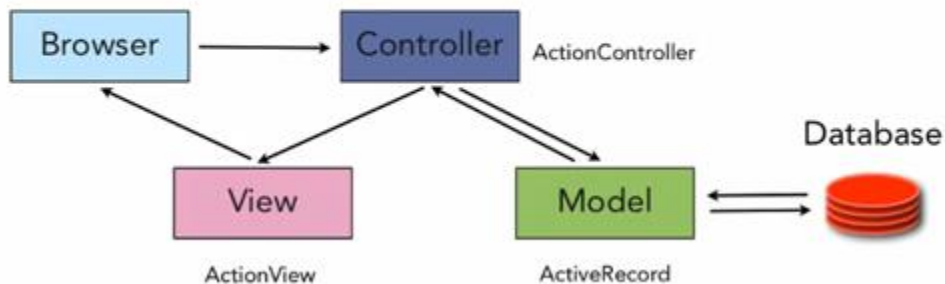
WEB MVC

Arhitectura MVC clasică nu este potrivită pentru aplicațiile web deoarece modelul nu poate transmite modificările către view (pagină web) folosind șablonul Observer.

De ce ?

Model2 – MVC = varianta arhitecturală specifică aplicațiilor web

MVC WEB ARCHITECTURE



WEB MVC

Arhitectura ***aplicației*** corespunde stilului MVC

Distincție între componentele aplicației MVC și repartizarea lor pe o arhitectură web.

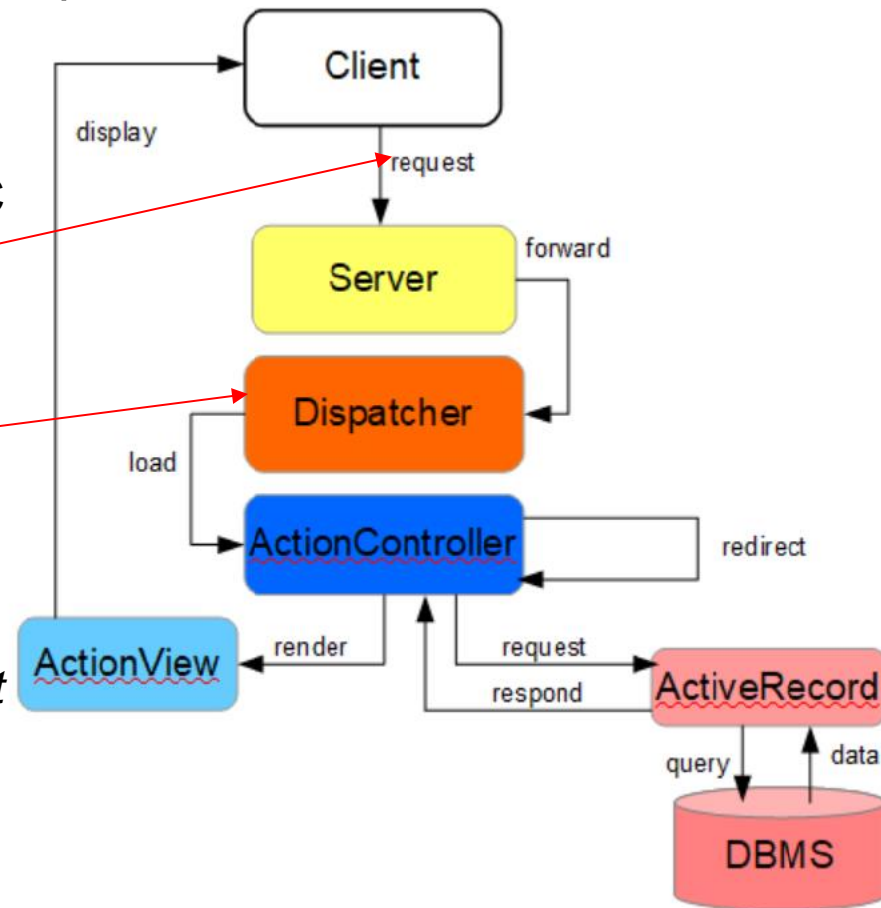
- Inițial – “thin” client, aproape toată logica M, V și C pe server
- Acum – tehnologii client mature au condus la componente M,V,C parțial executate pe client*

*(în framework-uri ca AngularJS, Ember.js, JavaScriptMVC, Backbone).

MVC WAF tipic

Caracteristicile comune majorității web MVC frameworks:

- request-driven
- organizate în jurul unui servlet central care dispeceerizează cererile către controller-e
- expresie a șablonului de proiectare *Front Controller** - punct centralizat de intrare pentru manipulare cereri



Unele includ suport pentru lucru cu servicii web (ex. RoR, Spring)

*ActionServlet / Struts, FacesServlet / JSF, AppServlet / DDUI, DispatcherServlet / Spring

AVANTAJE MVC

- Promovează *proiectarea OO* – dezvoltare aplicație prin extindere framework cu componente M, V și C.
- M, V, C – *independente*; multiple V și C pot fiecare folosi unul din mai multe limbaje specifice și se pot integra într-o aplicație complexă și disponibilă pe diferite tipuri de terminale și în diferite limbi.
- *Specializare dezvoltatori* independenți pentru fiecare tip de componentă M, V sau C.
- *Soluții simple* pentru categorii importante de *modificări*:
 - interfață linie de comandă – reutilizare M și C fără V
 - creare demo – C, V și un mockup M
 - transformare aplicație în serviciu – înlocuire V cu interfață serviciu
 - modificarea suportului de persistență – un nou translator pentru model

Cum ?

GHID ALEGERE FRAMEWORK

- gradul de modularizare
- funcțiile helper oferite
- cât de ușor se poate adauga codul propriu
- calitatea documentației (wiki-based nu asigură acoperire consistentă de la trăsătură la trăsătură (feature))
- analiza comentariilor de pe Internet ale utilizatorilor

Evaluare formativă

1. Care este elementul central al unui framework MVC tipic și ce rol are acesta?
2. Fie o aplicație web dezvoltată pe un framework MVC. Precizați cum ați realiza următoarele sarcini :
 1. Crearea unui demo
 2. Transformarea interfeței grafice în interfață în mod linie de comandă.
 3. Transformarea aplicației în serviciu web
 4. Modificarea suportului de persistență

<https://forms.gle/Wd7kCvJtoMjmnrDV8>

PLAN CURS

Proiectarea aplicațiilor Web

- Principii
- Modelul proiect pentru aplicații Web
- MVC (Model-view-controller)
- MVC Frameworks
- Proiectare aplicații MVC
- Concept-based URI
- Ruby on Rails MVC Framework
- Spring Web MVC Framework

PROCESUL DE PROIECTARE MVC

1. Definirea *modelului domeniului* (entitățile fundamentale și relațiile dintre ele) ⇒ **clasele de domeniu (M)**
2. Definirea *cazurilor de utilizare* (funcțiile principale ale aplicației) ⇒
 - **controller-ele (C)**
 - **interacțiunile browser – aplicație**⇒ **structura URL**
3. Crearea *prototipul GUI* ⇒ **pagini și porțiuni de pagini (V)**
4. Definirea *acțiunilor* (actions) ⇒ **metodele din C**

Categorii de acțiuni:

 - creare pagini web
 - realizare funcție apelând logica business din model și redirectare către altă pagină funcție de rezultat
5. Identificarea *paginilor statice*, care nu corespund acțiunilor de pe controller-e.

PROCESUL DE PROIECTARE MVC

Suportul oferit de framework

- legare model de baza de date
- legare model de forme web (HTML Forms)
- creare structură de fișiere și intrări în aceasta

PROIECTARE MODEL

Model = modelul domeniului

Ce conține modelul domeniului ?

Setul obiectelor ce reprezintă *entitățile* cu care aplicația operează și *relațiile* dintre acestea.

Doar secundar — majoritatea persistente în BD,
totuși modelul nu trebuie să fie un set de apeluri de funcții care înfășoară BD !!!

Obs. Framework-uri ORM (Object Relational Mapping) asigură *integrare cu baza de date* folosind *configurare explicită minimală*.

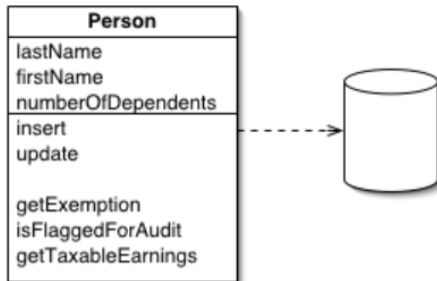
ORM

Abordări

Șablonul Active Record.

<https://www.martinfowler.com/eaCatalog/activeRecord.html>

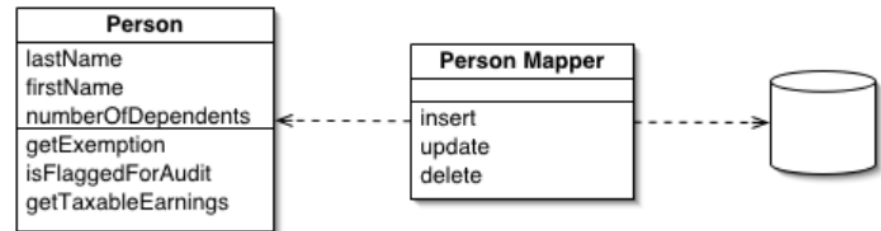
“Obiect ce împachetează un *rând dintr-o tabelă* din BD, încapsulează *accesul la BD* și adaugă *logică de domeniu* la datele respective” (Martin Fowler).



Șablonul DataMapper (DAO)

<https://www.martinfowler.com/eaCatalog/dataMapper.html>

Componentă intermediară ce asigură separare între obiectele din memorie și baza de date.



OBS. ActiveRecord este potrivit în cazul unei corespondențe directe între entitățile din modelul domeniului și tabelele din baza de date. Dacă modelul domeniului este mai complex (colecții, moștenire, etc) atunci DataMapper este mai potrivit.

<https://www.thoughtfulcode.com/orm-active-record-vs-data-mapper/>

ORM

Unele framework-uri folosesc **șablonul ActiveRecord** pentru a oferi **ORM**.

Ex.: Ruby on Rails, Laravel's Eloquent, Propel (Symfony), Yii Active Record, Django's ORM

Un obiect de acest tip împachetează un *rând dintr-o tabelă* din BD, încapsulează *accesul la BD* și adaugă *logică de domeniu* la datele respective (Martin Fowler).

Operațiile tipice ale unei clase model bazată pe `ActiveRecord`:

- creare instanțe din setul rezultat al unei interogări SQL
- crearea unei noi instanțe pentru a fi inserată ulterior în tabelă BD
- metode finder statice care includ interogări SQL tipice și returnează obiecte
- actualizarea BD cu inserarea datelor din obiect
- metode accesori (get/set attribute)
- logica business

Alte framework-uri folosesc **Data Mapping** pentru a oferi ORM.

Ex.: Java Hibernate, Doctrine2, SQLAlchemy in Python, EntityFramework for Microsoft .NET

PROIECTARE CONTROLLER

Definire *interacțiuni browser* ↔ *aplicație* folosind cazurile de utilizare și prototipul GUI.

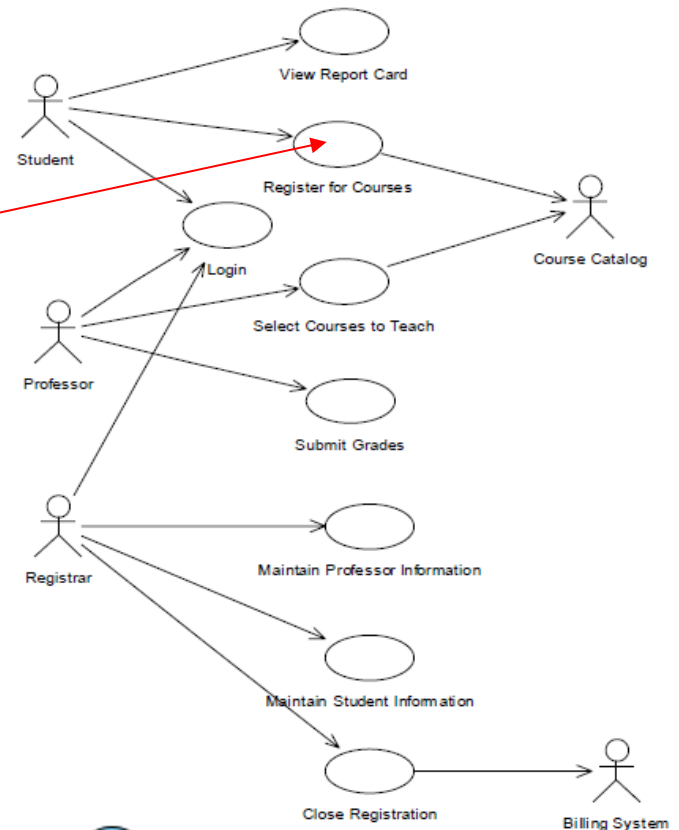
Creare câte un controller pentru fiecare caz de utilizare.

Rezultă structura URLs.

Controler-ul – apelează metodele business de pe obiectele din model și transferă rezultatele primite către view-ul corespunzător.



URL : www.RcApp.org/registerForCourses



PROIECTARE VIEW

- Detaliere *interacțiuni browser ↔ aplicație*
- Proiectare:
 - componentele fundamentale din GUI
 - aranjarea și compunerea lor

OBS.

- V – poate fi o pagină sau o parte dintr-o pagină; compozabilă cu alte V
- V pagină conține scheletul pe care se compun V parțiale pentru a forma pagina
- Recomandare: V – va conține doar cod pentru redare informații
V : codul HTML, fișierele CSS și fișierele Javascript

DETALIERE MODEL

Transfer date din “form” implică:

- *extragere câmpuri* din parametrii cererii HTTP
- *igienizarea datelor* (prevenire SQL injection și alte atacuri)
- *validare date*

Recomandări :

- plasare aceste operații pe M
- folosirea facilităților frameworkului de construire fraze SQL
- orice date de intrare de la un utilizator *remote* trebuie tratate ca risc și trebuie igienizată pentru a minimiza riscurile de atacuri intenționate și neintenționate.

CONCLUZII

Structurare aplicație a.î. să profite de infrastructura frameworkului și de facilitățile oferite de acesta

1. M, V, C
2. evitarea duplicărilor (DRY)
3. folosirea automatizării oferite de framework :

generatoare

- structuri de fișiere
- schelet conținut fișiere

mapări

{nume_câmp_formă ↔ atribut obiect model ↔ câmp tabel BD}

Evaluare formativă

1. Precizați din ce elemente ale modelului analiză al aplicației rezultă fiecare dintre componentele M, V și C.

<https://forms.gle/dd3qErTPfssyHSd4A>

PLAN CURS

Proiectarea aplicațiilor Web

- Principii
- Modelul proiect pentru aplicații Web
- MVC (Model-view-controller)
- MVC Frameworks
- Proiectare aplicații MVC
- **Concept-based URI**
- Ruby on Rails MVC Framework
- Spring Web MVC Framework

EVOLUȚIE ACCES ÎN WEB

1. Adresare directă fișiere statice

`https://mașina/cale/fisier_date`

Ex. https://www.info.uvt.ro/wp-content/uploads/2018/10/m_is_sem1_2018_2019_final.pdf.

2. Adresare directă cod executabil

`https://mașina/cale/fisier_cod`

Ex. https://www.w3schools.com/php/php_examples.asp

3. Adresare directă spațiu de concepte (și rutare)

`https://aplicația/nume_concept/instanta_concept #oper`

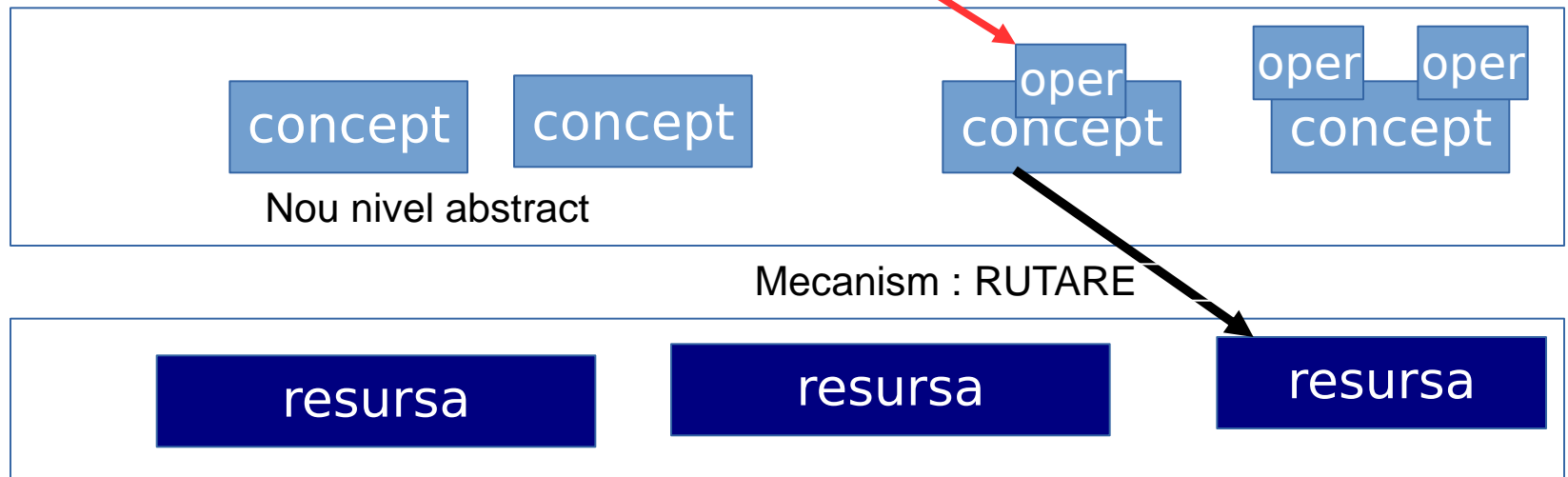
Ex. [#GET](https://www.RcApp.org/professors/Mindruta)

Ce prelucrare rezultă ? Dar pentru PUT, DELETE ?

Def. Concept-based URI – URI către concept și operație cu acesta.

Interfață directă către API = setul de concepte și operații cu concepte, oferite utilizatorilor.

- Aplicația - ierarhie de concepte / operații, ce au în background resurse
- Resursa = fișier cu informații / cod ce produce informații



PROIECTARE URI

URI - *identificator* al unui *concept* al aplicației, cu semnificație pentru utilizator
(nu ca înainte: al unei funcții care afișează acel concept și care avea semnificație pentru dezvoltator, sau al unui fișier fizic static)

Serviciul oferă un spațiu adresabil de **concepte organizate ierarhic și operații cu acestea** - nou nivel de abstractizare al aplicației

API = interfață (HTML, XML, RDF,...) la serviciu

Concepte organizate ierarhic și operații asupra lor

mecanism de
RUTARE

Fișiere cu info statice și cod ce produce informații (dinamice)

TRANSFORMARE APLICAȚIE ÎN SERVICIU

Creare **API** al aplicației.

Discuție.

URI (Uniform Resource Identifier) – referință la o capacitate virtuală a aplicației – la o **resursă**.

“A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.”

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2_1_1

* REST – Representational State Transfer (stil arhitectural pentru servicii web - Roy Fielding "Architectural Styles and the Design of Network-based Software Architectures,,)

Resursa poate produce:

- diferite reprezentări (cu date dinamice)
- diferite tipuri ale aceleiași reprezentări (HTML, XML, PDF, etc.)

DAR - API și prezentarea site-ului partajează același cod și diferă doar prin modul de redare a răspunsului.

Rezultă : Proiectare URI devine element important al procesului de proiectare.

PROIECTARE URI

URI – template cu *corespondențe în conceptele și operațiile* cu aceste concepte oferite de aplicație.

- conține identificare *controller, operație, parametri* (poate include și formatul datelor de ieșire).

Exemplu:

Template:

```
/exam/{an}/{luna}/{zi}/{fisier}.{format}
```

controller – exam

operația – view

parametrii - {an}, {luna}, {zi}, {fisier}, {format}

Instanță :

```
/exam/2017/iulie/11/IS.pdf
```

valori parametri - 2017, iulie, 11, IS, pdf

TRANSFORMARE APLICAȚIE ÎN SERVICIU – CREARE API

1. Definire URI

- API e definit de o *colecție de URI*
- API e legat de implementare prin *rutarea acestor URI către acțiuni (controller)*

2. Extindere cu noi formate de răspuns (XML, RSS, RDF) prin definire de view-uri specifice acestor formate.

3. Definire excepții în model pentru fiecare eroare. Acestea vor fi capturate de acțiuni și raportate utilizatorului în diferite formate.

4. Proiectare facilitate de măsurare a accesului:

- autentificare utilizator (pentru măsurare utilizare și, dacă e cazul, pentru protejare acces)
 - API key corelat cu o listă de IP-uri acceptabile
- stabilire număr limită de accese în unitatea de timp
- plasare în interceptoare (filtre) a codurilor de autentificare și de măsurare utilizate; răspuns cu anularea execuției la eroare autentificare sau depășirea limitei.

CREARE SERVICIU RESTful

Def. Resursă RESTful = noțiune abstractă identificată prin URI (într-un spațiu de nume global)

- URI al unei resurse este rezolvabil ca URL în Web
- Produce o reprezentare a conținutului curent, care poate fi transmisă în diferite formate (ex. HTML, XML, PDF, text, JSON, etc.)

Def. Serviciu RESTful

- definit în termeni de resurse RESTful
- oferă un set de operații de bază (CRUD / (POST, GET, PUT, DELETE)) pentru toate resursele
- comunică cu exteriorul în termeni de resurse și operații pe aceste resurse

OBS. Se poate construi o nouă aplicație combinând operații pe resurse din diferite servicii RESTful.

CREARE SERVICIU RESTful

Proiectare serviciu:

- colecție de resurse ce formează “modelul public” folosit să reprezinte funcționalitatea site-lui.
- câte un controller pe fiecare tip-de-resursă, pentru administrare operații CRUD.

Exemplu:

<http://www.myapp.org/courses/PSSW> va fi dirijat la CoursesController

<http://www.myapp.org/courses/PSSW/lectures/3> va fi dirijat la LecturesController

- serviciul = accesibil prin set de puncte de acces la resurse (*endpoints*)
- apelurile de metode din API = operații REST bazate pe HTTP

CREARE SERVICIU RESTful

Proiectare URI

URI-uri – intuitive și organizate în structuri arborescente

- ierarhie cu rădăcină unică
- implementarea unei resurse poate fi modificată fără implicații asupra URI

Format general URI

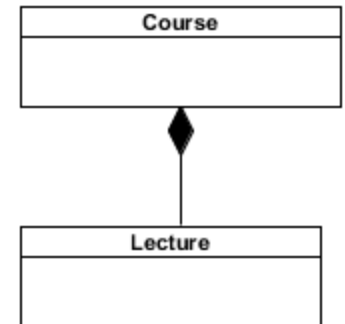
<protocol>://<nume_serviciu>/<tip_resursa><ID_resursa>

Exemplu:

- definire set structurat: <http://www.cm.org/courses/{course}>
- adresare resursă din set: <http://www.cm.org/courses/PSSw>

Resurse încuibate: relație de agregare/compoziție !

- adresare resursă : <http://www.cm.org/courses/PSSw/lectures/3>



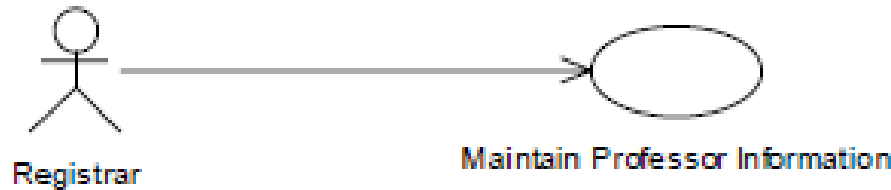
Operații non-CRUD pot fi refactorizate ca resurse pe care se pot aplica operații CRUD

Exemplu:

/accounts/55/close # POST	→	/accounts/55/account_closure # POST
/login # POST	→	/session # POST
/logout # POST	→	/session # DELETE

CREARE SERVICIU RESTful

Proiectare URI



Exemplu:

Tip resursă – **professors**

`www.RcApp.org/professors/{professor}`

create (creare profesor): `www.RcApp.org/professors` #POST

read (citire listă profesori): `www.RcApp.org/professors` #GET

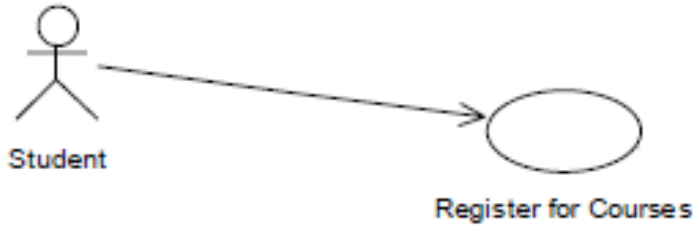
read (citire detalii profesor): `www.RcApp.org/professors/theProfessor` #GET

update (actualizare profesor): `www.RcApp.org/professors/theProfessor` #PUT

delete (ștergere profesor): `www.RcApp.org/professors/theProfessor` #DELETE

CREARE SERVICIU RESTful

Proiectare URI



Tip resursă ?

Exemplu:

Tip resursă - **schedules**

`www.RcApp.org/students/theStudent/schedules`

create (create schedule):

`www.RcApp.org/students/theStudent/schedules` #POST

read (citire lista schedule):

`www.RcApp.org/students/theStudent/schedules` #GET

read (citire detalii schedule):

`www.RcApp.org/students/theStudent /schedules/theSchedule` #GET

update (actualizare schedule):

`www.RcApp.org/students/theStudent /schedules/theSchedule` #PUT

delete (ștergere schedule):

`www.RcApp.org/students/theStudent/schedules/theSchedule` #DELETE

Exemplu : Detalii *schedule* din semestrul 4 al studentului *Grigore*

`www.RcApp.org/students/Grigore/ schedules/sem_4` #GET

Evaluare formativă

1. Fie www.psswcm.com/lectures/WebMVC_ URI-ul unei resurse oferită de o aplicație web prin RESTful API. Unde vor fi rutate solicitările cu acest URI ?

<https://forms.gle/E5G6WiizbsjNRrEN9>

PLAN CURS

Proiectarea aplicațiilor Web

- Principii
- Modelul proiect pentru aplicații Web
- MVC (Model-view-controller)
- MVC Frameworks
- Proiectare aplicații MVC
- Concept-based URI
- Ruby on Rails MVC Framework
- Spring Web MVC Framework

RUBY ON RAILS (RoR)

MVC Framework orientat pe limbajul Ruby

Interacțiunea cu RoR - prin scripturi shell care generează și administrează contextul proiectului (Rails).

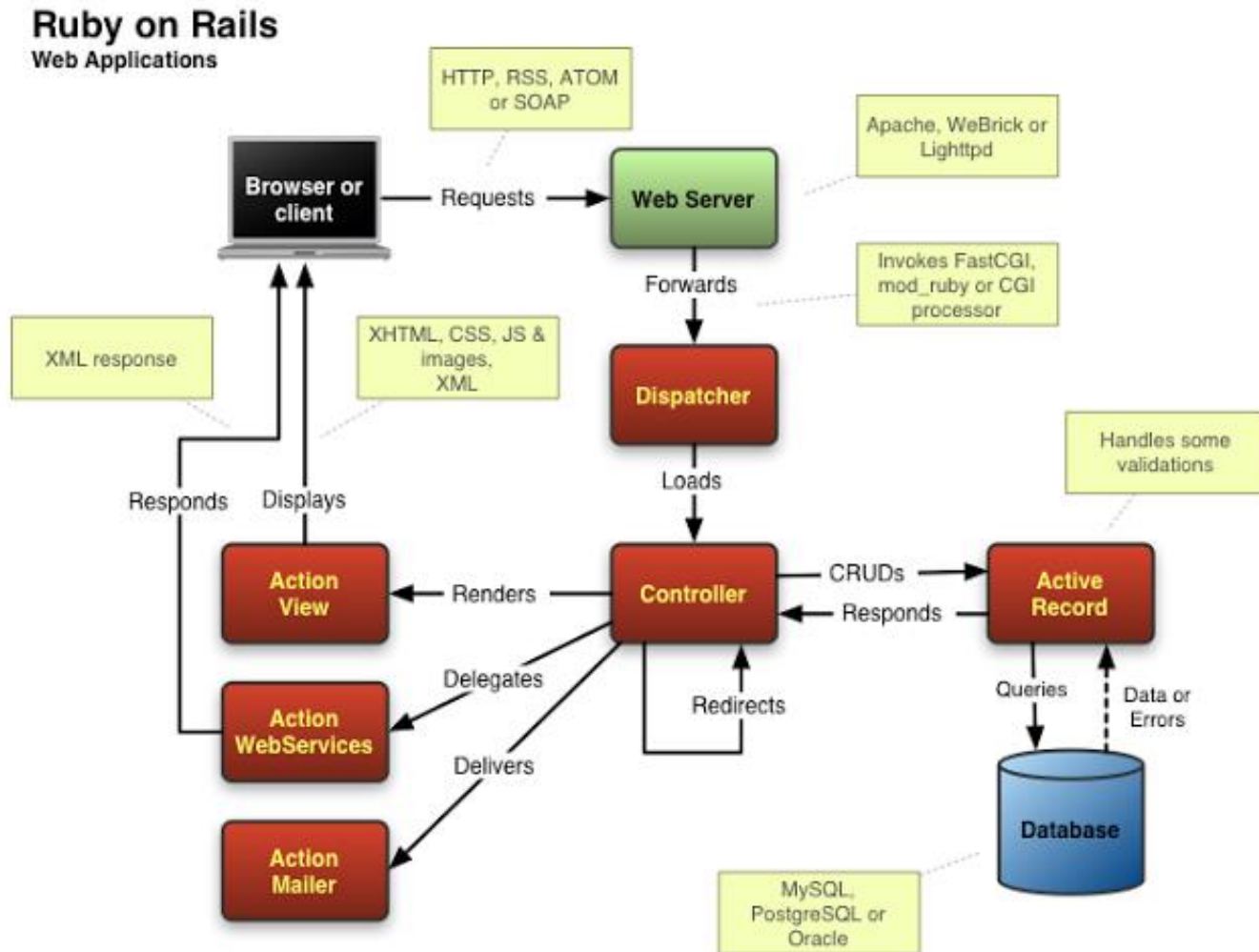
`/script` – folder ce conține

- un generator de cod
- *Stubs* de scheme BD
- server web bazat Ruby (folosit la testare pe măsură ce se dezvoltă cod)
- consolă pentru experimente și depanare

Alte taskuri – manipulate prin `Rake` (echivalent `Ant`)

- migrarea schemei BD
- testare automată
- instalare cod

RUBY ON RAILS (RoR)



RUBY ON RAILS (RoR)

Componente framework esențiale:

pentru **M**odel

- ActiveRecord
- ActiveResource
- Validations/ClassMethods

pentru **V**iew

- ActionView
- RJS* (Helper)

pentru **C**ontroller

- ActionController
- Filters
- Response Types / InstanceMethods pe MimeResponds
- Routing

* ruby-to-js : generează cod JS executat de browser ca răspuns la o cerere AJAX.

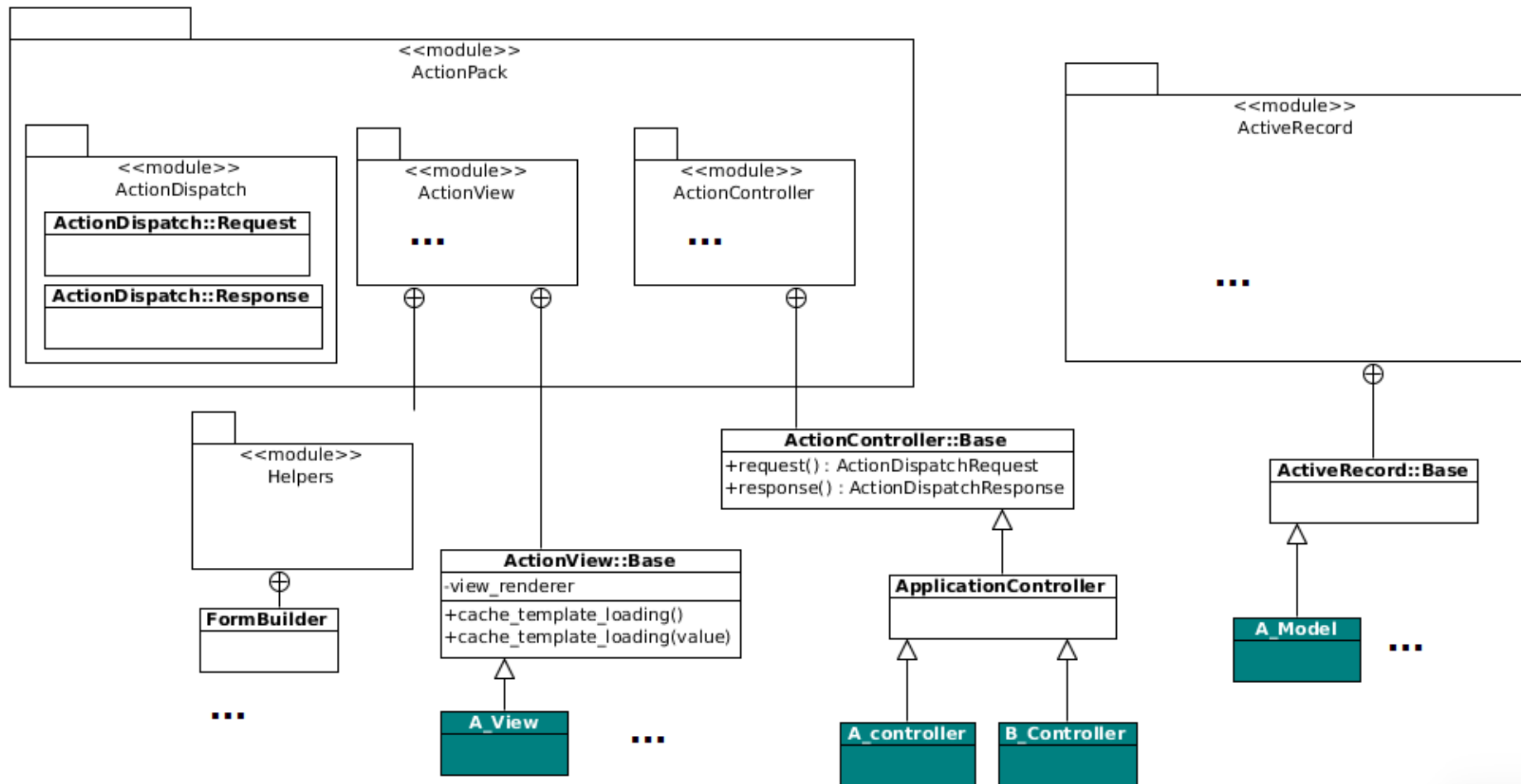
RUBY ON RAILS (RoR)- extras



Componentă framework



Componentă aplicație



PROIECTARE ȘI CREARE MODEL

Procedura:

1. Definire nume și attribute obiect model \Rightarrow *generare obiect model și schema de creare a tabeli în BD (migrarea)*.
2. Lista câmpurilor în schema migrării \Rightarrow *attribute și metode accesori în clasă*
3. Specificarea *asocierilor** cu alte modele, folosind modulul `ActiveRecordAssociations`
4. Specificarea *validărilor pentru instanțe*, ce trebuie trecute înainte de creare, salvare sau actualizare date din model, folosind modulul `ActiveRecordValidations`
5. Definirea celorlalte *metode* ale clasei

*Suportă asocieri polimorfice (asociere o tabelă cu diferite tabele și același tip de asociere – exemplu (User cu Photo, Post, etc., asociate cu Comentarii)

PROIECTARE MODEL

Metodele vor implementa *toate acțiunile relevante modelului* (controller-ul este responsabil doar să cunoască ce acțiune să invoce și când).

Modelele vor implementa *tratarea excepțiilor* (ex. înregistrări lipsă, tentativă de acces neautorizat la date, sesiune expirată, eroare de autentificare, validare date)

- detectarea problemei
- transferul erorii către controller, și de aici utilizatorului

Modelul este portabil.

Controller-ul este legat de o anumită aplicație.

Aceasta simplifică și testarea.

(De ce ?)

TRATAREA ERORILOR

Mecanism : transformare erori în date de ieșire.

- Folosește mesaje (Flash) de nivel *info*, *warning* sau *error* – un tip de *log* din care mesajele sunt trimise utilizatorului.
- Programatic - în blocul `catch` se pune mesajul în variabila `flash[:error]` și se redirectează utilizatorul către pagina care a generat cererea greșită.
- *Automat* - *Application layout* începe cu verificarea existenței acestor mesaje și inserează cod HTML pentru apariția unei casete cu mesajul respectiv.

Excepțiile definite de programator moștenesc din `RuntimeError`.

Programatorul poate defini ierarhii de excepții.

CREARE MODEL

Generare *schelet model* și generare *migrare* (script construire/modificare tabel în BD)

⇒ **clasa model** cu câmpurile indicate (inclusiv posibilă relație de moștenire folosind opțiunea `--parent`)

OBS. Relațiile și validările vor fi completate ulterior de programator

rails generate model {nume_model} {lista_câmp/tip/index} [optiuni]



automat invocare ActiveRecord și generare structura de fișiere cu conținut inițial:

`/app/models/ {nume_model}.rb`

(ex. **Clasa Grupa.rb** inițializată cu `class Grupa < ActiveRecord::Base end`)

`test/unit/ {nume_model}.rb`

`test/fixtures/ {nume_model}.yml`

Active Record – biblioteca ce manipulează obiectele modelului.

Creare campuri implicite:

ID (autoincrement)

`created_at`

`updated_at`

https://guides.rubyonrails.org/active_record_basics.html

CREARE MODEL

Exemplu de completare relații și validări:

```
class Grupa < ActiveRecord::Base
  ...
  has_many : Student
  attr_accessible: Grupa_nume //accesibilitate
  validates: Grupa_nume, presence: true //validare existență date
end
```

relație

validation helper

Rails maschează scrierea interogărilor SQL (ex. {nume_model}_**all**, {nume_model}.**new**, {nume_model}.**find**, {nume_model}.**find_by...**, {nume_model_item}) **save**, etc.)

PROIECTARE MODEL

ActiveResource

- alternativă orientată-resursă la `ActiveRecord`
- mapare obiect din model (M) pe punct de acces (*endpoint*) resursă RESTful (internă sau web) (i.e. sursa de date pentru M va fi resursa)
- Conectare obiecte business la servicii REST
- Wrapper ORM pentru servicii web
- Oferă capabilități proxy între un client și un serviciu RESTful

Mecanismul : din cererea pentru o resursă *remote* este generată o cerere REST XML, este transmisă și rezultatul recepționat este serializat și plasat într-un obiect Ruby utilizabil.

PROIECTARE ȘI CREARE VEDERI

Programatorul este responsabil să realizeze corespondența *variabilelor transferate* între View și Controller.

La redarea unei vederi *controller-ul parsează* un fișier `*.rhtml` (nu apelează o funcție) din care extrage codul Ruby pe care îl *execută*.

View poate fi reprezentat în diverse *formate* : HTML, XML, RDF, PDF, text (eventual trimis prin e-mail, poștă, etc).

`app/views/layouts/application.html.erb`

conține layout-ul general : codul HTML, fișierele CSS și Javascript comune tuturor vederilor

CSS în → `app/assets/stylesheets/`

cod javascript în → `app/assets/javascript/`

imagini în → `app/assets/images/`

PROIECTARE VEDERI

Partial = fișier reutilizabil, reprezentat în HTML; poate conține alte parțiale dar și structuri de control (ex. loop).

Bloc constructiv integrat în pagină prin referință.

Descompunere site în blocuri **partial** – etapă critică în procesul de proiectare a vederilor aplicației.

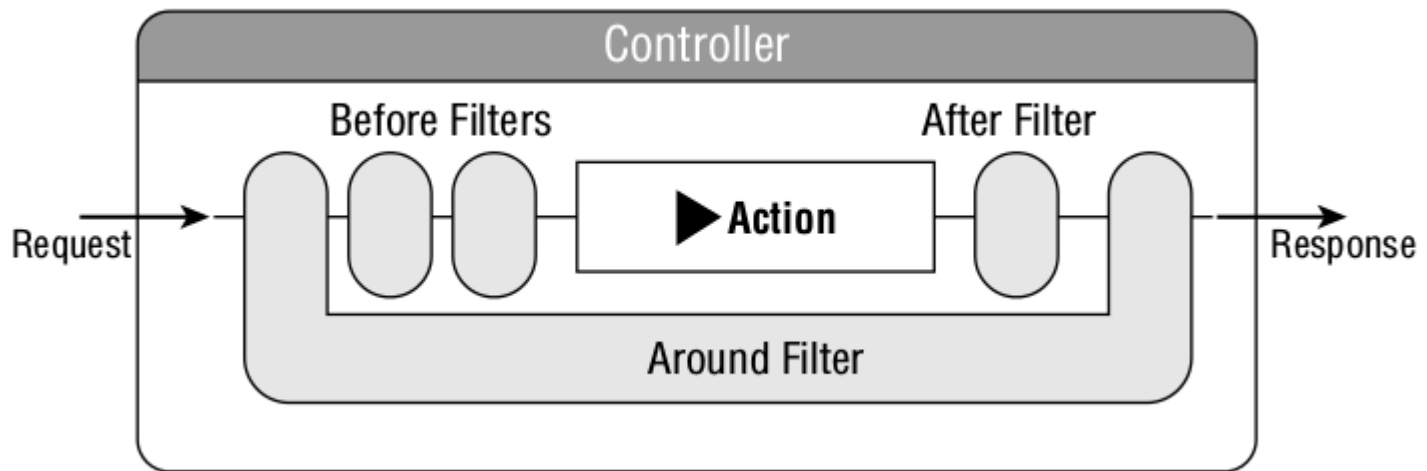
Partial tipice:

- Profil complet – informații complete despre un anumit obiect; devine de obicei o pagină dar poate fi înconjurat de mici controale (ex. sidebars, ads)
- Forma corespunzătoare unui obiect manipulat de utilizator
- Formele de căutare de bază și de căutare extinsă
- Rând rezumat – cu link la detalii obiect și cu acțiuni ce pot fi realizate asupra obiectului (delete, send, tag, etc) – obținut ca răspuns la comenzi de căutare
- Pictogramă rezumat – cu link la detalii

PROIECTARE CONTROLLER-e

Controller:

- *Orchestratorul* activității pe site
- *Nu* implementează *logică business*
- Reprezintă *punctele de acces* la site (endpoints)
- Răspunsul la cerere – combinare *acțiune* cu un *lanț de filtre*



PROIECTARE CONTROLLER

Controller – conține un set de **Action** executate la cerere

Action identificată prin *URI* și accesibilă pe o *rută prestabilită* și definită în `routes.rb`

Acces clasic la componentele cererii HTTP (inclusiv parametrii)

Arhetipuri de bază ale **Action**:

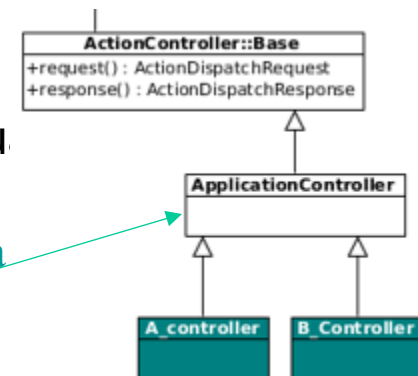
“get-and-show”

1. preluare obiecte din model în variabile instanță
2. redare template (implicit `app/views/numeController/numeAction.html.erb`) cu date din aceste variabile

“do-and-redirect”

1. operații asupra modelului
2. redirectare către altă acțiune (cerere browser către nou

Obs. `ApplicationController` – spațiu comun pentru protecție la cereri false și filtrarea parametrilor sensibili ai cererii, ș.a.



PROIECTARE CONTROLLER

Recomandări:

Plasare operații de *afișare* (`request.GET`) și *procesare* a unei *forme* (`request.POST`) în *aceeași Action*

Separare *afișare* de *modificare model* în *Action* diferite:

- *Action* pentru preluare date din model și afișare în formă
- *Action* pentru procesare date din formă și redirectare către o *Action* de afișare date din model

Capturarea erorilor de validare trimise de model și reafișarea formei cu indicații despre erori.

Set redus de operații corelate, altfel refactorizare prin extragere unele operații în alt controller.

PROIECTARE CONTROLLER

Filtre – implementate ca *metode pe controller* (sau pe un ancestor al acestuia)

Au acces la :

- obiectele `request` și `response` \Rightarrow pot realiza criptări, compresii, verificări erori
- datele instanță de la controller \Rightarrow pot examina și modifica realizarea operațiilor din `Action`

Returnează o valoare booleană \Rightarrow pot opri acțiunea, pot redirecta

Categorii de filtre:

- “before”
- “after” (dar înainte de trimitere răspuns la utilizator)
- “around” (pereche “before” și “after”)

CREARE CONTROLLER

Definire *nume controller* și *listă de acțiuni*.

Generatorul crează automat:

- Structură de fișiere și fișierele de tip
 - controller
 - view
 - test funcțional
 - helper pentru view
 - JavaScript
 - stylesheet
- Rutele pentru controller și acțiuni

CREARE CONTROLLER

```
rails generate controller {nume_controller} {lista_actiuni}
```

1. Generarea structurii de fişiere în /app

- controllers/{nume_controller}_controller.rb

cu metoda(e) empty în interior corespunzătoare pentru acţiunile din listă, care însă implicit extrag view-ul corespunzător (generat şi modificabil) şi îl afişează în browser.

- views/ {nume_controller} /
- views/ {nume_controller} / {nume_acţiune}.html.erb, ...
- helpers/ {nume_controller} _helper.rb
- test/functional/ {nume_controller} _controller_test.rb
- test/unit/helpers/ {nume_controller} _helper_test.rb
- assets/javascripts/ {nume_controller} _js_coffee
- assets/stylesheets/ {nume_controller} _css_scss

DETALIERE CONTROLLER

Adăugare *actions* și *views* la controller

- *action* – creare bloc `def {nume_actiune} ... end`
- *view* – creare
`/app/views/{nume_controller}/{nume_actiune}.html.erb`

Legare URL sau link (creat in prealabil) de *action*

Legare *form* de controller – prin variabila `params` cu perechi de tip *key-value* (nume camp – valoare, `nume_model/nume_camp` – valoare)

`redirect_to` redirectare către o *action*

`render` “redirectare” către un *view*

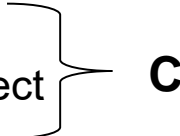
GENERARE SCHELET APLICAȚIE

```
generate scaffold {nume_resursă}
```

Generare schelet aplicație: model, *migration* pentru model, controller manipulare model, view manipulare și prezentare model, suită de teste pentru fiecare.

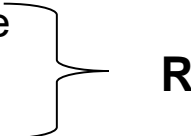
Funcțiile generate automat pentru obiecte model:

New – afișare formă pentru un obiect nou
create – preluare date și tentativă de creare obiect



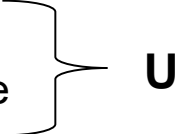
C

index – afișare listă cu toate obiectele
Show – afișare date obiect



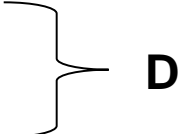
R

Edit – afișare formă editare populată cu starea curentă a obiectului
update – tentativă de a actualiza obiectul cu datele preluate din cerere



U

destroy – eliminare obiect



D

PROCEDURĂ TIPICĂ DE CREARE APLICAȚII - RoR

1. Creare structură directori (cda `rails`, sau odată cu generare)
2. Creare bază de date
3. Configurare aplicație cu locația BD și credențialele de acces
4. Creare modele (`ActiveRecord`) – obiectele business cu care vor lucra controller-ele
5. Generare scripturi de migrare pentru a simplifica crearea și întreținerea tabelor din BD
6. Scrierea codului din Controller(e)
7. Creare Views pentru prezentare date in GUI
8. Creare filtre pentru probleme la nivel de aplicație în `ApplicationController`
9. Aplicare filtre corespunzătoare la acțiuni

LUCRU CU EVENIMENTE

Variante:

- AJAX polling : sondare permanentă a serverului pentru a obține actualizările.
- Web Sockets : suport pentru mecanism subscribing/publishing de partea client.
- HTML5 SSE (Server-Sent Events) : paginile web subscriu la surse de evenimente de pe serverul web care transmite modificările.

Evaluare formativă

1. Ce trebuie să definească proiectantul și ce elemente se generează automat pentru crearea unui element de tip Model.
2. Cum se poate realiza, în controller, separarea afișării de modificarea modelului.
3. Care sunt tipurile de filtre ? La ce sunt utile filtrele ?

<https://forms.gle/mhdTYq2mDRUyGVsq7>

OBS. Întrebările se referă la framework-ul Ruby on Rails.

Bibliografie RoR

<https://guides.rubyonrails.org/>

<https://api.rubyonrails.org/>

Practical Object-Oriented Design in Ruby, 2012

Edward Benson, **The Art of Rails**, ed. Wiley Publishing, Inc.2008

Chad Pytel, Tammer Saleh, **Rails AntiPatterns**, ed. Addison-Wesley, 2011

https://en.wikibooks.org/wiki/Ruby_on_Rails

REZUMAT RoR

Ruby on Rails MVC Framework

- Arhitectura
- Componentele
- Clasele de bază
- Procedură pentru dezvoltare aplicație:
 - Creare structură directori și creare bază de date
 - Configurare aplicație cu locația BD și credențialele de acces
 - Creare modele (`ActiveRecord`) – obiectele business cu care vor lucra controller-ele
 - Generare scripturi de migrare – simplificarea creării și întreținerii tabelor din BD
 - Scrierea codului din Controller(e)
 - Creare Views pentru prezentare date in GUI
 - Creare filtre pentru probleme la nivel de aplicație în `ApplicationController`
 - Aplicare filtre corespunzătoare la acțiuni

PLAN CURS

Proiectarea aplicațiilor Web

- Principii
- Modelul proiect pentru aplicații Web
- MVC (Model-view-controller)
- MVC Frameworks
- Proiectare aplicații MVC
- Concept-based URI
- Ruby on Rails MVC Framework
- Spring Web MVC Framework

SPRING FRAMEWORK

Spring Framework – soluție de dezvoltare aplicații enterprise

- un model de componente și un set simplificat și consistent de APIs
- infrastructură modulară – permite selectarea modulelor necesare

Separare

- domeniul *problemei* – responsabilitatea *dezvoltatorului*
- probleme de *infrastructură* (persistență, tranzacții,...) – responsabilitatea *framework-ului*.

SPRING FRAMEWORK

Abordare bazată pe: POJO, injectare dependențe, template-uri, suport pentru biblioteci terțe, AOP (Aspect Oriented Programming), configurări bazate pe XML.

Structură:

- **Nucleu** : suport pentru dezvoltare baze de date, dezvoltare aplicații web, mapare ORM, tranzacții, servicii web RESTful.
- **Extensii** : securitate, web flow, servicii web SOAP, integrare enterprise, batch processing, mobile, rețele sociale, NoSQL, BlazeDS/Flex, AMQP/Rabbit, etc.

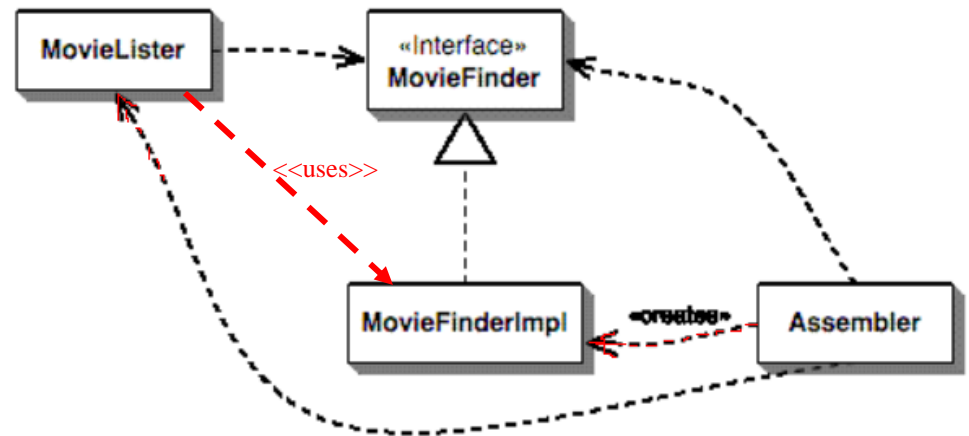
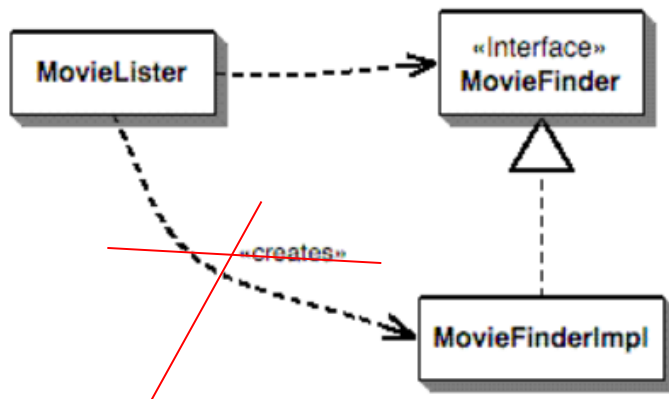
Spring Tool Suite – extensie Eclipse

Obs. Adaptare la contextul de piață : Spring Roo și Grails – (clone Rails) pentru Java și Groovy.

MECANISMUL DEPENDENCY INJECTION implementat în Spring Framework

Dependency injection* (un tip de IoC – Inversion of Control)

Responsabilitatea creării și localizării unui obiect este preluată de un obiect terț, eliminând dependența inițială pe care o preia.



Dependency Injection is a design pattern where the dependencies of a class are provided externally rather than created internally. In Java, DI is

commonly implemented using frameworks like Spring Framework, which provides support for dependency injection through features like @Autowired annotation, constructor injection, and setter injection.

* www.martinfowler.com/articles/injection.html

MECANISMUL DEPENDENCY INJECTION

Responsabilitatea creării și localizării unui obiect este preluată de framework.

CONTAINERUL IoC crează/localizează obiectul necesar și îl transferă (prin constructor, prin metodă setter, sau folosind suportul AOP) obiectului client care îl utilizează.

Consecință – asamblare aplicație la runtime, de către framework, fără participarea codului clientului ce solicită dependența.

Ciclul de viață al dependențelor este manipulat în exterior, de framework.



Obiectul client devine mai ușor de testat.

Obiectul client nu are cod specific de legare la o platformă particulară.

MECANISMUL DEPENDENCY INJECTION

Consecință – proiectare aplicații în manieră OO fără implicarea framework-ului în logica business.

- folosire POJO, fără restricții din partea platformei
 - independență față de orice framework sau context
 - concentrare pe logica business și pe principii de proiectare OO solide
1. construirea modelului domeniului
 2. extins de către framework și expus ca aplicație web.

SPRING FRAMEWORK

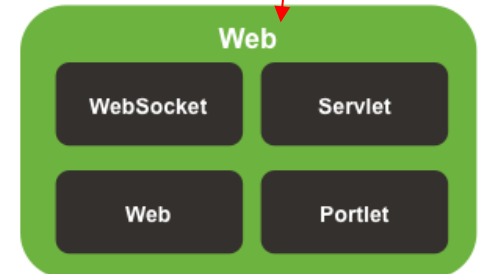
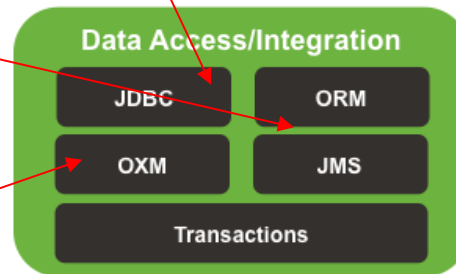
Administrare automată conexiuni, mapare date extrase din BD pe liste de obiecte de domeniu, executare proceduri stocate, suport pentru tranzacții

Framework MVC propriu și integrare cu alte framework-uri și tehnologii (Struts, JSF, Velocity, JSP, FreeMarker)



Spring Framework Runtime

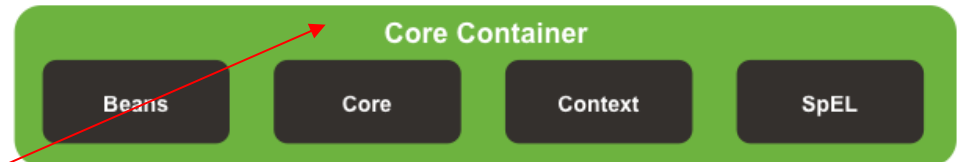
Suportă Hibernate, MyBatis, JDO, JPA



Suportă Castor, JAXB, JiBX, XMLBeans, XStream



Încapsulare probleme transversale sub formă de aspecte (ex. securitate, jurnalizare, management tranzacții, etc)



Facilități de deculpare de codul aplicației a creării, configurării și administrării beans



Suport pentru utilizare JUnit și TestNG

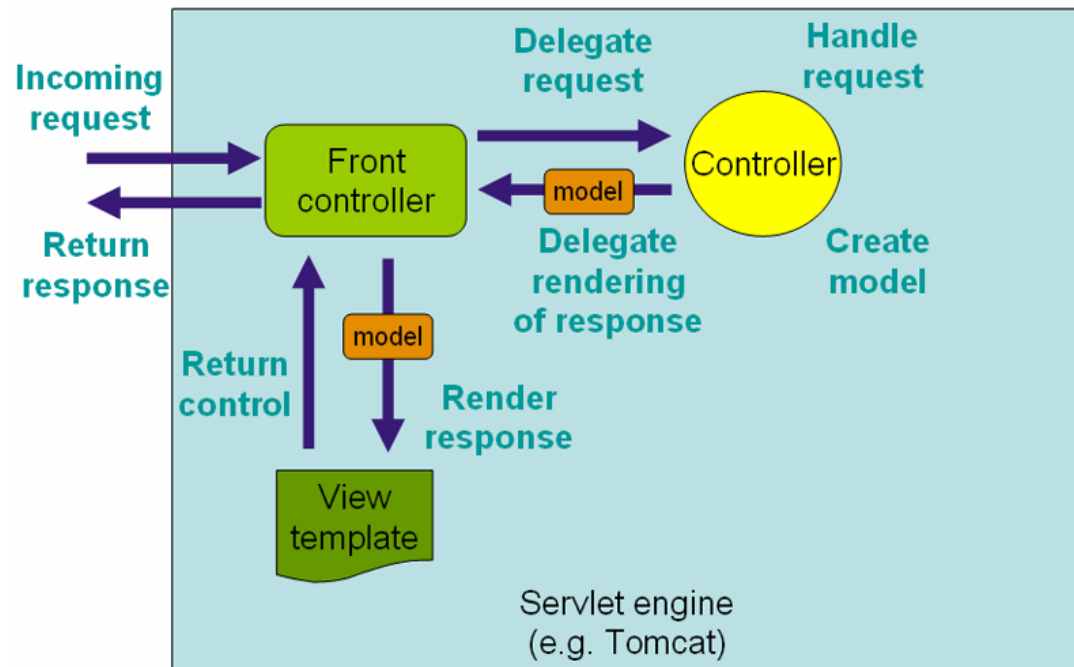
SPRING WEB MVC FRAMEWORK

Construit peste Servlet API

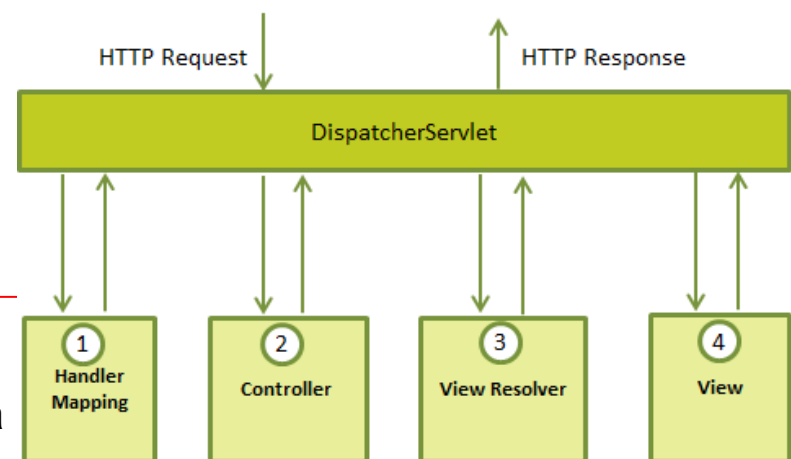
Front controller:

`DispatcherServlet` -servlet central ce dispeceerizează cererile către controller-e

Complet integrat cu container Spring IoC \Rightarrow utilizarea celorlalte facilități ale Spring



SPRING WEB MVC FRAMEWORK



Servlet-ul dispecer:

- determină, din URI, acțiunea ce trebuie invocată
- instanțiază clasa controller corespunzătoare
- populează bean cu valorile parametrilor din cerere
- apelează metoda corespunzătoare din obiectul controller
- transferă controlul către view-ul corespunzător

Configurare bazată pe XML, modificată prin editare fără a recompila aplicația.

Legare automată a datelor introduse de utilizator *la tipul corect*, prin parsarea șirului de intrare și extragerea valorilor proprietăților prin conversie la tipul acestora.

Aplicare *procedură de validare* și redirectare la forma de intrare în caz de eroare; validarea este opțională și se poate defini programatic, declarativ sau se pot folosi validatori built-in.

Suportă *internaționalizare* și *localizare*.

Suportă mai multe *tehnologii de prezentare*.

SPRING WEB MVC FRAMEWORK

DispatcherServlet – trimite cererile la handler-e

- configurări : mapări handler-e, rezoluții view și theme (look&feel, skin), locale, timezone.
- suport pentru încărcare (uploading) fișiere.

WebApplicationContext

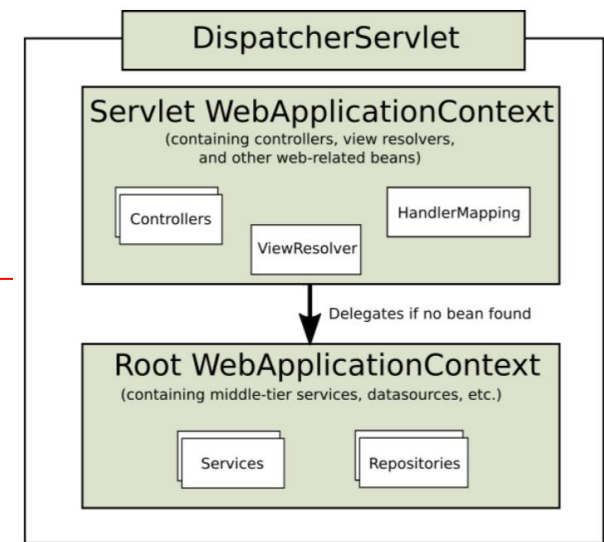
extends BeanFactory

- registru de obiecte și punct de integrare
- creare *beans*, cablare dependențe, oferire facilitate de căutare *beans*, administrează ciclul de viață al *beans*, include mecanism de rutare evenimente pentru producători și consumatori slab cuplați, etc.
- configurat (inclusiv dependențele) prin fișier XML ⇒ configurație aplicație.

Root WebApplicationContext

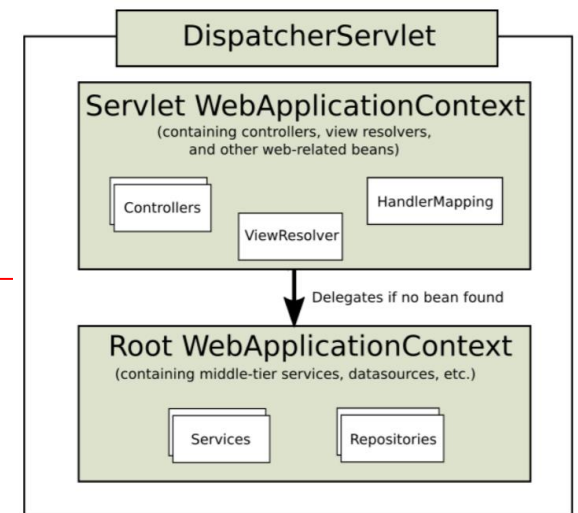
- Conține componente de infrastructură partajate de instanțe de Servlet
- Moștenite (pot fi suprascrise) în WebApplicationContext specific

Default handler – bazat pe adnotările `@Controller` și `@RequestMapping` plus - `@PathVariable` permite crearea de site-uri web și aplicații RESTful.

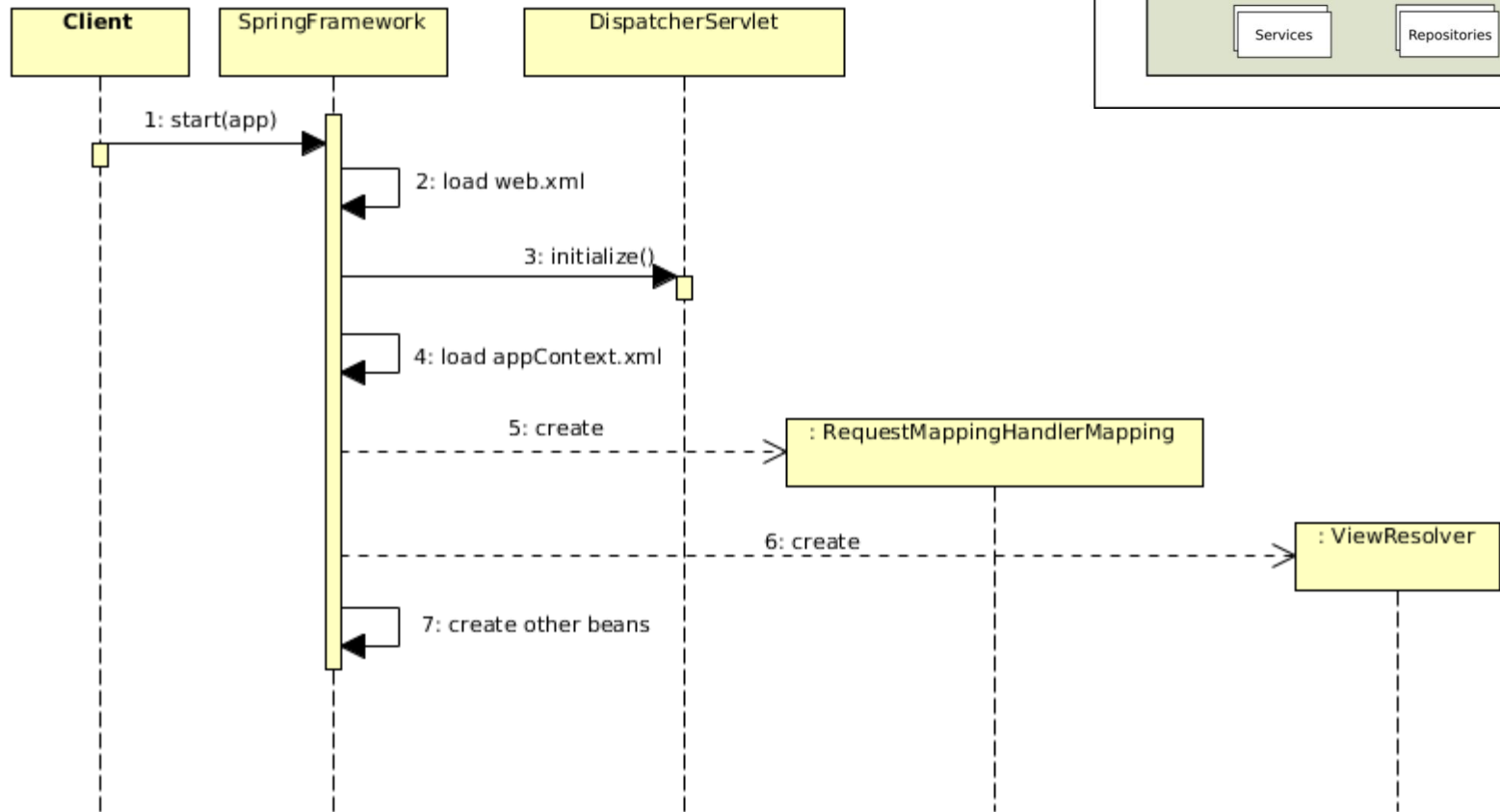


<https://docs.spring.io/spring-framework/docs/1.1.x/reference/mvc.html>

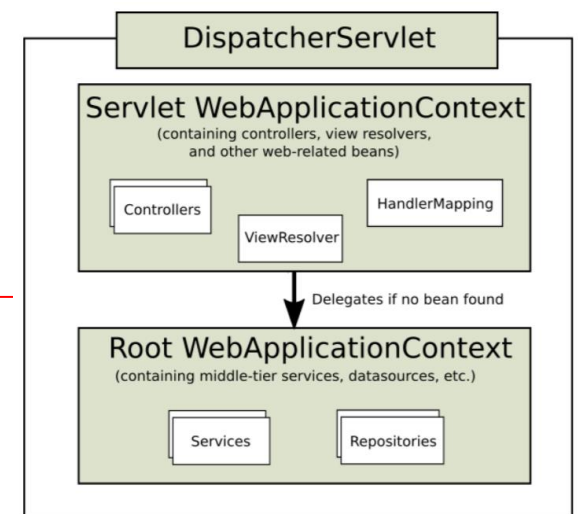
SPRING WEB MVC FRAMEWORK



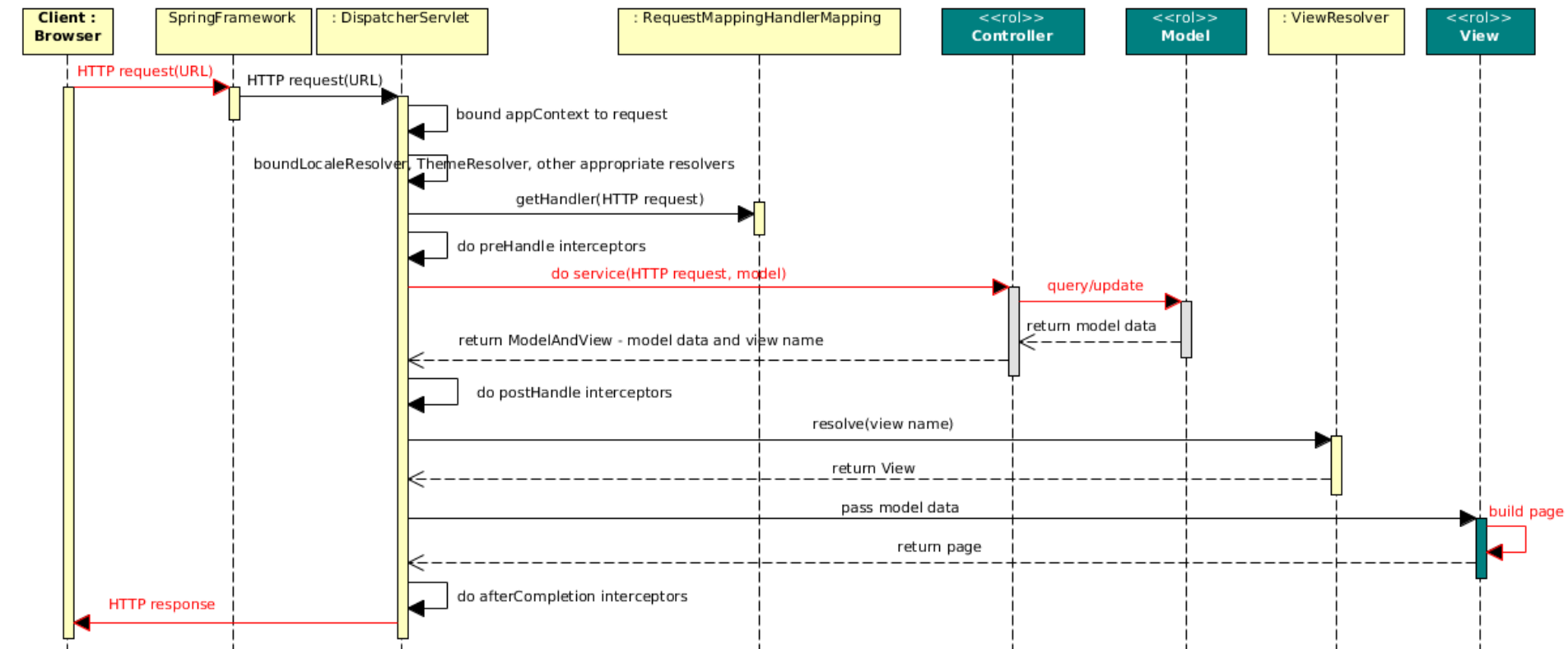
sd Spring Initialize application



SPRING WEB MVC FRAMEWORK



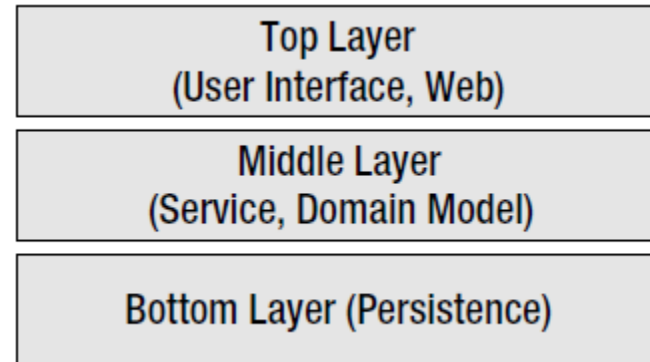
sd Spring Handle HTTP request



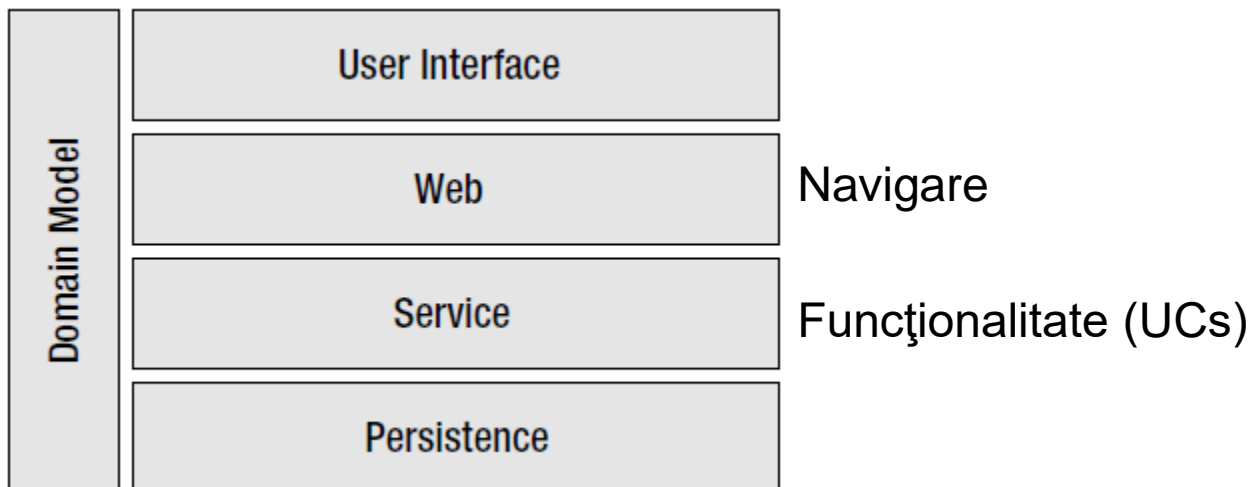
SPRING WEB MVC FRAMEWORK

ARHITECTURA APLICAȚIEI : PERSPECTIVA STATICĂ

Arhitectură generală aplicație web



Arhitectură aplicație Spring Web MVC



SPRING WEB MVC FRAMEWORK

COMPONENTE APLICAȚIE

Model – vehicol de comunicare între controller și vederi

- colecție de obiecte identificate prin nume, majoritatea instanțe ale modelului domeniului
- încapsulează datele și logica aplicației, în general în POJO*

View – redarea datelor din model, în general HTML interpretat de browsere

- poate folosi una din multiple tehnologii de redare (XML, JSON, PDF, etc)
- vederile sunt construite server-side
- legatura la model prin $\$(numeAtribut)$

Controller – procesarea cererii utilizator, construirea reprezentării curente a modelului și trimiterea lui spre redare la un View

- primește model ca parametru
- actualizează datele din model
- returnează numele view

*Plain Old Java Objects – obiecte Java simple (nu urmează modele, convenții sau frameworks)

SPRING WEB MVC FRAMEWORK

COMPONENTE APLICAȚIE

Locul componentelor **MVC** în structura de fișiere a unei aplicații:

SpringApp

- src
 - springapp.web
 - POJOs (**model**)
 - Controller-ele (**controller**)
- war
 - WEB-INF
 - lib
 - web.xml (mapare URL → controller)
 - springapp-servlet.xml (WebApplicationContext)
 - pages
 - views(**view**)

Obs. În plus față de **M**, **V** și **C** există *fișierele de configurare, resolverii*, etc. (unele implicite, dar modificabile).

SPRING WEB MVC FRAMEWORK

ADNOTĂRI

Model de programare bazat pe adnotări*.

Exemple:

`@Controller` – clasă controller (specializare pentru `@Component`)

`@ModelAttribute` – legare la attribute din model

`@RequestMapping` – legare metodă la URI

`@PathVariable` – extragere variabile din URI

(ex. <http://www.cm.org/courses/{course}/lectures/{lecture}>)

`@RequestParam` – legare parametri din cerere la parametrii metodei

`@RequestBody` / `@ResponseBody` – legare body cerere/răspuns la un parametru al metodei,

`@RequestHeader` – extrage informații din antetul cererii

`@SessionAttributes`

`@CookieValue`, ...

* Annotations provide metadata about a program that can be used by the compiler or at runtime

SPRING WEB MVC FRAMEWORK

ADNOTĂRI

Alte adnotări:

`@PostConstruct` – metodă apelată imediat după inițializare bean

`@PreDestroy` – metodă apelată exact înainte de distrugere bean

`@Required` – legare la attribute din model

`@Bean` – legare metodă la un obiect bean pe care îl returnează și care trebuie gestionat de Spring IoC container

`@Component` – componentă generală

`@Service` – specializare pentru `@Component`, de tip serviciu

`@Repository` – specializare pentru `@Component`, de tip Data Access Object (DAO)

Diferite servicii sunt solicitate declarativ si aplicate automat.

Exemple:

`@Autowired` – obiecte obținute prin dependency injection, pe bază de tip

`@Resource` – obiecte obținute prin dependency injection, pe bază de nume

`@Transactional` – tranzacții atomice

SPRING WEB MVC FRAMEWORK SERVICII

Data binding – leagă intrările de la utilizator cu modelul domeniului.

- Utilizează parametrii (de tip String) din cererea HTTP pentru a popula proprietățile, de diferite tipuri, ale obiectelor.
- Repopulează forma HTML dacă validarea intrărilor eșuează
- Utilizează o bibliotecă specializată “form tag library”

SPRING WEB MVC FRAMEWORK SERVICII

Există conversii implicite bazate pe nume, altfel:

Converter – componentă generică de conversie tipuri, utilizabilă pe orice treaptă (*tier*) a aplicației.

Formatter – componentă generică de conversie String în alt tip Java, utilizabilă doar pe treapta web (*web tier*).

Validator – lucrează la nivel de obiect, pentru toate câmpurile acestuia, după ce s-au executat conversiile corespunzătoare.

JSR 303 – specificație formală – set de API pentru a aplica constrângeri pe proprietățile obiectelor prin adnotări

Hibernate Validator – o implementare JSR 303

SPRING WEB MVC FRAMEWORK

INTERCEPTORI

Implementează interfața `HandlerInterceptor`.

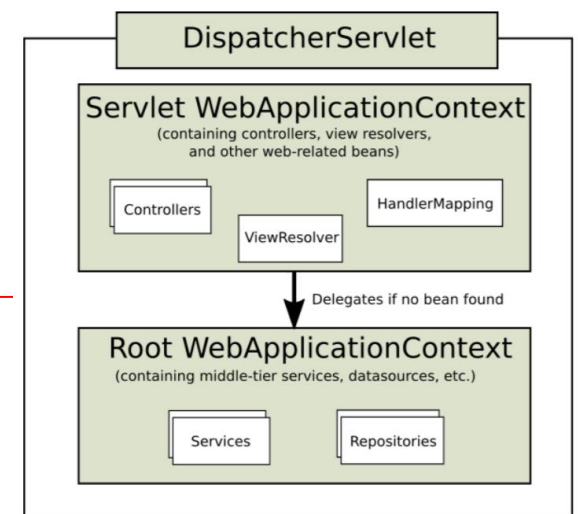
`preHandle()` – se execută înainte de lansarea operațiilor din *handler*, permite anularea execuției handlerului dacă întoarce *false*.

`postHandle()` – permite manipularea obiectului `ModelAndView` înainte de redarea lui în view.

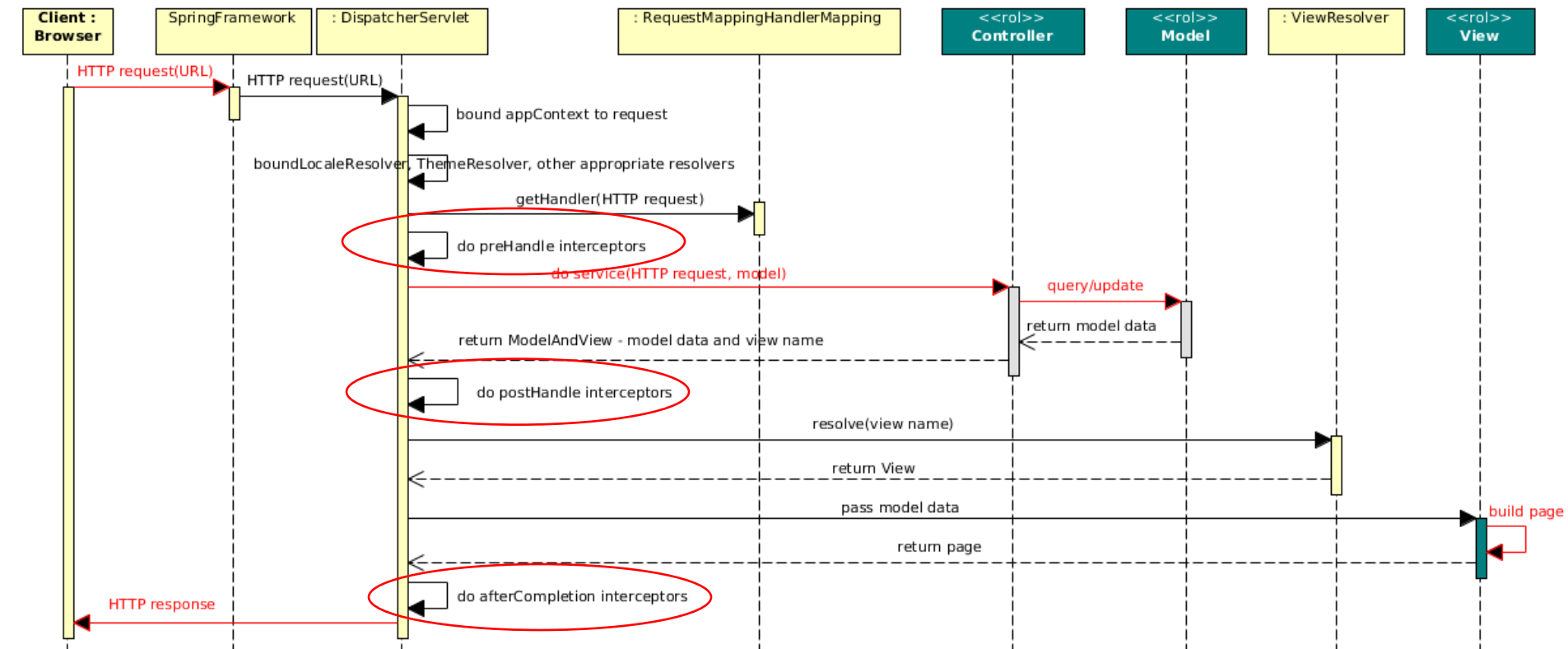
`afterCompletion()` – se execută la finalizarea cererii, dar înainte de trimiterea răspunsului.

Utilitate: autentificare, validare securitate, jurnalizare, etc.

SPRING WEB MVC FRAMEWORK



sd Spring Handle HTTP request



SPRING WEB MVC FRAMEWORK

PROIECTARE NAVIGARE

Proiectare reguli și fluxuri de navigare.

- Aplicații ce au un *model conversațional* al procesului business, nu un model pur cerere/răspuns.
- Construire fluxuri logice de pagini (module reutilizabile în diferite situații) – module ce ghidează utilizatorul prin navigări controlate care dirijează procesul business.

Spring Web Flow (SWF)

- Definirea regulilor de navigare și administrarea fluxului de pagini într-o aplicație web.
- Folosire limbaj declarativ pentru definire fluxuri pe nivel înalt de abstractizare.

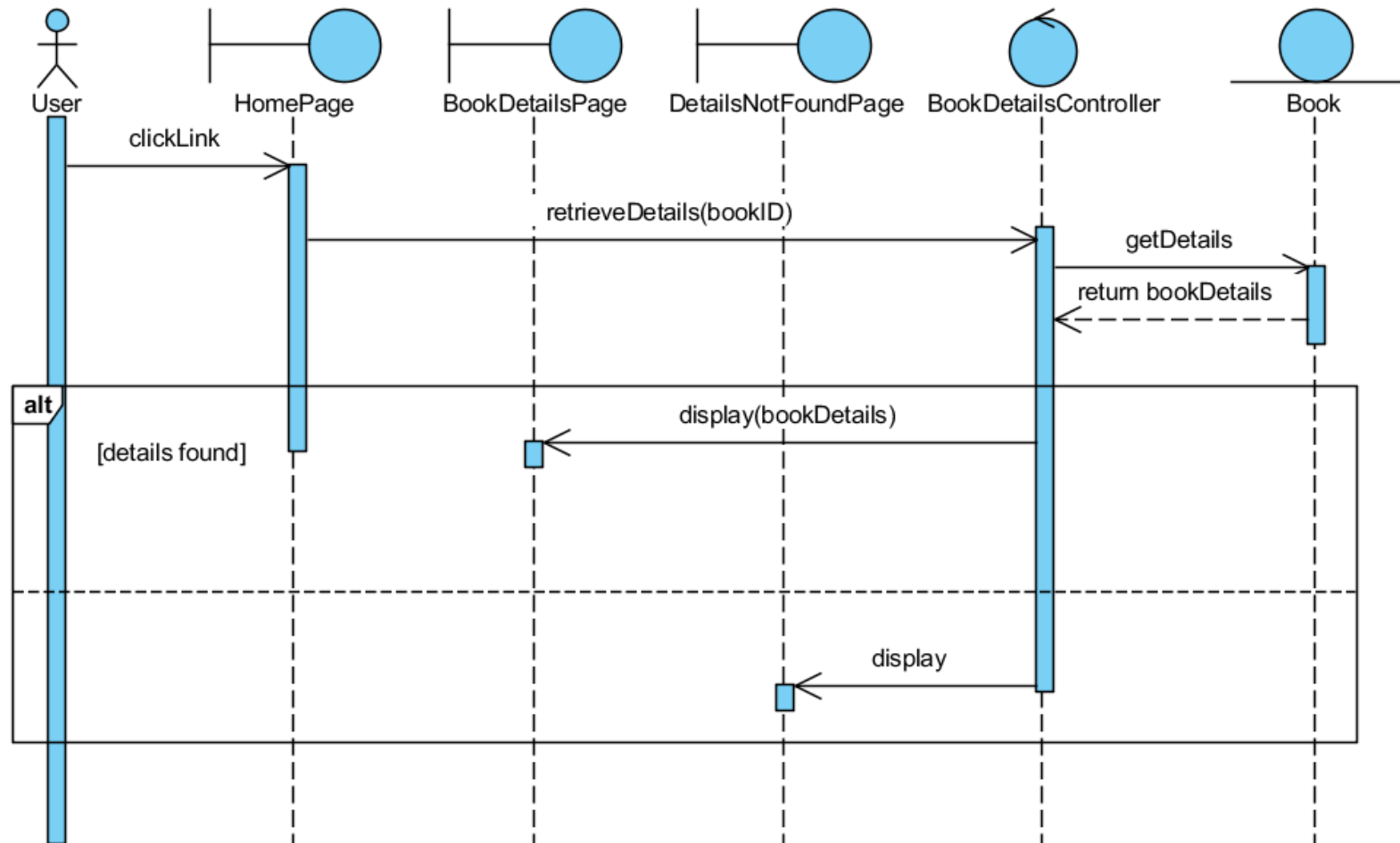
SPRING WEB MVC FRAMEWORK

Exemplu de procedură de dezvoltare aplicație:

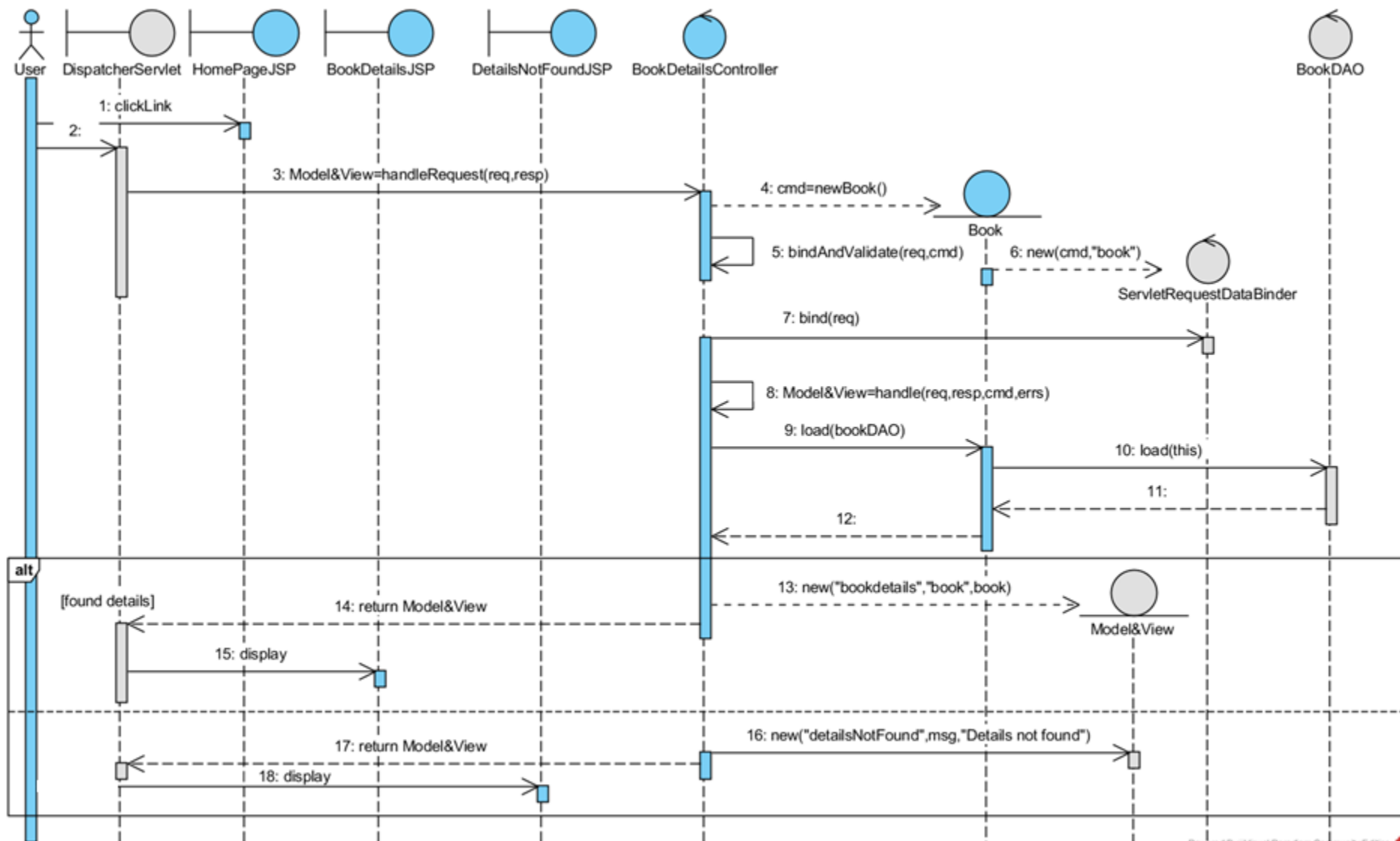
- creare Front Controller – declarat în `web.xml`
- creare Application Context – declarat în `web.xml`
- creare componente – declarate în `spring-mvc.xml`
 - creare controller-e
 - creare model
 - creare vederi
- creare URI pentru servicii RESTful și legare la controller-e

Exemplu – MODEL INDEPENDENT DE PLATFORMĂ

sd ShowBookDetailsUC - PIM



Exemplu – MODEL DEPENDENT DE PLATFORMA SPRING



SPRING WEB MVC FRAMEWORK ORM

DAO / Data Mapping

Spring oferă :

- Implementare suport DAO
- Strategii pentru lucrul cu tranzacții
- Integrare cu Hibernate, JDO, Oracle TopLink, iBATIS SQL Maps and JPA
 - DAO: pot fi configurate prin *dependency injection* și participă la managementul resurselor și tranzacțiilor implementat în Spring.

<https://docs.spring.io/spring-framework/docs/2.5.5/reference/orm.html>

Evaluare formativă

1. Ce rol are aplicarea mecanismului *dependency injection* ?
2. Care sunt operațiile interfeței `HandlerInterceptor` ? Ce credeți că se interceptează ?

<https://forms.gle/yVBJDTHCWQPRaAsh9>

OBS. Întrebările se referă la framework-ul Spring Web MVC.

Bibliografie Spring

Spring MVC

<https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/spring-web.html>

Paul Deck, **Spring MVC – A Tutorial**, second ed., Brainy Software Corp., 2016

<https://github.com/gigamailer/simplenin/blob/master/Spring%20MVC%2C%20A%20Tutorial%2C%20second%20edition%20-%20Paul%20Deck.pdf>

Spring Framework

www.pivotal.io/products/spring

Spring Framework Reference Documentation

<https://docs.spring.io/spring/docs/current/spring-framework-reference/>

Spring Framework API

<http://docs.spring.io/spring-framework/docs/current/javadoc-api/overview-summary.html>

Alte framework-uri MVC:

ASP.NET MVC

CodeIgniter

Django

AngularJs

Apache Struts

Grails

PureMVC

...

REZUMAT

Spring Web MVC Framework

- Dependency injection și IoC
- Arhitectura
- Mecanisme inițializare aplicație și manipulare cerere HTTP
- Componente aplicație (M,V,C) și structura de fișiere
- Model de programare bazat pe adnotări
- Servicii
- Interceptori
- Procedură pentru dezvoltare aplicație:
 - creare Front Controller – declarat în `web.xml`
 - creare Application Context – declarat în `web.xml`
 - creare componente – declarate în `spring-mvc.xml`
 - creare controller-e
 - creare model
 - creare vederi
 - creare URI pentru servicii RESTful și legare la controller-e