Disseny i desenvolupament d'un *framework* MVC en PHP

Memòria de Projecte Final de Carrera

Enginyeria Informàtica (2n cicle)

Autor: Josep Humet i Alsius

Consultor: Ignasi Lorente Puchades

Professor: Daniel Riera Terrén

13 de juny de 2016

Copyright

Dedicatòria

Aquest treball el dedico als meus pares Mercè i Agustí, per donar-me totes les oportunitats que m'han donat i per confiar sempre en mi. També als meus germans Joan, Jordi i Gemma, per fer de germans grans i ensenyar-me lliçons importants de la vida encara ara, i des de sempre.

Al Xavier Casahuga, pels seus coneixements sobre la matèria i per encomanar-me les seves ganes de fer les coses ben fetes.

De forma molt especial, vull dedicar aquest treball a l'Anna, per ser-hi sempre i ajudar-me a trobar el costat positiu de les coses, per acompanyar-me en aquest llarg camí i aconseguir que no tirés mai la tovallola, ni als moments més desesperants.

Abstract

El projecte que tot seguit es presenta és una solució al problema que suposa la creació d'un *framework* en PHP seguint el patró de disseny MVC. Això vol dir que l'objecte d'estudi és la programació web i l'objectiu principal que s'ha abordat ha estat aconseguir un conjunt de llibreries i fragments de codi per tal d'establir unes bases per a la programació de portals web en llenguatge PHP seguint el patró de disseny Model-Vista-Controlador.

La idea inicial era la creació del *framework* amb propòsits merament acadèmics, ja que no és un problema trivial, i es pensava que els resultats obtinguts no serien comparables als *frameworks* existents. Una vegada acabat el projecte, però, s'ha vist clarament que hi havia camí per fer però la base és prou sòlida com per seguir fent créixer el projecte i proposar-lo com a una alternativa més del mercat.

Paraules clau: *framework*, MVC, PHP, PFC, programació, enginyeria del software, A&D (arquitectura i disseny).

Abstract (English version)

The following pages contain an approach focused on the problem to develop a PHP *framework* based on Model View Controller design pattern. That means the main goal to reach is to create a core and code fragments to establish the coding bases to develop web applications using PHP and facing Model View Controller design pattern.

The original idea was to create a *framework* only as an academic study but without future perspectives, because it's not that easy to create something which it's practical and useful enough. There are a lot of existing *frameworks* which are incredibly powerful and the thought was that it would be impossible to build a *framework* good enough if comparing with any other that already exists. But once it's done (or at least, the first version), there still are a long way to go but the base it's rather strong to allow the project growth and maybe, in a few years, it can be considered a real option when choosing a *framework*, and can compete with the other alternatives.

Keywords: *framework*, MVC, PHP, PFC, programming, software engineering, A&D (Analysis and Design)

Índex

Figures i taules

Índex de figures

Índex de taules

1. Prefaci

La programació web està a l'ordre del dia, i hi ha moltes eines que permeten programar de forma molt més fàcil que no pas començant de zero amb cada projecte. Així doncs, la idea d'aquest projecte és la mateixa, fer una proposta de conjunt de llibreries amb la intenció de competir amb les solucions ja existents d'aquest mateix problema.

El projecte que aquest document detalla consta del disseny i desenvolupament d'un *framework* seguint el patró de disseny MVC (Model-Vista-Controlador), en llenguatge PHP.

En termes generals, entenem per *framework* un conjunt estandarditzat de conceptes, pràctiques i criteris que donen solució a un tipus particular de problema per establir-lo com a referència per a resoldre problemes similars. I més concretament, en el cas que ens ocupa, un *framework* defineix una estructura conceptual de suport que actuarà com a base per l'organització i desenvolupament de software. Així doncs, aquesta base constarà de mòduls i artefactes concrets que donaran solució a problemes parcials, per tal de brindar un conjunt d'eines, llibreries i components al programador i així aquest pugui desenvolupar de forma còmode, estructurada i eficient. Partim de la base que el patró de disseny MVC és un dels més emprats en programació web degut a la gran quantitat de *frameworks* que ja l'implementen i també a què és, possiblement, la forma més lògica de separar el codi d'una aplicació per funcionalitat del mateix (lògica, dades i interfície).

Després de provar diversos *frameworks* com Symfony, CodeIgniter, Slim o altres (per conèixer-ne els avantatges i inconvenients) s'ha procurat agafar les millors idees de cada un d'aquests *frameworks* ja existents per tal de construir-ne un de nou, que incorporés el mínim necessari per ser útil i lleuger a la vegada. Per tant s'ha prioritzat en tot moment lleugeresa i funcionalitat, sempre mirant de no sacrificar cap mòdul important.

Veiem doncs, que hi ha solucions existents per a tots els gustos, però la que aquest treball proposa vol fer una simplificació d'aquestes eines per tal de tenir un marc de treball senzill, intuïtiu i lleuger per a poder fer el desenvolupament d'aplicacions web de forma molt ràpida. I a més, s'ha volgut donar la possibilitat en tot moment de poder extendre el codi que conforme la base ja existent del *framework* per incorporar noves funcionalitats desitjades pel programador; així com també és possible, mitjançant el mateix sistema, modificar el flux d'execució. Per tant aquest *framework* constitueix una solució amb un alt grau de cohesió i un baix grau d'acoblament.

He cregut oportú prioritzar aquestes dues màximes en el disseny del *framework* perquè, basant-me en la meva experiència com a programador web, els *frameworks* amb què he programat sempre pecaven del mateix, que és precisament no prioritzar en com es programarà una aplicació sinó en quantes coses pot requerir el programador mentre estigui desenvolupant qualsevol cosa amb el seu *framework*. Això fa que l'enfoc i el plantejament dels objectius canviï lleugerament ja que en comptes d'intentar posar-li la vida fàcil al programador el què han volgut emfatitzar ha estat la quantitat de utilitats que li donaran. En canvi en aquest treball s'ha intentat prioritzar el com es programarà, i per compensar les poques eines que incorpora de moment, en comparació amb altres *frameworks*, s'ha donat la possibilitat d'extendre qualsevol dels components bàsics que componen el *framework*.

2. Objectius

2.1 Principals

L'objectiu del treball és crear un *framework* en PHP partint de zero. El *framework* en qüestió serà una implementació del patró de disseny MVC (Model-Vista-Controlador) i incorporarà algunes de les utilitats habituals que solen incorporar aquest tipus d'eines de desenvolupament., com podria ser un mòdul per a la gestió de base de dades, la gestió de vistes, etc.

Aquest objectiu global, doncs, el podem desglossar en els següents tres pilars:

1. Com a projecte d'enginyeria del software s'entén que el més important serà obtenir tot un conjunt de diagrames UML que descriguin el disseny del software.
2. Per altra banda, un cop fet el disseny, serà necessari programar tot el què es dissenyi ja que aquest és el producte final que es vol obtenir.
3. Programar una aplicació web d'exemple per validar el funcionament del *framework*.

2.2 Secundaris

Per tal d'assolir els objectius globals, tot seguit es detallen un conjunt d'objectius parcials que constitueixen les fites a curt termini necessàries per a donar cobertura a una solució genèrica del problema plantejat, el disseny d'un *framework*.

1. Estudiar i entendre els diversos *frameworks* existents al mercat. Cal conèixer quines funcionalitats incorpora cada un d'ells. Per a realitzar aquest anàlisis es farà un estudi centrant el punt de mira en els *frameworks* Symfony i CodeIgniter, ja que són dos dels més utilitzats actualment.
2. Definir quines seran les funcionalitats que incorporarà el *framework*. És important seleccionar quines són les prioritats, i per això serà important fer l'anàlisi d'altres solucions existents, per veure què incorpora cadascuna i tenir diverses visions de com afrontar un problema.
3. Estudiar els patrons de disseny que caldrà implementar. Un cop decidides les funcionalitats que es voldran implementar caldrà decidir com fer-ho, i això implica decidir quins patrons de disseny es poden utilitzar per a la implementació.
4. Definir jocs de proves per validar cadascuna de les funcionalitats per separat, així com de la solució global. Donat que hi haurà diversos mòduls al *framework*, és important decidir quins jocs de proves s'han de definir per poder validar el correcte funcionament de cadascuna de les parts del *framework*.
5. Fer un *benchmarking*[1] per comparar el rendiment del *framework* creat en relació amb la resta d'opcions estudiades.

[1] Comparativa respecte els estàndards marcats per la indústria.

3. Marc teòric

Davant la gran quantitat de *frameworks* existents a data d'avui per desenvolupar pàgines i aplicacions web en PHP, podria semblar absurd crear-ne un de nou. Però també és cert, que precisament aquest ímpetu per treure nous productes és el què fa que cada vegada hi hagi eines més completes, que deixen obsoletes les anteriors molt ràpidament.

La motivació a emprendre un projecte d'aquestes característiques no és altra que entendre quines són les necessitats d'un programador a l'hora de desenvolupar una aplicació web, i conèixer quines eines li poden posar les coses fàcils. Fixar-se en què sobra (o seria prescindible) dels *frameworks* actuals i què els falta (o seria interessant que tinguessin). Per tant, el marc teòric, o l'escenari que es pren com a punt de partida, no és altre que el disseny dels *frameworks* que s'han agafat com a referència: CodeIgniter, Symfony2, Laravel i Slim.

3.1 CodeIgniter

Projecte nascut de la mà d'EllisLab l'any 2006. És dels *frameworks* més ràpids i lleugers que hi ha. És potent i té una comunitat sòlida de desenvolupadors al darrera. Va patir una forta crisi quan va aparèixer Symfony2 al mercat i va perdre gran quantitat de membres de la comunitat que tenia, però va recuperar-se al moment en què el British Columbia Institute of Technology va adoptar el projecte i va donar-li suport i continuïtat.

Els punts més forts de CodeIgniter són la lleugeresa en quan a consum de recursos, la velocitat d'execució i la senzillesa de tota la lògica que implementa.

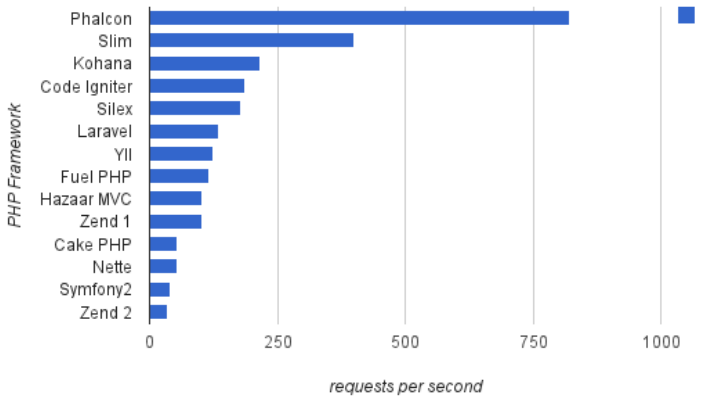

Figura 1 Diagrama de rendiment de diversos *frameworks* de PHP

El millor de CodeIgniter és la facilitat amb què es fan les coses. Si bé és cert que és pobre en alguns aspectes, com per exemple l'absència d'ORM[2] (*Object Relational Mapping*); el què fa que resulti una eina agradable de tracte és el fet que mentre programis, en qualsevol moment pots entrar al codi del nucli del *framework* i es pot entendre perfectament. És un codi molt net, ben estructurat i comentat de forma molt meticulosa per tal de justificar el per què de cada línia.

[2]ORM (*Object Relational Mapping*) en català és el mapeig d'objectes relacional i és una tècnica de programació que converteix les dades de la base de dades en objectes. Consisteix en afegir una capa d'abstracció a la base de dades.

CodeIgniter aparenta ser l'evolució lògica del desenvolupament de webs en llenguatge PHP, i és el millor candidat per un programador que s'iniciï en el desenvolupament amb el model MVC, ja que podrà navegar per tot el codi i entendre què es fa i per què. En certa manera, la sensació és de què CodeIgniter ha estat creat amb finalitats acadèmiques, i pot molt ben ser que això sigui per gràcies a l'adquisició de CodeIgniter per part del British Columbia Institut of Technology.

3.1.1 Objectius d'arquitectura i disseny

La màxima de CoreIgniter pel què a arquitectura i disseny respecte, és:

> *"Maximum performance, capability, and flexibility in the smallest, lightest possible package."*
> (Màxim rendiment, funcionalitat i flexibilitat en el paquet més petit i lleuger possible)

Per tant les prioritats són clares, i per tal d'aconseguir-ho han hagut de fer alguns sacrificis de disseny, ja que afegir gaires nivells d'abstracció et pot penalitzar el rendiment. Així doncs, a nivell tècnic els objectius que es van marcar són els següents:

- Instal·lació dinàmica: els components es carreguen i se n'executen les rutines només quan es demanen.
- Baix acoblament: quan hi ha un baix grau d'acoblament això implica que les interdependències entre components és baixa. Per tant els components són molt independents entre ells i podrien inclús ser utilitzats en un entorn diferent sense necessitat d'haver de disposar de tota la configuració del *framework.*
- Singularitat dels components: per tal d'aconseguir que els diversos components no s'acoblin entre ells el que cal fer és brindar un alt grau de focalització en l'objectiu de cada component. De tal manera que cada component és per si sol, no només autosuficient sinó també especialista en una funcionalitat concreta, ja que els components són tant atòmics i concrets que només es focalitzen en realitzar una tasca concreta i específica.

3.1.2 Flux d'execució

El flux de dades a través dels diversos components d'una aplicació desenvolupada en CodeIgniter és el següent:

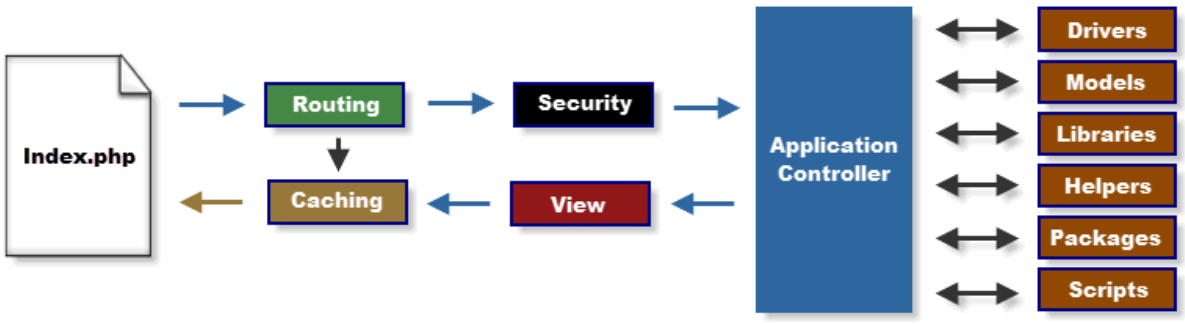

Figura 2 Flux d'execució d'una aplicació amb CodeIgniter

Sempre es parteix de l'arxiu index.php, que és qui rep la petició HTTP. Aleshores les dades comencen a circular. Primer reacciona el mòdul de Routing, que processa la petició per veure quina informació s'ha demanat. Aquest s'encarrega de traduir la ruta, continguda a la petició, en un nom de

controlador i un mètode concret a executar. Abans, però, es comprova la seguretat de les dades contingudes tant a la petició en si mateixa com a tot el conjunt de variables d'entrada ($_GET, $_POST, $_HEADER, …), per tal d'evitar possibles injeccions de codi maliciós o bé gestionar possibles problemes de codificació, etc.

En cas de trobar-nos davant d'una adreça vàlida, el mòdul de Routing obtindrà un controlador, que serà instanciat per posteriorment cridar-ne el mètode que s'ha decidit a partir de la ruta, o bé el mètode per defecte (que és la funció index() fins del controlador).

Podem observar, també, un mòdul de *Caching*. Aquest s'encarrega de servir arxius de forma automàtica, en cas de trobar-se davant d'una petició que resol a un fitxer que està en memòria (arxiu cachejat). D'aquesta manera el *framework* es pot saltar la resta d'execució, millorant enormement el rendiment i la velocitat de resposta. Aquest és un mecanisme molt útil que ens permet alliberar el servidor de gran quantitat de càlculs, a canvi d'un major consum de memòria. En cas de no tractar-se d'una pàgina que està en *cache*, però, el què es farà serà carregar el controlador, que a la vegada interactuarà amb tantes llibreries del nucli del *framework*, *helpers* (llibreries de suport) i models com siguin necessaris.

Finalment, també el controlador, tindrà la responsabilitat de definir un valor de retorn, que bé pot ser una vista, o una resposta estructurada amb formats tipus json, xml, etc. Aquests darrers formats, sovint utilitzats per a l'intercanvi d'informació entre client i servidor en aplicacions que funcionen a través de APIs (*Application Programming Interface*). En cas de tractar-se d'una vista, aquesta es renderitzarà, rebent com a paràmetres les variables que decideixi el controlador, i s'enviarà al navegador per ser visualitzada. I en cas que la *cache* estigui activada, aquesta vista serà guardada per ser servida en properes peticion.

3.1.3 Estructura interna

Pel què fa a l'esquelet de CodeIgniter a nivell de fitxers, quan descarreguem el *framework* ens trobem amb una estructura que conté tres carpetes i dos fitxers (entre altres coses que no tenen importància pel cas que ens ocupa):

- **/application**. És el directori on s'implementarà tot el codi de la web o aplicació en qüestió. Aquest conté diversos directoris per organitzar es fitxers segons la lògica que tenen associada.
- **/system**. Conté el nucli de CodeIgniter que fa que tot funcioni com ha de funcionar. Aquest contingut no s'hauria de modificar. En cas que no hi hagi més remei que modificar el codi contingut dins de system, és molt recomanable anar en compte a l'hora d'actualitzar l'aplicació a futures versions de CodeIgniter, ja que les actualitzacions de versió bàsicament es redueixen a substituir la carpeta system de la teva aplicació per la de la nova versió. Així que caldrà propagar els canvis efectuats a cada actualització
- **/user_guide**. És una guia d'usuari. Molt útil ja que així amb cada instal·lació del *framework* pots tenir a l'abast en local, sense

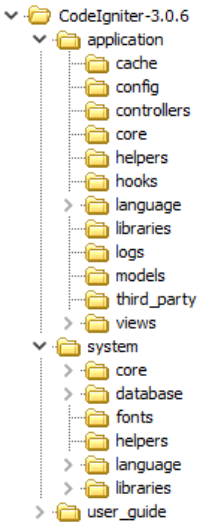

Figura 3 Organització interna del directoris de CodeIgniter

necessitat de recórrer a una connexió a Internet, tota la documentació requerida sobre tots els mòduls, i llibreries que incorpora CodeIgniter.

- **/index.php**. És el punt de partida de tota l'execució del *framework*. Actua de controlador frontal de tota l'aplicació.
- **/license.txt**. Defineix la llicència sota la qual es distribueix el projecte, que en aquest cas és una llicència MIT que pertany al British Columbia Institute of Technology.

A part, la distribució interna de les dues primeres carpetes (application i system) és la següent:

- **Application** conté una carpeta per a cada tipus de dada diferent que gestiona el *framework*. De tal manera que tenim una carpeta **config**, per exemple, que contindrà els arxius de configuració tant de l'entorn i del *framework* en sí mateix com d'alguns dels mòduls que es poden utilitzar dins de CodeIgniter. A més també tenim una carpeta pels controladors, una pels models i una per les vistes, però a part veiem altres directoris amb noms una mica més confusos, com podria ser **core, helpers** o **hooks.**
    - La carpeta **core** serveix per guardar-hi totes aquelles classes que extenen o modifiquen alguna cosa del nucli del *framework*. Si per exemple, es desitja que les variables de sessió funcionin d'una manera determinada, diferent de com les gestiona CodeIgniter, s'hauria de crear la classe dins de la carpeta Core i mitjançant un prefix concret, definit a l'arxiu de configuració, es pot fer que el sistema agafi directament les classes que han de substituir l'objecte bàsic que incorpora el *framework*.
    - La carpeta **helpers** contindrà col·leccions de funcions, organitzades per fitxers segons tipus concret de dades que gestionin les funcions en qüestió.
    - Els **hooks** són com petits trossos de codi que vols que es puguin executar en alguns moments concrets.
- **System** és, com ja s'ha dit, la carpeta que conté el nucli del *framework*, però quin és aquest nucli?
    - **Core**: conté les llibreries base de tot el *framework*.
    - **Database**: hi ha totes les implementacions per a poder connectar el projecte amb qualsevol tipus de sistema gestor de base de dades.
    - **Fonts**: el codi font de les tipografies utilitzades a les vistes per defecte que porta CodeIgniter de sèrie (per als missatges d'error, per exemple).
    - **Helpers**: incorpora una gran quantitat de llibreries de suport molt recomanades per a qualsevol programador. Des de formatació de números, passant per creació de formularis, fins a un gestor de captches (imatges de seguretat per a confirmar que l'usuari que t'envia una determinada informació no és un robot).
    - **Languages:** conté una traducció dels textos que ja porta de sèrie el *framework*, encara que d'entrada només afecten a nivell visual. Un exemple d'aquests textos podria ser la descripció d'un tipus d'error determinat.
    - **Libraries**: conté una gran col·lecció de classes ben estructurades que ens permeten treballar de forma més còmode i polida que no pas amb la carpeta de helpers.

3.2. Symfony2

Symfony és un *framework* que va començar a crear Fabien Potencier l'any 2003. L'objectiu desde el principi va ser aglutinar diverses eines el suficientment madures i estables, i integrar-les en un únic *framework*. Un any després de començar aquest projecte, va haver desenvolupat el nucli de Symfony, basat en el patró de disseny Model Vista Controlador, amb Propel com a ORM (*Object Relational Mapping*) i el gestor de plantilles de Ruby on Rails.

No va ser fins dos anys més tard, al 2005, quan Fabien Potencier va fer el llançament de la primera versió de Symfony. Originalment va ser creat per al desenvolupament de les aplicacions de SensioLabs (empresa de la qual Fabien Potencier és CEO) però després de l'èxit que va suposar el desenvolupament d'una pàgina web per a comerç electrònic, juntament amb algun altra projecte, va decidir alliberar-lo sota una llicència open source. I així va néixer la primera versió del Symfony que coneixem ara.

Les característiques principals que fonamenten aquest *framework* són:

- Facilitat d'utilització i d'instal·lació.
- És compatible tant amb sistemes Windows com Unix estàndards.
- Utilitza programació orientada a objectes i noms d'espai, que són una manera d'encapsular elements del codi.
- És senzill d'utilitzar en la majoria de casos, tot i que és preferible per a desenvolupar grans aplicacions web.
- Tot i utilitzar MVC, en fa una pròpia aproximació ja que conté algunes variants.
- Segueix la majoria de bones pràctiques que es recomanen seguir a nivell d'enginyeria de software, a més de utilitzar nombrosos patrons de disseny.
- És suficientment estable com per desenvolupar aplicacions a llarg termini.
- El codi resulta fàcil de llegir ja que inclou comentaris que permeten facilitar el manteniment.

En resum, els avantatges que ens ofereix Symfony a nivell tecnològic són: rapidesa i poc consum de recursos, a la vegada que flexibilitat, escalabilitat, estabilitat, comoditat i facilitat d'ús.

També cal recalcar que hi ha una gran quantitat de *bundles* (conjunts de funcionalitats diferents) de Symfony, amb la qual cosa es pot utilitzar una configuració diferent del *framework* segons les necessitats de cada projecte.

3.2.1 Objectius d'arquitectura i disseny

Entrar més en detall de com funciona i s'organitza el *framework*, ens centrarem en la versió 2 d'aquest, que incorpora millores substancials respecte la primera.

Tot seguit es mostra un diagrama on es mostra la interacció entre els diversos components que conté Symfony2, que com es pot veure a simple vista és bastant més complex que el de CodeIgniter.

Els components més importants que conté són els següents:

- Controladors: els controladors bàsicament contindran diversos mètodes que estaran associats a una petició HTTP, de la mateixa manera com passa amb la majoria de *frameworks*. Els controladors poden extendre el controlador base de Symfony per a disposar de les funcionalitats bàsiques que es poden necessitar a aquest nivell de l'aplicació, com per exemple:

    - Invocar el gestor de plantilles:

```
return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
```

    - Redireccionar la petició cap a una altra pàgina:

```
return $this->redirect($this->generateUrl('homepage'));
```

    - Setejar *cookies* o variables de sessió:

```
$this->get('session')->setFlash('notice','Your changes were saved!');
```

    - Etc.

- Enrutament: el component de *routing* de Symfony2 és bàsicament un mapeig per transformar les URL de les peticions que es reben en un mètode dins d'un controlador.
    - El fitxer de rutes pot estar especificat de tres maneres diferents:
        - YAML
        - XML
        - PHP
    - La utilització d'aquest tipus de sistemes ens permet obtenir dos beneficis bàsics que són:
        - URL més llegibles ja que és el programador qui controla l'adreça concreta, i no depèn del sistema de fitxers:
            - Per exemple, de /index.php?article_id=1 passaríem a tenir una URL que podria ser /read/article/1 o bé /read/intro-to-symfony.
        - No hi ha una relació directa entre un fitxer i la URL per accedir-hi, i per tant no depenem del nom d'aquests per a construir la nostra aplicació. Això és interessant per exemple, quan es vol poder mantenir múltiples versions d'una aplicació, en cas d'haver-hi codi comú, no ens caldria duplicar-ho tot i posar-ho en un altre directori sinó que ho podríem tenir tot junt i crear les funcionalitats de la versió nova en un arxiu diferent, o inclús al mateix, però en qualsevol cas les funcionalitats compartides no quedarien mai duplicades ja que tot estaria al mateix projecte.
    - Des del fitxer de rutes es pot especificar múltiple informació relativa a les peticions que acceptarà una ruta concreta, els valors per defecte que s'haurien de carregar, el

tipus de dades de les variables que es reben com a paràmetre a través de la URL, etc.

```
# app/config/routing.yml
blog:
	pattern:  /blog/{page}
	defaults: { _controller: AcmeBlogBundle:Blog:index, page: 1}
	requirements:
		page: \d+
```

A l'anterior exemple es pot observar com hi ha un paràmetre anomenat 'page', rebut a través de la URL, que tal com s'especifica als valors per defecte, valdrà 1 si no rep valor. A més, com a requeriments s'ha definit que el paràmetre 'page' ha de validar l'expressió regular \d+, i per tant s'ha de correspondre a un valor numèric d'un dígit o més, i si no es compleix aquesta condició, la URL no validarà aquesta regla i per tant no accedirà al controlador en qüestió que se li ha especificat (classe Blog del bundle AcmeBlogBundle), ni al mètode index d'aquest controlador.

- Plantilles: en el cas de la primera versió, ja s'ha comentat que utilitzava el motor de plantilles de Ruby on Rails, però a la segona versió va passar a utilitzar Twig. Els beneficis d'utilitzar un motor de plantilles són molts, però el principal és que s'evita posar lògica dins de les vistes.

  Twig és un motor de plantilles que es caracteritza per ser ràpid, segur i flexible, a la vegada que ofereix un llenguatge concís, amb una sintaxi que pretén ser mínima (es vol reduir la quantitat de text que no sigui contingut de la vista), una gran quantitat de funcionalitats i una gran facilitat de lectura, entre d'altres coses. Veiem un exemple de com heretar una plantilla per extendre-la.

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
		<title><{% block title %}Test Application{% endblock %}</title>
	</head>
	<body>
		<div id="sidebar">
			{% block sidebar%}
			<ul>
				<li><a href="/">Home</a></li>
				<li><a href="/blog">Blog</a></li>
			</ul>
			{% endblock %}
		</div>

		<div id="content">
			{% block body %}{% endblock %}
		</div>
```

```
        </body>
</html>
```

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
        {% for entry in blog_entries %}
                <h2>{{ entry.title }}</h2>
                <p>{{ entry.body }}</p>
        {% endfor %}
{% endblock %}
```

- Models: Symfony2 no utilitza models com a tal, sinó que utilitza un sistema ORM (*Object Relational Mapping*) que bàsicament funciona de la següent manera: el programador ha de definir una estructura d'objectes en PHP per a les diferents entitats amb què vol treballar, especificant quines relacions hi ha entre elles, i aleshores l'ORM s'encarrega de crear tant la base de dades com les SQL que duran a terme les diferents accions sobre les entrades concretes que es vegin afectades després de realitzar modificacions sobre els objectes de PHP. D'aquesta manera, sempre hi ha una relació directe entre els objectes amb què es treballa i el què conté la base de dades, però el programador no s'ha de preocupar de gestionar dites dades, sinó únicament els objectes de PHP. El sistema ORM que utilitza Symfony2 és Doctrine.
- Validacions: també incorpora una eina per validar les dades. D'aquesta manera ens podem assegurar que les dades que la informació que arriba al sistema a través dels diversos formularis que pugui contenir la nostra aplicació, és correcte i amb el format esperat.
- Formularis: i a part també conté una eina que ens permet crear formularis de forma molt senzilla:
    1. Es crea una classe.
    2. Es crida el mètode createFormBuilder passant-li la classe creada, des del controlador.
    3. Es defineix la plantilla a Twig per a renderitzar el formulari a una pàgina. S'utilitza el mètode form_widget, disponible a la plantilla.
- Seguretat: donat que la majoria d'aplicacions webs incorporen espais que requereixen autenticació, Symfony2 ja incorpora una eina que ens permet gestionar l'accés a usuaris que ja han estat autentificats, i revocar l'accés als qui encara no. Tot això mitjançant un fitxer configuració on s'especifiquen rols, usuaris, accessos, etc. Veiem-ne un exemple:

```
# app/config/security.yml
security:
        firewalls:
                secured_area:
                        pattern:       ^/      #qualsevol petició entrant
```

```
                    anonymous:     ~      #valor per defecte
                    http_basic:
                            realm: "Secured Demo Area"

    access_control:
            - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
            in_memory:
                    users:
                            user1: { password: user1pass, roles: 'ROLE_USER' }
                            user2: { password: user2pass, roles: 'ROLE_ADMIN' }

    encoders:
            Symfony\Component\Security\Core\User\User: plaintext
```

3.2.2 Flux d'execució

Com ja s'ha comentat, Symfony no segueix el model clàssic de patró MVC. Aquest patró com a tal, va ser inventat per a les aplicacions d'escriptori mentres que HTTP és *stateless* (no té estats), i per tant, és molt coherent adaptar el patró per cobrir les necessitats concretes que presenta una infraestructura diferent respecte l'entorn original on s'aplicava el patró MVC. Així doncs, Symfony es basa en la interacció pregunta – resposta que fonamenta la comunicació a través del protocol HTTP.

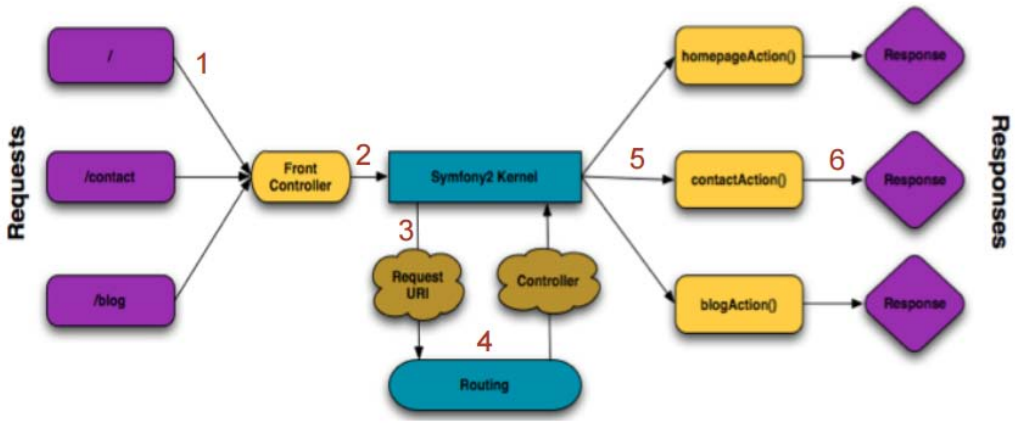

Figura 4 Diagrama de la interacció pregunta-resposta que presenta Symfony2

Com es pot veure al gràfic anterior, els diversos nombres indiquen el flux d'informació que segueix una aplicació desenvolupada amb Symfony:

1. El *front controller* rep una petició a través d'una URL concreta.
2. S'envia la petició al nucli de Symfony, qui realitzarà diverses comprovacions per validar que la informació està ben formatada i és correcte.
3. El nucli envia la petició al mòdul de *routing* perquè decideixi què s'ha de fer.
4. Dins del *routing* s'utilitza el fitxer de configuració per cercar quina és la regla que es confirma amb la URL continguda a la petició rebuda.
5. El nucli crida el mètode concret que ha obtingut del *routing* gràcies a les regles contingudes al fitxer de configuració.

6. El mètode retorna una resposta concreta, amb qualsevol format, depenent de la informació que s'hagi demanat a través de la URL, i depenent de com estiguin definides les regles d'enrutament.

3.2.3 Estructura interna

Les aplicacions amb Symfony 2 són conjunts del què ells mateixos (la comunitat) anomenen *bundles,* i cada *bundle* implementa un conjunt de funcionalitats. I tot això contingut en una estructura de fitxers basat en quatre carpetes que tindrà cada *bundle:*

- /app: conté els arxius de configuració de l'aplicació i el *mapping* del budle cap al seu fitxer de *routing* concret.
- /src: conté el codi de l'aplicació. És a dir que aquí van els controladors, les vistes, el fitxer de *routing* i la resta de fitxers que necessiti l'aplicació.
- /vendor: conté totes les llibreries proveïdes per servidors externs i que són necessàries per al correcte funcionament de l'aplicació. Aquesta col·lecció de llibreries està controlada mitjançant composer, que és una eina de control de dependències molt potent per a projectes PHP, la qual serà explicada amb més profunditat en apartats següents.
- /web: aquí hi ha el *front controller* de l'aplicació, que cridarà al nucli del *framework* per a iniciar l'execució de l'aplicació i respondre a la petició en qüestió. A més també hi ha tots els *assets* (col·lecció d'imatges, fulls d'estil, codis javascript, …) i, en resum, tots els arxius de la web que siguin d'accés públic.

3.3. Conclusions

Un dels punts que són importants a l'hora de realitzar una comparació entre qualsevol llibreria de codi, és el rendiment. I és un tema crític a l'hora de dur a terme la planificació de qualsevol projecte ja que triar un *framework* o un altre et pot condicionar la continuïtat del servei, en cas que es tracti d'un projecte d'alta disponibilitat, per exemple. Per tant, sempre que ens trobem davant la decisió de quina tecnologia triar, o quina plataforma utilitzar, així com quina volem que sigui la base de codi sobre la que es desplegarà la nostra aplicació, ens haurem de preguntar quin és l'ecosistema en què haurà de funcionar dita aplicació: la quantitat d'usuaris que n'hauran de fer ús, el grau d'intensitat amb què haurà d'operar, etc.

Una vegada tinguem clar quin és el rendiment que necessitem que tingui al nostre projecte, podem analitzar els altres aspectes, no menys importants, que han de condicionar l'elecció del millor *framework* per al projecte en qüestió.

Si bé és cert que hi ha *frameworks* que ofereixen molt més rendiment que els que s'han triat com a referència, cal destacar que, com ja s'ha comentat, el rendiment no ho és tot, i en moltes ocasions, serà preferible sacrificar rendiment a canvi de brindar un més alt nivell d'escalabilitat al codi, o bé més versatilitat en quan al conjunt d'eines de què disposa el *framework* en si. Però el què és evident és que els dos *frameworks* que s'han agafat com a referència són suficientment diferents i suficientment potents, tant l'un com l'altre, com per tenir-los de punt de referència. A més, les coses bones d'ambdós, són diferents, i per tant el còmput global dóna una bona col·lecció d'eines que es poden disposar a un *framework* per tal de fer-lo únic a la vegada que el suficientment interessant.

4. Continguts

El *framework* desenvolupat (d'ara en endavant també referit com a JepiFW) està dividit en diversos mòduls segons funcionalitats, de tal manera que tenim 12 directoris diferents, que es corresponen cadascun a una funcionalitat concreta, i entre ells tenen, en alguns casos, interdependències d'altres mòduls. Cal dir que els mòduls duen a terme tasques molt concretes, en la majoria dels casos, de tal manera que no són molt complexos en sí mateixos, tot i que junts aconsegueixin un *framework* amb una funcionalitat plena i prou rica.

Els diversos mòduls que componen el *framework* estan dins la carpeta System, que és la que constitueix el nucli. Aquests mòduls són els següents:

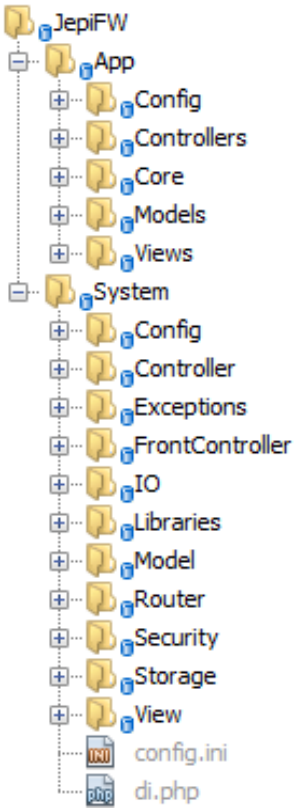

Figura 5 Organització interna del directoris de JepiFW

- **/Config**: conté la lògica necessària per administrar les variables de configuració, que es troben a un fitxer anomenat config.ini. Les variables de configuració són necessàries a molts punts del *framework* i és per això que les classes aquí contingudes són les satisfan més dependències.
- **/Controller**: la base dels controladors del *framework* és totalment necessària i indispensable, ja que ens permetrà utilitzar les funcionalitats del *framework* des del codi de la nostra aplicació.
- **/Exceptions**: les abstraccions del diversos tipus d'errors que pot gestionar el JepiFW.
- **/FrontController**: és el controlador principal. Inicialitza el *framework* i en crea els components necessaris per tal de poder gestionar les peticions i servir-les.
- **/IO**: consisteix en les estructures necessàries per a gestionar les dades d'entrada i de sortida que rebrà i enviarà el *framework*, respectivament. Tant les variables que ens arribin per post o get, com les respostes i el format amb què es serviran, tot això és gestionat des del mòdul d'IO (Input/Output en anglès, Entrada/Sortida en català).
- **/Libraries**: recull de llibreries que no estan catalogades com a elements del *framework* pròpiament dit, i que a més no tenen cap dependència del mateix. A data d'entrega, hi ha una classe i prou, que ens permet gestionar els fitxers i directoris. Donat que és fàcilment reutilitzable en qualsevol altra *framework*, està totalment deslligada del nucli d'aquest, per tal de minimitzar l'impacta que tindria.
- **/Model**: és la capa d'abstracció que ens permet accedir a les dades a través d'una connexió a base de dades i la gestió de les consultes a la mateixa.
- **/Router**: el mòdul de *routing* és l'encarregat de traduir les peticions HTTP que arriben al servidor en una funció a executar. Per defecte, el mecanisme que dóna el *framework* per dur a terme aquesta tasca és directe: una URL està directament relacionada amb un controlador que s'instanciarà i un mètode que se n'executarà. Tot i així, gràcies al disseny que disposa el *framework*, amb unes petites modificacions del mòdul de *routing* actual es podria fer que la traducció passés per un fitxer de rutes per decidir alternatives a una traducció directa.

- **/Security**: conté les classes que s'encarreguen de gestionar la seguretat. D'entrada el *framework* només incorpora un filtre per evitar atacs de tipus XSS (Cross-Site Scripting), que s'inclou a la classe XssFilter i bàsicament filtra una dada determinada per assegurar-se que conté dades d'un tipus permès i que no conté caràcters estranys, així com tampoc fragments de codi.
- **/Storage**: tenim un mòdul que s'encarrega de guardar dades a bases de dades, però una aplicació sovint necessita gestionar informació de forma temporal o informació de la pròpia execució, que no té prou transcendència o importància com per guardar-se a la base de dades. Així que el mòdul de *storage* s'encarrega de la gestió de dades temporals, com són les *cookies* o les variables de sessió.
- **/View**: finalment les vistes són gairebé quelcom imprescindible, perquè la majoria d'aplicacions requereixen de una part visual que permeti gestionar la informació o interactuar amb un usuari final. Per tant aquí dins hi ha les classes que s'encarreguen de gestionar la interfície d'usuari.

A més del directori /System, però, també es pot observar una carpeta /App que contindrà els fitxers de l'aplicació. El *framework* per defecte planteja una organització que consta de la separació de controladors, models i vistes, i a més contempla una quarta carpeta anomenada /core. Aquesta carpeta contindrà les ampliacions que es vulguin fer del *framework*. És a dir que si una funcionalitat base del *framework* no s'ajusta a les necessitats del programador aquí és on haurà de definir la nova classe que ha de substituir la classe en qüestió que no arriba a cobrir les seves necessitats.

5. Metodologia i eines

La metodologia a seguir determina en tot projecte l'èxit o fracàs del mateix, així doncs, és una decisió important. En un projecte de les característiques del que aquest document exposa, un projecte d'enginyeria del software, és important triar la metodologia que es seguirà durant el disseny i desenvolupament del mateix. En aquest cas s'ha optat per seguir una metodologia TDD[3].

El desenvolupament guiat per tests és un procés de desenvolupament de software que es basa en la creació de tests que en primera instància sempre fallaran, per posteriorment desenvolupar el mínim codi possible per validar aquests tests. I repetint aquest cicle s'aniran fent millores del software, sempre definint primer les regles d'acceptació, i després programant el codi que les passi.

El cicle detallat que ha de seguir un desenvolupament guiat per tests és el següent:

1. Triar un requeriment: de la llista d'especificacions del projecte, es tria un dels requeriments que se'n desprenen.
2. Escriure un test: el fet d'escriure el test abans de programar-ne la funcionalitat, que aquest validarà, obliga al desenvolupador a considerar el problema des del punt de vista del client final; en el cas que ens ocupa, el client final és un programador que utilitzi el *framework*.
3. Verificar que el test falla: en cas que el test s'executés amb èxit i no fallés, voldria dir que el requeriment ja estava implementat o bé que el test no és correcte. Així doncs, és important que un cop escrit el test, aquest falli.
4. Implementar el requeriment: cal escriure un codi el més senzill possible per fer que la prova no falli. La prioritat és passar el test, per això s'ha de minimitzar el codi que aconsegueixi fer-lo passar.
5. Executar tots els tests automatitzats: si tots els tests són passats, el programador pot estar segur que la funcionalitat implementada cobreix el requeriment objectiu sense generar cap error en els requeriments ja implementats anteriorment.
6. Eliminar duplicitats: és important netejar el codi eliminant regularment les possibles duplicitats.
7. Tornar al pas 1.

A més dels beneficis que aporta el fet que TDD sigui una metodologia tant pautada, ens ajuda a tenir un coneixement força fidedigne respecte el percentatge de completesa en què es troba el desenvolupament.

[3] *Test-driven Development,* en anglès, o desenvolupament guiat per tests, en català.

6. Procés de desenvolupament

El projecte s'ha dividit en quatre fases, dues d'elles clarament diferenciades per a ser focalitzades en desenvolupar el codi del *framework* i dues més per la part documental. Aquestes quatre entregues han suposat afrontar reptes molt diferents, ja que no es necessiten explotar les mateixes habilitats al programar que a l'escriure, i per tant s'han hagut d'afrontar des de diferents perspectives.

Les primeres dues entregues s'havia de fer tot des del punt de vista del programador: amb esperit crític i visió analítica de la situació; mentre que a l'hora de posar-se a escriure la memòria cal mirar-ho des del punt de vista d'una persona que no sap de què va el projecte i ha d'entendre-ho de la manera més clara i conscisa possible.

La primera fase del projecte pretenia obtenir un mínim codi que implementés el patró de disseny Model-Vista-Controlador, per a partir d'aquí poder-lo fer créixer en funcionalitats per tal d'obtenir quelcom més seriós que únicament el patró MVC. El què era necessari, però, era tenir coneixements de quines són les diverses aproximacions que es poden fer de l'implementació d'aquest mateix patró de disseny. Si bé és cert que hi ha patrons de disseny que són molt concrets i per aplicar-los has de seguir-los al peu de la lletra, en el cas del MVC s'ha comprovat que hi ha diverses maneres d'interpretar-lo. Un cop recopilada una bona pila d'informació sobre el tema, es van prendre les decisions que marcarien el desenvolupament de la versió més incipient del *framework*.

La segona fase ja pretenia incorporar la gestió de *cookies*, memòria de sessió, gestió d'entrada i sortida de dades, etc. Una de les parts més interessants del projecte va ser quan es va integrar amb les llibreries de tercers (PHP-DI i composer). L'arribada de PHP-DI va suposar donar-li un valor afegit al *framework* en quan a comoditat, mentre que l'ús de composer ajudava a posicionar el *framework* en un repositori global i amb molta activitat. D'aquesta manera automàticament t'obligava a programar pensant en què el teu codi estaria disponible per ser descarregat en qualsevol moment per qualsevol programador del món.

La tercera fase va incloure la creació de la memòria que bàsicament consistia en abocar tot el què havia après en un document. La part de disseny estava clara perquè era la que d'alguna manera havia originat el codi resultant, i per tant ja hi havia moltes notes preses durant el procés d'anàlisi i disseny i, posteriorment, de desenvolupament, és a dir que el traspàs de informació no va ser difícil, però si que va portar moltes més hores de les que es preveien.

I fins aquí el projecte, ja que la darrera fase únicament consisteix en polir aquest document i preparar la presentació i la defensa del mateix.

7. Arquitectura de l'aplicació

El JepiFW és un *framework* lleuger que vol brindar solucions senzilles a problemes habituals en el marc del desenvolupament web en llenguatge PHP. Com a punt de partida s'han escollit els *frameworks* CodeIgniter i Symfony, per conèixer quin és l'enfoc que tenien aquestes dues eines i així poder dissenyar una solució diferent per a problemes similars, sempre pensant el disseny per tal d'oferir el major grau d'escalabilitat. Tant és així que, cadascun dels mòduls que componen aquest *framework* poden substituir-se per d'altres equivalents però més complets, o menys, sempre i quan es mantinguin les abstraccions que defineixen cada mòdul.

Veiem doncs, el diagrama de classes que descriu el *framework* en la seva totalitat:

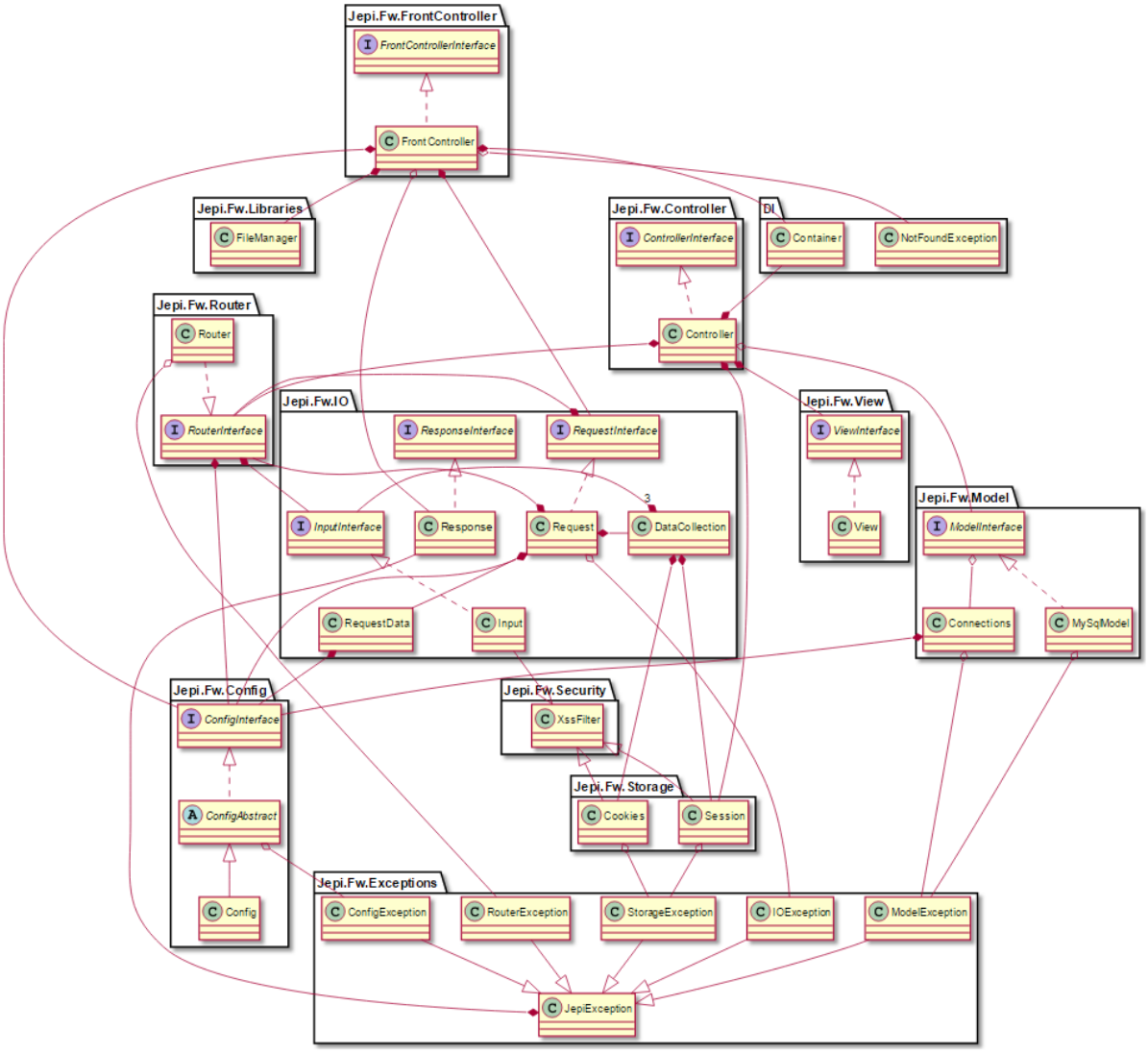

Figura 6 Diagrama de classes del JepiFW

Fixem-nos que al disseny hi ha moltes interfícies. Això és degut a què la majoria de components estan plantejats a partir del patró de disseny *abstract factory*, que proveeix d'una interfície per crear objectes dependents de la mateixa sense especificar les classes concretes. Així la relació entre components es pot fer a partir de les interfícies i es pot evitar utilitzar classes concretes i per tant el software és més abstracte.

El concepte d'abstració, a aquest nivell, implica que no ens cal conèixer el detall d'un component o d'una classe concreta perquè en tenim el detall de l'estructura que tindrà. Podem entendre la interfície que detalla l'abstracció com la descripció d'una família d'objectes, els quals cadascun pot tenir les seves peculiaritats, però sempre tindran com a mínim el què descriu la interfície.

Abans d'entrar al detall de cada mòdul, es pot veure com el *framework* està compost de diversos components, relacionats entre ells. I, seguint l'ordre lògic que seguiria l'execució d'una petició HTTP a través del *framework*, podem entendre el perquè de cadascun dels components. El següent diagrama de seqüència ens ensenya el detall de per on passa la informació dins el *framework*:

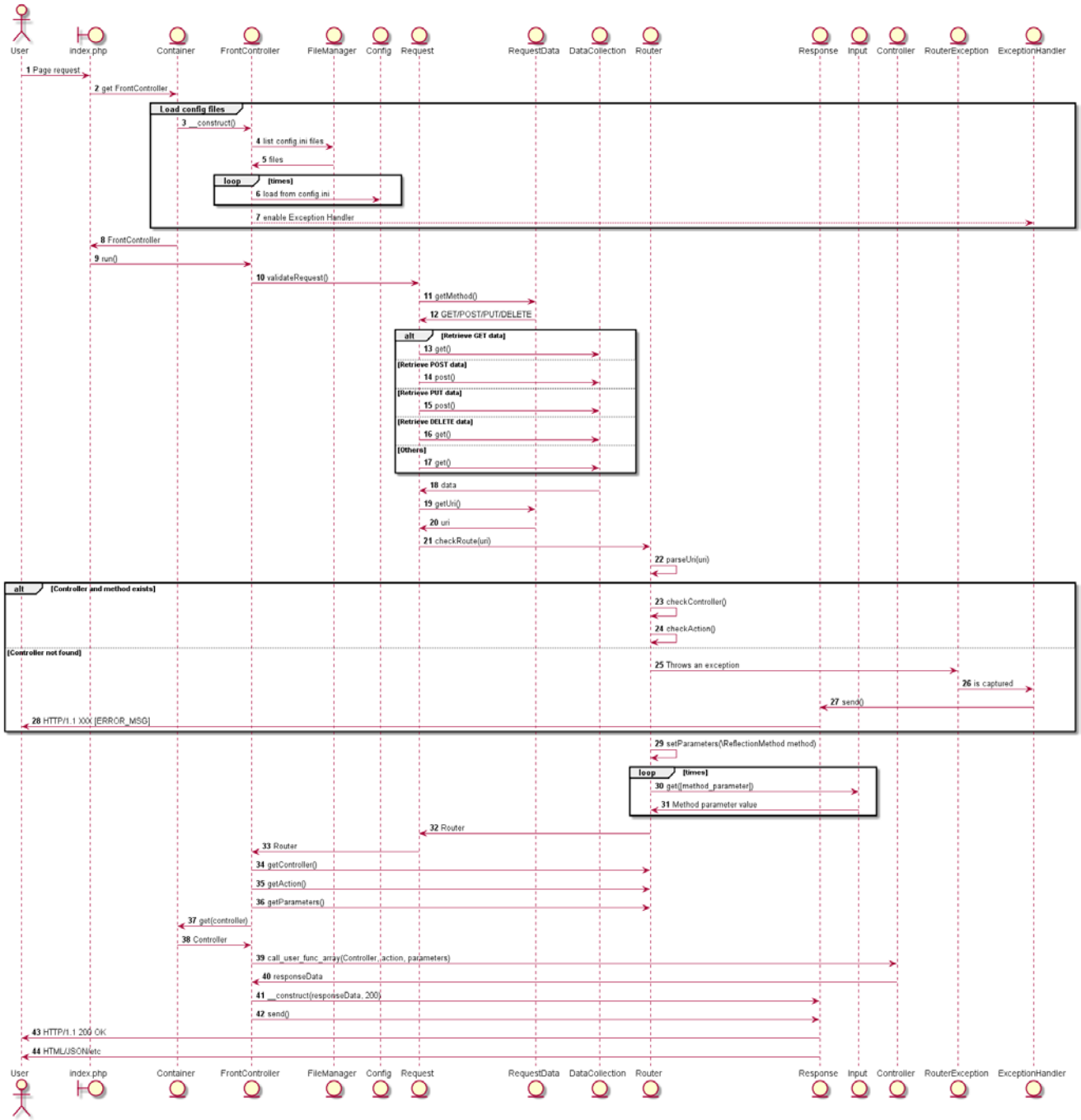

Figura 7 Diagrama de seqüència de la recepció i gestió d'una petició HTTP amb el JepiFW.

La interpretació simplificada del diagrama anterior és que el FrontController rep una petició les dades de la qual s'organitzen dins l'objecte Request, que a la vegada mitjançant la utilització de l'objecte Router decideix quin controlador ha d'instanciar. Seguidament el FrontController instancia el

controlador que el Router li indica a la Request, i aquest a la vegada disposa de l'objecte Container que pertany al mòdul de DI (Dependency Injection[4]) i que li permet instanciar de forma automàtica totes les dependències que tingui, que majoritàriament seran o bé models o bé l'objecte View, que li permetrà carregar vistes. En el cas de les instàncies de Model, aquests necessiten de la classe Connections, que gestiona les connexions a base de dades, creades a partir de les credencials que s'obtenen dels fitxers de configuració mitjançant l'objecte Config. Una vegada duts a terme tots els càlculs que s'hagin de fer amb les dades obtingudes dels models, i havent configurat les dades de sortida (que tant poden ser una vista com dades amb un format com Json o XML, per exemple), es retornaran les dades de sortida, que seran capturades pel FrontController com a resposta de la funció cridada del controlador instanciat, i es crearà un objecte de tipus Response que configurarà la sortida i retornarà les dades en resposta a la petició HTTP rebuda.

Si en qualsevol moment de l'execució es produís un error, aquest seria capturat també pel FrontController i es crearia una objecte Response amb el contingut de l'error on s'informaria a l'usuari de quin problema hi ha hagut i en quin mòdul s'ha produït.

Una vegada vist quin és el funcionament genèric del *framework*, ja podem entrar més en detall de quin és el funcionament concret de cada un dels seus components.

7.1 Inicialització del *framework*

Abans de què comencin a interactuar totes les classes del JepiFW, cal recordar que PHP és un llenguatge de scripting i no va ser fins a la versió PHP 3 que no es va incorporar el suport a la programació orientada a objectes, per tant l'estructura del llenguatge no situa l'inici del codi a una classe o una funció concreta com passa amb altres llenguatges amb suport natiu a la programació orientada a objectes, sinó que situa l'inici del codi a un fitxer concret que és l'index.php. Així doncs, necessitem definir un script que s'encarregui d'inicialitzar tota la informació necessària per a iniciar l'execució del *framework* com a tal.

Un altre element interessant pel què fa a la configuració d'accés als diversos fitxers que componen el *framework* és l'.htaccess. El fitxer .htaccess ens permet definir o extendre els paràmetres de configuració del servidor (que podríem trobar, per exemple, en el fitxer httpd.conf d'un servidor Apache). En el cas del JepiFW, el fitxer .htaccess conté la següent informació:

```
RewriteEngine on
RewriteBase /
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

RewriteCond $1 !^(index.php|license.txt|robots.txt|public)
RewriteRule ^(.*)$ index.php?$1 [L,QSA]
```

Figura 8 Contingut del fitxer .htaccess

[4] Dependency Injection, en anglès, injector de dependències, en català. Per més informació, veure apartat de llibreries externes.

El què ens diu aquesta configuració és que el directori arrel del *framework* és la carpeta on es troba l'.htaccess, i és relatiu a aquesta que s'aplicaran les regles que es defineixen seguidament. A partir d'aquí es defineixen un conjunt de condicions, seguides d'una regla de traducció que s'aplicarà en cas que aquestes es compleixin. Les condicions són les següents:

- La ruta que es sol·licita no és un fitxer
- La ruta que es sol·licita no és un directori
- La ruta que es sol·licita no acaba amb index.php, license.txt, robots.txt, public:
    - Aquesta condició defineix els fitxers o carpetes sobre les quals volem evitar de forma explícita que s'apliqui la regla de traducció.

Si es compleixen totes tres condicions, aleshores s'aplicarà la regla definida a la darrera línia, que diu que s'ha de redirigir la petició cap al fitxer index.php?$1, on $1 és una variable que conté la ruta de la petició inicial. I a més aquesta redirecció es fa amb els flags L i QSA activats. El primer flag serveix per evitar que es segueixin comprovant regles per a altres condicions definides al fitxer .htaccess, i el segon flag ens permet fer extensius tots els paràmetres GET que incloïa la ruta petició original a la nova ruta definida per la regla de traducció. Un exemple pràctic de com funcionaria aquesta regla és el següent:

El domini http://jepifw.jepihumet.com/ conté una web que utilitza JepiFW com a *framework* de PHP. Si realitzem una crida a la ruta /todo/tasks?id=1, aquesta és interpretada pel nostre arxiu .htaccess i donat que no és un fitxer, ni un directori, ni coincideix amb cap de les rutes definides com a excepcions, aquest decideix que valida totes les condicions i en fa la traducció. Per tant la petició que originalment no coincidia amb cap ruta existent de la nostra estructura de fitxers s'acaba enviant a /index.php?todo/tasks/&id=1. I aquest fitxer és el que serà el fitxer principal que s'encarregarà d'arrencar el *framework* i iniciar tot el què cal.

El fitxer index.php en sí és molt bàsic, però perquè part de la lògica la ubica a un altre fitxer anomenat bootstrap.php. L'index.php del JepiFW conté el següent codi:

```php
/** @var DI\Container $container */
$container = require_once __DIR__.'/bootstrap.php';
/** @var Jepi\Fw\FrontController\FrontController $frontController */
$frontController = $container->get('Jepi\Fw\FrontController\FrontController');
$frontController->run();
```

Figura 9 Fitxer index.php del JepiFW

El què fan les tres línies del fitxer index.php és obtenir un objecte de tipus Container, que en el cas en que ens ocupa s'ha decidit utilitzar el de la llibreria PHP-DI (per més informació veure secció de llibreries externes). Aquest objecte és l'encarregat d'instanciar tots els objectes que necessitem durant l'execució del *framework* i guardar-los en un contenidor, d'aquesta manera ens podem despreocupar de les dependències entre components. Seguidament, un cop tenim el contenidor únicament ens cal sol·licitar-li una instància de la classe FrontController, i cridar-ne el mètode `run()`.

El motiu pel qual s'ha separat la inicialització del *framework* en dos fitxers (boostrap.php i index.php) és que al haver plantejat el desenvolupament com a un projecte de desenvolupament basat en tests, ens resulta molt útil tenir un fitxer que ens retorni un gestor de dependències, perquè la llibreria

phpunit que s'utilitza per a definir tots els *test cases* ens permet passar-li la ruta d'aquests fitxer per tal d'oblidar-nos de instanciar totes les classes que siguin necessàries a l'executar un test. Això pot semblar poca cosa, però donat que cada component, en més o menys mesura, té dependències d'alguns altres components i aquests també poden tenir-ne amb d'altres, si no disposéssim del gestor de dependències, hauríem d'instanciar manualment totes i cadascuna de les classes que podrien ser necessàries durant l'execució *end-to-end* del test, i això per cadascun dels tests de cada component.

A més, aquesta pràctica s'ha convertit en pràctica habitual en la majoria d'eines de desenvolupament, ja que cada vegada es tendeix més a realitzar desenvolupament basat en test.

Vegem com queda el fitxer bootstrap.php:

```php
error_reporting(E_ALL);
ini_set("display_errors", 1);
date_default_timezone_set('UTC');

if (!defined('ROOT')) {
    define('ROOT', dirname(__FILE__));

    define('FW_ROOT', ROOT . DIRECTORY_SEPARATOR . 'JepiFw');
    define('APP_ROOT', FW_ROOT . DIRECTORY_SEPARATOR . 'App');
    define('SYSTEM_ROOT', FW_ROOT . DIRECTORY_SEPARATOR . 'System');
}
$loader = require __DIR__.'/vendor/autoload.php';

$containerBuilder = new \DI\ContainerBuilder;
$containerBuilder->addDefinitions(SYSTEM_ROOT.'/di.php');
$containerBuilder->useAnnotations(true);
$container = $containerBuilder->build();

return $container;
```

Figura 10 Fitxer bootstrap.php del JepiFW

Primer es comença redefinint paràmetres de configuració de PHP per assegurar-nos que independentment de com estiguin definits per defecte, són els que ens interessen. Aquests tres primers paràmetres defineixen el sistema horari en UTC (Coordinated Universal Time), i forcen a què els errors que es produeixin, no siguin obviats, per tal que el *framework* els pugui gestionar. Seguidament es defineixen diverses constants relatives a les rutes base de la carpeta arrel del *framework*, la carpeta on es troba l'aplicació i la carpeta on s'ubica el nucli del *framework*. Tot seguit s'inicialitza l'autoloader, que en el cas del JepiFW utilitzem l'autoloader de composer (per més informació veure secció de llibreries externes). Aquest objecte s'encarrega d'incloure els fitxers necessaris abans de què s'instanciïn les classes que aquests contenen.

I finalment s'instancia la classe ContainerBuilder, que ens permet definir les característiques del contenidor que volem utilitzar, per posteriorment construir-lo i ser retornat.

7.2 Configuració

L'objectiu del mòdul de configuració és únicament administrar les variables de configuració, que poden carregar-se a partir d'un fitxer o bé definir-se en temps d'execució.

El fitxer de configuració és un arxiu amb extensió .ini que conté o variables de configuració necessàries per al propi funcionament del *framework* o variables definides pel programador. Els fitxers .ini estan organitzats en seccions, per tal d'agrupar les sentències en múltiples nivells, i en el nostre cas utilitzem les seccions per referir-nos a funcionalitats o grups de configuració, com per exemple *Database*. La manera com es defineix l'arxiu config.ini que utilitza el JepiFW és la següent:

```
[Database]
defaultConnection = default

host[default] = localhost
port[default] = 3306
user[default] = foo
pass[default] = foobar
name[default] = foo

host[todo] = localhost
port[todo] = 3306
user[todo] = todo
pass[todo] = CaMFAsyzeb9D7nyL
name[todo] = todo
```

Figura 11 Exemple de definició de variables de configuració al fitxer config.ini

A l'anterior exemple es pot observar com s'organitzen les variables necessàries per a realitzar connexions a base de dades. Veiem també, que hi ha dues connexions diferents, i que s'especifica de les dues quina és la que s'utilitzarà per defecte (per més detalls, veure manual d'ús).

El mòdul de configuració consta de tres classes, que defineixen tres nivells d'abstracció. El primer és la interfície ConfigInterface, on definim els mètodes que necessitem que tingui la classe Config. Quan ens referim a què els mètodes són necessaris implica que són necessaris perquè s'utilitzen a altres components del *framework*. Després també hi ha la classe abstracte ConfigAbstract, que és on s'ha definit una possible implementació dels mètodes de la interfície. I per acabar tenim la classe concreta Config, que incorpora tres mètodes més, que ens serviran per a gestionar la lectura de fitxers de configuració.

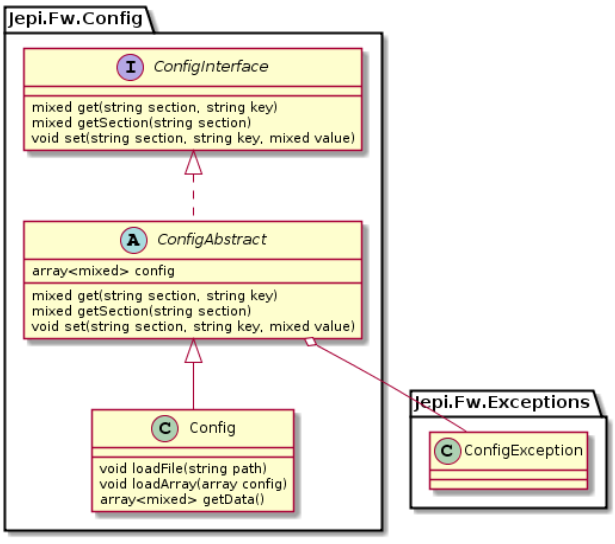

Figura 12 Diagrama de classes del component Jepi.Fw.Config

7.3 FrontController

EL FrontController, és el punt de partida. És la classe que s'encarrega de gestionar l'execució, i per aquest motiu està relacionada amb tants components, perquè necessita les instàncies de diversos objectes per a preparar el *framework*.

Veiem que el FrontController només disposa de dos mètodes públics que són el constructor i `run`. El darrer serà cridat a l'index.php per tal d'iniciar tot el procés. Aquest mètode ja està definit a l'interficie que impementa FrontController; i per tant dedïm que aquest component també utilitza el patró de disseny *abstract factory*. L'objectiu final del mètode `run` no és altra que instanciar un controlador i cridar-ne un mètode concret passant-li una sèrie de paràmetres, que això es fa amb un mètode de PHP anomenat `call_user_func_array`. La qüestió és que prèviament haurem de validar que tota la informació rebuda és correcte, i per això la classe té tantes dependències, perquè ha d'interactuar amb altres components per observar els resultats relatius a les dades que gestionen. Un exemple clar és el cas de l'objecte Request, que ens serveix per validar la informació de la petició HTTP, o bé la instància de la classe Router, que ens proporciona el nom del controlador i del mètode que s'hauran d'instanciar i cridar, i els paràmetres que se li hauran de passar. Finalment també veiem que s'utilitza la classe Response que permet formatar la resposta per a què tots els missatges que surtin del JepiFW tinguin el mateix format, o almenys estiguin gestionats per la mateixa entitat.

```php
public function run() {
    $router = $this->request->validateRequest();

    $controller = $router->getController();
    $action = $router->getAction();
    $parameters = $router->getParameters();

    $output = call_user_func_array(array($this->container->get($controller), $action), $parameters);
    $response = new Response($output, 200);
    $response->send();
}
```

Figura 13 Mètode run de la classe FrontController

Al diagrama de classes que es detalla a la Figura 11, es mostren les propietats de FrontController, entre d'altres coses. Una d'elles és del tipus DI.Container, que és una classe externa que s'ha utilitzat per fer la gestió de dependències dins dels controladors, i a part, aquí la utilitzem per instanciar el controlador en si mateix. Per més informació sobre el component de DI, veure l'apartat de llibreries externes.

Gràcies al diagrama, també es pot identificar l'ús que es fa d'altres patrons de disseny com és el *facade pattern*, que busca principalment reduir el grau d'acoblament del disseny i seguir la llei de Demeter [5]. El patró el podem detectar al constructor, quan es detecta que s'estan passant com a paràmetres les instàncies dels objectes que componen la classe. És interessant utilitzar aquest patró de disseny aquí perquè novament, ens brinda beneficis pel què fa a l'ús de interfícies. Vegem com els paràmetres que se li passen són tot interfícies i no classes finals, i això és perquè, d'aquesta

[5] La llei de Demeter són unes regles per al desenvolupament de software, bàsicament mitjançant paradigmes orientats a objectes, i que pretén reduir l'acoblament entre unitats del software. La llei de Demeter o principi del menor coneixement, es basa en reduir la informació que es coneix d'un objecte dins d'un altre, al mínim; de tal manera que un depengui de l'altre en el menor grau possible.

manera, a fora del *framework* es poden definir les classes que s'utilitzaran i independentment de la implementació que facin d'una funcionalitat, si segueixen l'estructura definida a la interfície de la classe, no hi haurà problemes.

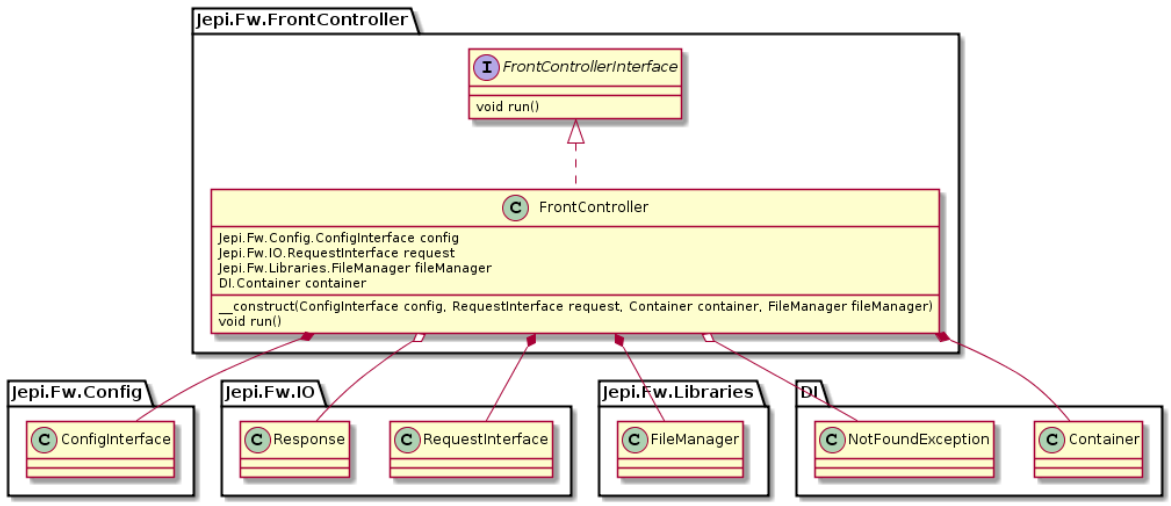

Figura 14 Diagrama de classes del component Jepi.Fw.FrontController

El constructor de la classe és ben senzill, ja que únicament assigna els valors rebuts com a paràmetres a les corresponents variables que caracteritzen la classe FrontController i crida el mètode privat `initErrorManagement`, que s'encarrega de capturar totes les excepcions que es produeixin durant l'execució per treure'n un missatge en conseqüència.

```php
private function initErrorManagement() {
    set_exception_handler(function (\Exception $e) {
        //Load Error View
        $traceStr = $e->getTraceAsString();
        $trace = nl2br($traceStr);
        $errorMsg = $e->getMessage();
        $implements = class_implements($e);
        if (in_array('JepiException', $implements)) {
            $errorType = $e->getExceptionType();
        } else {
            $errorType = 'PHP Error';
        }
        $errorCode = $e->getCode();
        if ($errorCode == -1) {
            $errorCode = 500;
        }

        $content = sprintf('<h2>Error %s: %s</h2><p>%s</p></br>%s',
                $errorCode, $errorType, $errorMsg, $trace);

        //Create Response and send it
        $response = new Response($content, $e->getCode());
        $response->send();
    });
}
```

Figura 15 Funció per gestionar la captura d'excepcions a FrontController

7.4 Gestió de rutes

Gestionar les rutes és imprescindible per a qualsevol *framework*, ja que les URL són el mitja a través del qual una aplicació web demana informació a un servidor. Així que per tal de facilitar aquesta comunicació, cal donar a les eines de desenvolupament els mecanismes necessaris per a definir regles d'enrutament.

El component Router també aplica un *abstract pattern* per a definir els mètodes bàsics d'enrutament amb què interactua tot el *framework* i, d'aquesta manera, si un desenvolupador volgués aplicar un altre tipus de regles, podria crear la classe final que gestionés les rutes de la manera que més ho desitgés.

```
[Routing]
SiteUrl = http://www.domini.cat/
DefaultController = Home
DefaultAction = index
AutoRedirect = false
```

Figura 16 Secció de configuració per el component de *routing*

El sistema d'enrutament que utilitza el JepiFW determina el nom del controlador i el mètode a executar a partir dels dos primers fragments de la URL, separats pel caràcter '/'.En cas de què la URL tingui dos fragments, el segon explicita el mètode, sinó s'executarà el mètode que es defineixi com a valor per defecte al config.ini. En cas de no haver-hi un primer fragment a la URL, s'utilitzaria el controlador configurat com a valor per defecte al config.ini.

Seguint l'exemple plantejat a la secció d'inicialització del *framework*. Al domini http://jepifw.jepihumet.com/ tenim una aplicació funcionant amb el JepiFW, i si ens dirigim a la ruta /todo/tasks?id=1 el mòdul de *routing* el què farà serà executar el mètode `tasks($id)` del controlador Todo. Veiem que dit mètode rep un paràmetre anomenat `$id`, que serà setejat a partir del valor que s'ha enviat com a paràmetre dins la variable súperglobal $_GET, nativa de PHP.

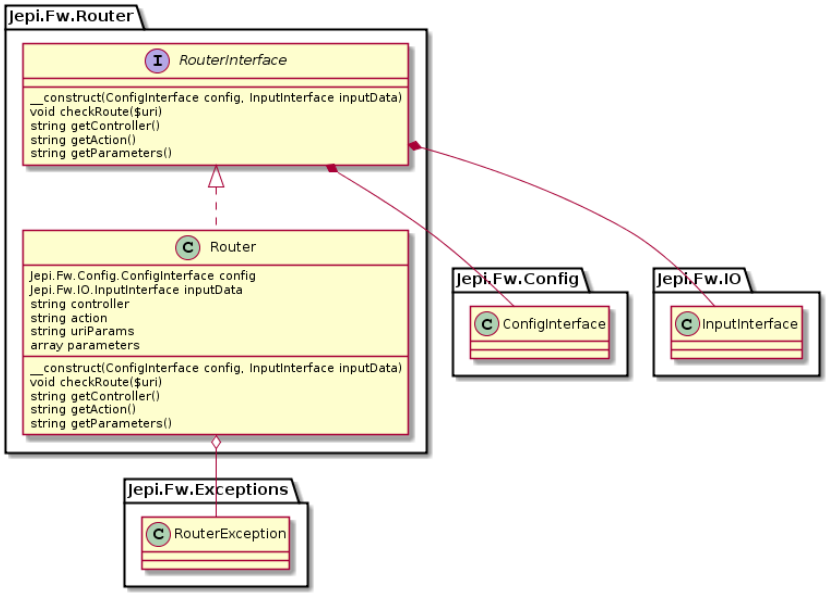

Figura 17 Diagrama de classes del component Jepi.Fw.Router

7.5 Gestió de dades

La gestió de dades està dividida en dos components, que són els models, que gestionen dades persistents a la base de dades, i el component de Jepi.Fw.Storage, que s'encarrega de gestionar les dades temporals. Aquestes dades temporals poden ser de dos orígens diferents: variables de sessió o *cookies*.

Les variables de sessió són la manera de guardar informació com a variables per a ser utilitzades a través de múltiples pàgines. Aquesta informació és guardada al servidor i està vinculada a un identificador únic. La necessitat de variables de sessió sorgeix del fet que, a diferència d'una aplicació que quan l'obres, realitzes canvis i la tanques, l'aplicació sap exactament què fas, quan comences i quan acabes, en el cas dels servidors web això no és així. La infraestructura d'un entorn web consta de dos components clarament diferenciats: el client i el servidor; i el servidor no sap què fas al costat del client a no ser que li ho comuniquis, ja que cada comunicació entre ells sempre és única en si mateixa. Gràcies a l'identificador de sessió, però, podem vincular totes les connexions entre un client i un servidor per tal de considerar-les parts d'una mateixa sessió.

A diferència de les variables de sessió, que són emmagatzemades al servidor, les *cookies* són informació que s'emmagatzema al client, però tenen una naturalesa similar, ja que el seu objectiu també és poder gestionar informació a través de les diferents pàgines que constitueixen un servei o una aplicació web. A diferència de les sessions, que és el servidor qui en determina el temps que tardarà en esborrar-se la informació des de la darrera connexió, les *cookies* al moment en què es defineixen, porten associada una data en què expirarà aquella informació a no ser que s'actualitzi posteriorment a la creació de la mateixa. I aquesta data d'expiració és individual per a cada *cookie*.

Tot seguit es mostra el diagrama de classes del component de *Storage* que ens permet gestionar ambdues tipologies de variables temporals:

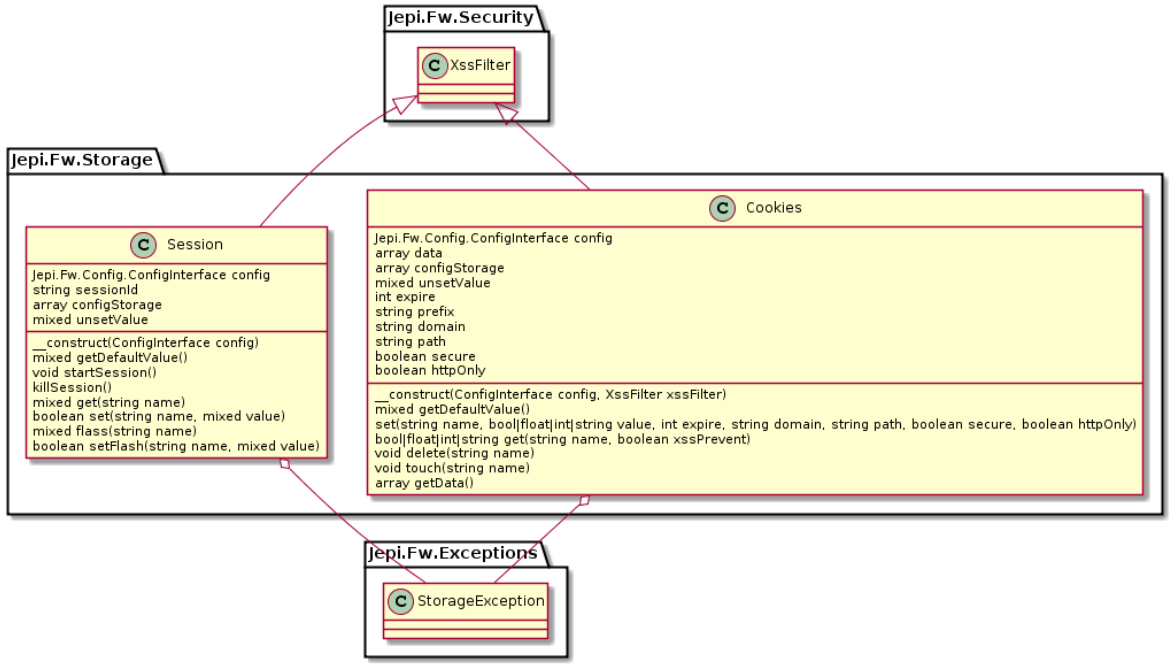

Figura 18 Diagrama de classes del component Jepi.Fw.Storage

Pel què fa a les variables de sessió, són gestionades per la classe Session, que primerament inicia la sessió amb el mètode `startSession` que crida la funció `session_start` de PHP i a partir d'aquí ja es permet afegir, actualitzar o eliminar variables de sessió. I a més el JepiFW permet gestionar un tipus de variables a les quals s'ha anomenat variables *flash*, que són elements d'una sola lectura.

I les *cookies* veiem que tenen molts altres paràmetres de configuració que les variables de sessió. Això és perquè les *cookies* venen definides per múltiples variables, com són la data d'expiració, el domini i *path* on s'ubiquen, etc. Tal com està dissenyat el *framework*, aquests paràmetres es poden definir de forma individual per a cada variable o es poden definir al fitxer de configuració config.ini per tal de ser recuperats d'allà i utilitzats amb totes les *cookies* de igual forma.

```
[Cookies]
DefaultExpiration = 60000
DefaultPrefix = ''
DefaultDomain = ''
DefaultPath = '/'
DefaultSecure = false
DefaultHttpOnly = false
```

Figura 19 Secció de configuració per a es *cookies*

De forma genèrica també es poden definir dos paràmetres de configuració més, relatius a la seguretat. Aquests fan referència a la possibilitat de xifrar les dades que gestiona el component de *Storage* i en cas de desitjar-se que es xifrin, amb quina clau.

```
[Storage]
Encrypt = true
EncryptionKey = blablabla
```

Figura 20 Secció de configuració del component Jepi.FW.Storage

7.6 Seguretat

El component de seguretat del JepiFW únicament incorpora la protecció davant d'un atac, que és el XSS[6], possiblement el més important a tenir en compte. És tan important que multitud de classes extenen la classe XssFilter per poder aplicar el filtre `xssPreventFilter`, ja que totes les dades que venen del client poden portar codi maliciós injectat.

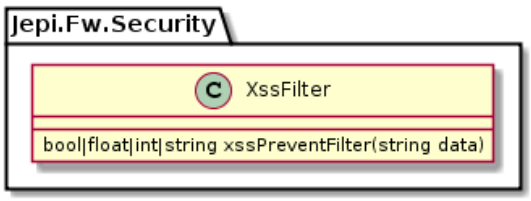

Figura 21 Diagrama de classes del component Jepi.Fw.Security

[6] *Cross-site scripting*, per més informació sobre aquest atac, consultar l'apartat de Seguretat.

7.7 Mòdul d'entrada i sortida (IO)

El mòdul d'entrada i sortida és el més gran de tot el *framework* ja que incorpora tots els elements relatius a la comunicació entre client i servidor, en ambdues direccions. I a part, s'ha seguit la mateixa filosofia que amb la resta de mòduls, que és la de basar el disseny en l'ús d'abstraccions per a cada entitat.

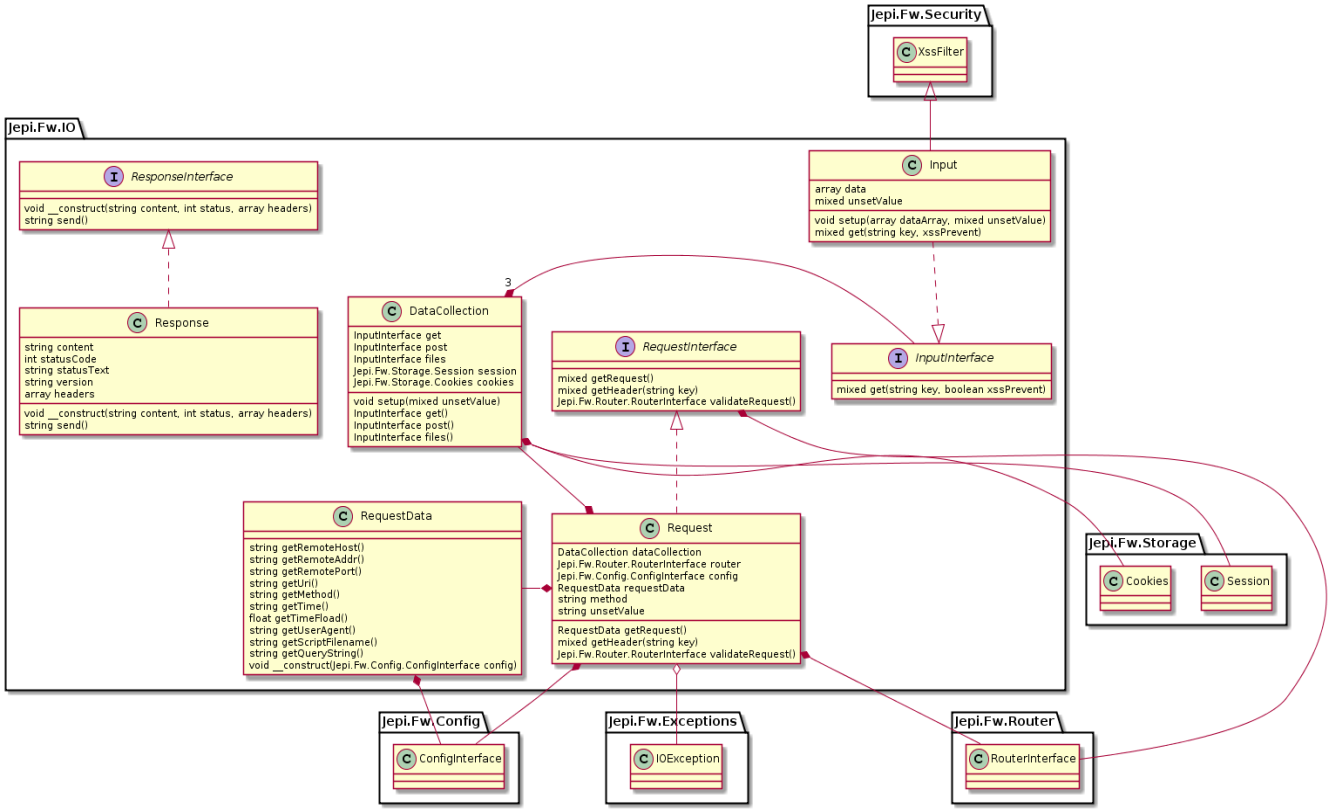

Figura 22 Diagrama de classes del component Jepi.Fw.IO

Les diverses classes que apareixen a l'anterior diagrama constitueixen els objectes Response, RequestData, Request, Input i DataCollection. Tant Response, com Request, com Input, implementen la seva corresponent interfície, que novament ens serveix per a determinar l'estructura necessària perquè el *framework* pugui funcionar sense problemes, però a la vegada dóna l'opció al programador d'implementar o extendre aquestes funcionalitats al seu desig.

La classe RequestData serveix d'interfície de la variable súper global $_SERVER per a l'aplicació. Al instanciar aquesta classe es guarda el valor dels paràmetres més necessitats en el desenvolupament web pel què respecte les dades contingudes dins de $_SERVER. Aquest conjunt de dades són:

- REMOTE_HOST: el host on s'està rebent la petició.
- REMOTE_ADDR: adreça IP del servidor en el qual s'està executant l'script.
- REMOTE_PORT: port a través del qual s'ha establert la connexió.
- REQUEST_URI: url que referencia la pàgina que es demana.
- REQUEST_METHOD: el mètode que s'ha utilitzat per accedir a la pàgina (GET, HEAD, POST, PUT, etc.).
- REQUEST_TIME: el *timestamp* de l'inici de la request.
- REQUEST_TIME_FLOAT: el *timestamp* de l'inici de la request amb precisió de microsegons.

- HTTP_USER_AGENT: és un text que defineix l'agent que està accedint a la pàgina. Ens permet conèixer els detalls del navegador que està utilitzant el nostre client.
- SCRIPT_FILENAME: la ruta absoluta de l'script que s'està executant.
- QUERY_STRING: el text utilitzat per a formalitzar la petició.

Així doncs, totes les variables recentment esmentades, estan emmagatzemades dins RequestData per a servir-les mitjançant els *getters* corresponents.

La classe Input defineix un contenidor de dades en format diccionari (clau – valor), i permet accedir a les claus del mateix assegurant que no es produirà un error en cas de sol·licitar una clau inexistent. Per assegurar aquesta condició, es passa la variable `$unsetValue` que indica el valor que prendrà el retorn de la funció `get` en cas de no rebre una clau existent dins el contenidor. A més, en cas que la clau si que existeixi, s'aplicarà un filtre al valor per tal d'evitar que ens realitzin atacs com l'XSS (Cross-Site Scripting).

La classe DataCollection és continguda per Request, i centralitza totes les dades d'entrada que pot tenir l'aplicació: variables POST i GET, fitxers d'entrada, variables de sessió i *cookies*. Els dos últims tipus de variables únicament s'han d'instanciar dins la classe per poder-ne disposar, els altres tres, però, requereixen d'una inicialització una mica especial. Les variables POST i GET s'envien a través de les variables súper globals $_POST i $_GET, respectivament. I els fitxers s'envien a través de l'accés temporal php://input, per tant, la classe DataCollection necessita disposar d'un mètode que accedeixi a tota aquesta informació i la emmagatzemi a la classe per tal de permetre utilitzar-la de forma còmode. Aquest mètode és el `setup`, que fa exactament això:

```
public function setup($unsetValue){
    $inputJSON = file_get_contents('php://input');
    $files = json_decode($inputJSON, TRUE);

    $this->get = new Input();
    $this->get->setup($_GET, $unsetValue);
    $this->post = new Input();
    $this->post->setup($_POST, $unsetValue);
    $this->files = new Input();
    $this->files->setup($files, $unsetValue);
}
```

Figura 23 Funció setup de DataCollection, que inicialitza els objectes per gestionar les dades d'entrada.

Veiem que crea una instància de la classe Input per a cada tipus de dades, i els inicialitza amb la variable súper global o el contenidor de dades que li correspongui. A més es passa com a segon paràmetre una variable anomenada `$unsetValue`, que és el valor que retornaran les funcions d'accés a valors, dels diversos tipus de variables, en cas que no existeixi el què s'està demanant.

La classe Request s'utilitza per recuperar i gestionar totes les dades de la petició que es realitza al servidor, i ho fa a través de l'objecte RequestData. Aquesta classe conté tota la informació necessària per a processar el què s'està demanant, a part que centralitza tots els tipus de variables pels diversos canals des d'on poden obtenir-se: variables POST, GET, de sessió i cookies; contingudes a l'objecte de tipus DataCollection. El sentit d'aquesta classe no és altre que la necessitat de tenir un únic lloc que serveixi de referència per gestionar tota la informació continguda en la petició que s'ha fet, per tal de tenir un punt de referència i no dispersar les dades en molts

components separats, com passa en altres *frameworks*, que per aconseguir un desacoblament major, separen els components, fent més difícil l'accés del mateixos per culpa de la dispersió. En el cas del JepiFW s'ha cregut convenient crear aquesta dependència a favor d'incrementar la facilitat d'ús del *framework*.

La classe Response únicament serveix per emmagatzemar les dades relatives a la resposta que es donarà a la petició rebuda. El fet d'utilitzar una única classe per a formatar la resposta ens assegura que la informació que rebrà el client sempre estarà estructurada de la mateixa manera, la qual cosa ens assegura estabilitat pel què fa a la comunicació entre client i servidor.

El constructor de Response rep tres paràmetres que són el contingut a retornar, l'*status* i les possibles capçaleres que es vulguin afegir a la sortida. El protocol HTTP ha de retornar una capçalera que indiqui si l'execució ha estat un èxit o no, i ho ha de fer amb un format concret:

HTTP/[Versio] [Staus] [Missatge]

Per exemple, la versió utilitzada pel JepiFW és 1.1, i per tant els missatges quedarien així:

- En cas d'èxit:
  HTTP/1.1 200 OK
- En cas d'error al servidor:
  HTTP/1.1 500 Internal Server Error

Veiem, per tant, que el paràmetre més important que rebrà la classe Response és l'*status*, ja que és el que ens permetrà indicar al navegador si l'execució ha estat satisfactòria o hi ha hagut cap problema. Però per no haver d'afegir un paràmetre més al constructor, i haver de gestionar el missatge de la capçalera desde tots i cada un dels controladors i mètodes, això es fa de forma automàtica dins de l'objecte Response, al moment d'instanciar-se l'objecte. Així doncs, al constructor es va a buscar el missatge que es correspon al codi d'estat rebut, per tal de retornar-lo a les capçaleres juntament amb el codi en el format especificat anteriorment.

El protocol HTTP des de la seva versió 1.0 contempla 4 rangs de *status codes*, que són els següents:

- Els codis 2xx estan reservats per a missatges d'èxit.
- Els codis 3xx estan reservats per a redireccions.
- Els codis 4xx estan reservats per als errors del client.
- Els codis 5xx estan reservats per als errors del servidor.

Així doncs, veiem que la resta de codis no tenen una restricció imposada per protocol sinó que queda obert. Això vol dir que podem utilitzar qualsevol codi que no estigui reservat, per això a la taula que es mostra tot seguit n'hi ha alguns que no estan especificats a cap RFC, perquè de la mateixa manera que hi ha protocols que especifiquen codis d'estat que no es contemplaven a la definició inicial del protocol HTTP, el programador també té la possibilitat d'afegir-ne d'altres per al seu propi ús.

A la següent taula es llisten tots els codis d'estat que contempla el JepiFW.

Codi	Missatge	RFC
100	Continue	
101	Switching Protocols	
102	Processing	RFC2518
200	OK	
201	Created	
202	Accepted	
203	Non-Authoritative Information	
204	No Content	
205	Reset Content	
206	Partial Content	
207	Multi-Status	RFC4918
208	Already Reported	RFC5842
226	IM Used	RFC3229
300	Multiple Choices	
301	Moved Permanently	
302	Found	
303	See Other	
304	Not Modified	
305	Use Proxy	
307	Temporary Redirect	
308	Permanent Redirect	RFC7238
400	Bad Request	
401	Unauthorized	
402	Payment Required	
403	Forbidden	
404	Not Found	
405	Method Not Allowed	
406	Not Acceptable	
407	Proxy Authentication Required	
408	Request Timeout	
409	Conflict	
410	Gone	
411	Length Required	
412	Precondition Failed	
413	Payload Too Large	
414	URI Too Long	
415	Unsupported Media Type	
416	Range Not Satisfiable	
417	Expectation Failed	
422	Unprocessable Entity	RFC4918
423	Locked	RFC4918
424	Failed Dependency	RFC4918
425	Reserved for WebDAV advanced collections expired proposal	RFC2817
426	Upgrade Required	RFC2817
428	Precondition Required	RFC6585
429	Too Many Requests	RFC6585
431	Request Header Fields Too Large	RFC6585
500	Internal Server Error	

501	Not Implemented	
502	Bad Gateway	
503	Service Unavailable	
504	Gateway Timeout	
505	HTTP Version Not Supported	
506	Variant Also Negotiates (Experimental)	RFC2295
507	Insufficient Storage	RFC4918
508	Loop Detected	RFC5842
510	Not Extended	RFC2774
511	Network Authentication Required	RFC6585
612	Unexpected error	

Taula 1 Missatges d'error i excepcions reconegudes pel JepiFW

7.8 Models

Els models resulten una part indispensable de qualsevol *framework* ja que són els encarregats d'accedir a la base de dades per recuperar la informació que hi ha guardada, afegir-ne de nova, actualitzar-la o eliminar-la.

En el nostre cas, la estructura del component que gestiona els models és molt senzilla, ja que únicament consta de la classe ModelInterface, que és l'abstracció del model que ens permetrà treballar amb diversos gestors de base de dades. Per ara, però, només s'ha considerat MySQL com a sistema gestor de base de dades. Així que per ara només tenim la classe MySqlModel, que implementa la interfície ModelInterface, i a la vegada serà heretada pels models de les aplicacions desenvolupades amb el *framework*.

La classe MySqlModel rebrà una classe de tipus Connections que conté totes les connexions a base de dades que s'han obert. Per establir una connexió nova només cal cridar el mètode `openMySqlConnection` de la instància de la classe Connections, passant-li per paràmetre el nom de la base de dades que està definida al fitxer de configuració. I després perquè els models de l'aplicació utilitzin la connexió que toca, han de definir una variable de tipus *protected* anomenada `$dbConnection`. D'aquesta manera, la classe MySqlModel (classe pare) sabrà quina és la connexió que s'ha d'utilitzar en aquest model en qüestió.

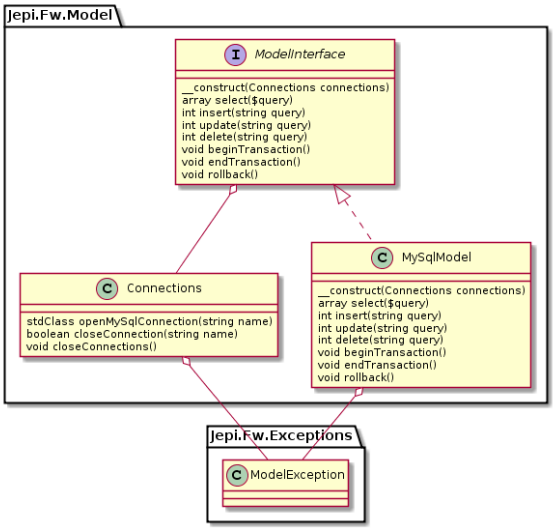

Figura 24 Diagrama de classes del component Jepi.Fw.Model

Les connexions a base de dades s'inicialitzen utilitzant la llibreria del sistema PDO, vegem com utilitza la classe Connections aquest objecte:

```php
/**
 * Open connection if is not already open.
 *
 * @param string $name optional parameter if nothing is send it gets
 * the default connection.
 * @return \PDO
 * @throws ModelException
 */

public function openMySqlConnection($name = null) {
    $defaultConnection = $this->config->get('Database', 'defaultConnection');
    //If $name doesn't exists, get the default database from config.ini
    if (is_null($name)){
        $name = $defaultConnection;
    }
    //If connection already exists, return it
    foreach ($this->connections as $connection) {
        if($connection->name == $name && !is_null($connection->link)) {
            return $connection->link;
        }
    }
    //Open new connection
    try {
        //Get the configuration of the connection.
        $host = $this->config->get('Database', 'host');
        $port = $this->config->get('Database', 'port');
        $user = $this->config->get('Database', 'user');
        $pass = $this->config->get('Database', 'pass');
        $db = $this->config->get('Database', 'name');

        //Create the new connection.
        $connection = new \PDO($connData = 'mysql:host=' . $host[$name] . ';port=' . $port[$name] . ';dbname=' . $db[$name], $user[$name], $pass[$name],
            array(\PDO::MYSQL_ATTR_FOUND_ROWS => true, \PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8" ));
        $connection->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);

        //Add connection link to connections array
        $this->connections[] = (object)array('name'=>$name,'link'=>$connection);

        return $connection;
    } catch (\Exception $e) {
        throw new ModelException("Error trying to connect to database [$name]");
    }
}
```

Figura 25 Funció de la classe Connections que inicialitza una connexió a base de dades

El què es fa és, primerament, comprovar si s'ha proveït el paràmetre `$name` corresponent al nom de la connexió a base de dades. En cas que no s'hagi passat, el recupera de les variables de configuració. Tot seguit es comprova si ja hi ha una connexió oberta amb el mateix nom, i en cas d'haver-li, en retorna el link. En cas que no hi hagi cap connexió amb el nom obtingut, es van a buscar les variables de configuració relatives a la connexió a base de dades i s'obra una connexió mitjançant l'objecte PDO utilitzant aquestes les variables corresponents a la connexió el nom de la qual el tenim a la variable `$name`. Una vegada oberta la connexió, la guardem dins de l'array de connexions del propi objecte Connections.

Per a treballar amb múltiples connexions, l'arxiu config.ini necessita tenir les connexions a base de dades definides de la següent manera:

```ini
[Database]
defaultConnection = default

host[default] = localhost
port[default] = 3306
user[default] = foo
pass[default] = foobar
name[default] = foo

host[todo] = localhost
port[todo] = 3306
user[todo] = todo
pass[todo] = CaMFAsyzeb9D7nyL
name[todo] = todo
```

Figura 26 Definició de connexions a base de dades al config.ini

7.9 Vistes

Les vistes esdevenen també una part fonamental, juntament amb els models i els controladors, del *framework* objectiu que detalla aquest document. En el cas que ens ocupa, l'objecte que gestiona les vistes és una classe anomenada View, que implementa una interfície la qual defineix únicament dos mètodes.

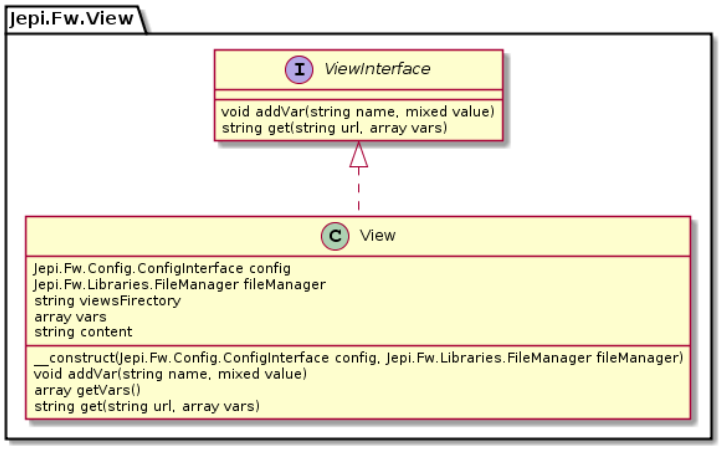

Figura 27 Diagrama de classes del component Jepi.Fw.View

La lògica continguda a la classe View ens permet gestionar documents que seran retornats des dels nostres controladors, per renderitzar-se posteriorment com a resultat de l'execució. Dita lògica, per tant, és molt concreta ja que l'únic que cal fer és executar un arxiu, tot i què prèviament cal definir la informació que serà compartida entre el punt del codi en el qual ens trobem i la vista que es vol mostrar. Això es fa a través del mètode `addVar`, que afegeix claus i valors a un diccionari que conté les variables que voldrem enviar a la vista. Aleshores al moment previ de llegir la vista, es creen les variables en un punt del codi en què siguin accessibles per l'*script* en qüestió que s'inclourà com a vista.

```
public function addVar($name, $value)
{
    $this->vars[$name] = $value;
}
```

Figura 28 Mètode per definir variables per a les vistes

El mètode per recuperar una vista és el mètode `get`, que rep dos paràmetres que són `$url` i `$vars`, i aquests serveixen, el primer per indicar la ruta del fitxer dins el JepiFW on es troba l'arxiu en qüestió que constitueix la vista que es vol mostrar. El segon paràmetre és el que ens permet definir les variables que es podran llegir desde la vista. Aquestes variables són afegides a la llista de variables de què ja disposàvem i s'extreuran dins la funció abans d'incloure el document de la vista.

No oblidem que les variables les tenim en un diccionari, per tant és necessari traduir aquest diccionari a un conjunt de variables que estiguin definides com a tal. Això és exactament el què fa la funció `extract`, que és una funció de PHP. Aquesta funció s'encarrega de definir les variables com

si estiguessin declarades pel programador, però aquestes provenen d'un array les claus del qual són el nom de les variables que es definiran.

A més, també utilitzem una altra funció de PHP, que serveix per netejar els *buffers* i retornar el què contenen a una variable. Això s'utilitza per capturar el contingut de la vista per tal que se'n pugui donar l'ús que es cregui convenient. Si bé és cert que es podria retornar el resultat directament a pantalla directament (fent `print` o `echo`), donat que el *framework* segueix una estructura clara i tota la informació de retorn es gestiona des de l'entitat Response, aquesta información s'ha de retornar i no escriure per pantalla, i per això utilitzem la funció de PHP `ob_get_clean;`

```
public function get($url, $vars = array())
{
    $this->vars = array_merge($this->vars, $vars);

    ob_start();
    extract($this->vars);

    include APP_ROOT . $this->viewsDirectory . DIRECTORY_SEPARATOR . $url;

    $this->content = ob_get_clean();
    return $this->content;
}
```

Figura 29 Mètode per recuperar una vista

7.10 Controladors

Els controladors constitueixen el darrer component imprescindible per a un *framework* que segueixi el patró MVC. És on organitzaran tota la lògica de l'aplicació els desenvolupadors que vulguin treballar amb el JepiFW. L'objectiu que es pretén assolir en els controladors és el fet de tenir disponible tota la informació possible de l'execució, ja sigui la informació procedent del client com la informació de l'execució en si mateixa. Així doncs, s'ha de preveure que el *Controller* pugui tenir accés a totes les classes que componen el *framework.* Això ho podem aconseguir gràcies al component DI (*Dependency Injection*) que hem utilitzat, del qual es parlarà amb més detall a l'apartat de llibreries externes.

Entrant en el detall del què és el component Jepi.Fw.Controller, podem observar com conté una interfície de la classe Controller, de la mateixa manera que la resta de components, que tenen abstraccions de les estructures per les entitats principals. Veiem, però, que aquesta interfície no té cap mètode definit. Això és perquè de moment no s'ha cregut oportú definir-ne cap, tot i que s'ha volgut mantenir la mateixa lògica que a la resta de components, i per això s'ha definit l'abstracció, tot i que buida. Anant un pas més enllà, si ens fixem en la classe Controller, tampoc té cap mètode definit, sinó únicament propietats; perquè els mètodes dels controladors que es defineixin com a públics seran accessibles públicament desde la web, per com funciona el mòdul d'enrutament, i per tant no ens interessa definir cap ruta accessible ja que això interferiria en el desenvolupament que es dugués a terme amb el JepiFW.

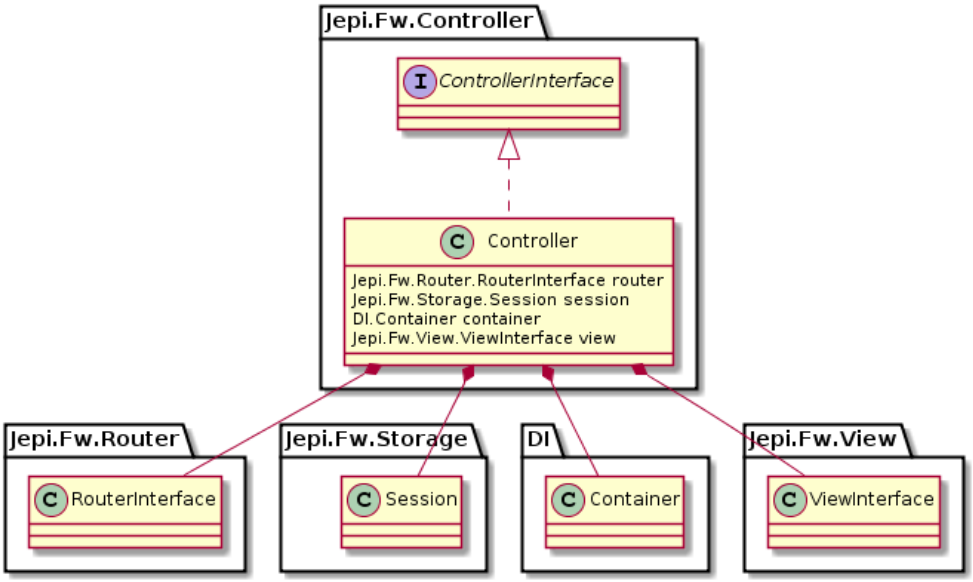

Figura 30 Diagrama de classes del component Jepi.Fw.Controller

La funció del controlador no és altra que enviar comandes als models per gestionar la base de dades, i posteriorment enviar comandes a les vistes associades per actualitzar la presentació i mostrar-la acord amb les dades que retornen els models.

En el patró de disseny MVC, que queda completat ara que hem definit els controladors, tenim tres actors principals, que són els Models, les Vistes i els Controladors, que estan relacionats entre ells tal com es mostra a la següent figura 28. L'usuari visualitza una vista, i realitza un acció sobre d'aquesta. Aquesta acció es tradueix en una crida a un controlador, que manipularà un seguit de models per tal d'actualitzar la informació concreta que aquests gestionin, i així després actualitzar la vista perquè l'usuari pugui realitzar-hi noves accions.

Justifiquem així, doncs, la necessitat del controlador de disposar de tota la informació possible per a poder dur a terme les tasques que s'hagin d'executar en conseqüència de les accions dels usuaris. Per tal de poder garantir això, necessitem poder accedir a totes les classes que componen el *framework*, i aquesta és tasca del mòdul de DI amb què està relacionat a través de la classe Container. El Container ens permet instanciar qualsevol classe que estigui indexada dins del gestor de dependències, evitant haver de passar totes les classes que es puguin necessitar al constructor de la classe Controller, ja que evidentment volem que aquestes instàncies compartides siguin úniques a tota l'execució i tots els components del *framework* i per tant no es poden instanciar dins sinó que s'han de recuperar d'un espai comú, que en el nostre cas és el contenidor de la llibreria PHP-DI.

Tot seguit es mostra com interactuen els diversos components del *framework* a partir d'una acció realitzada per un usuari:

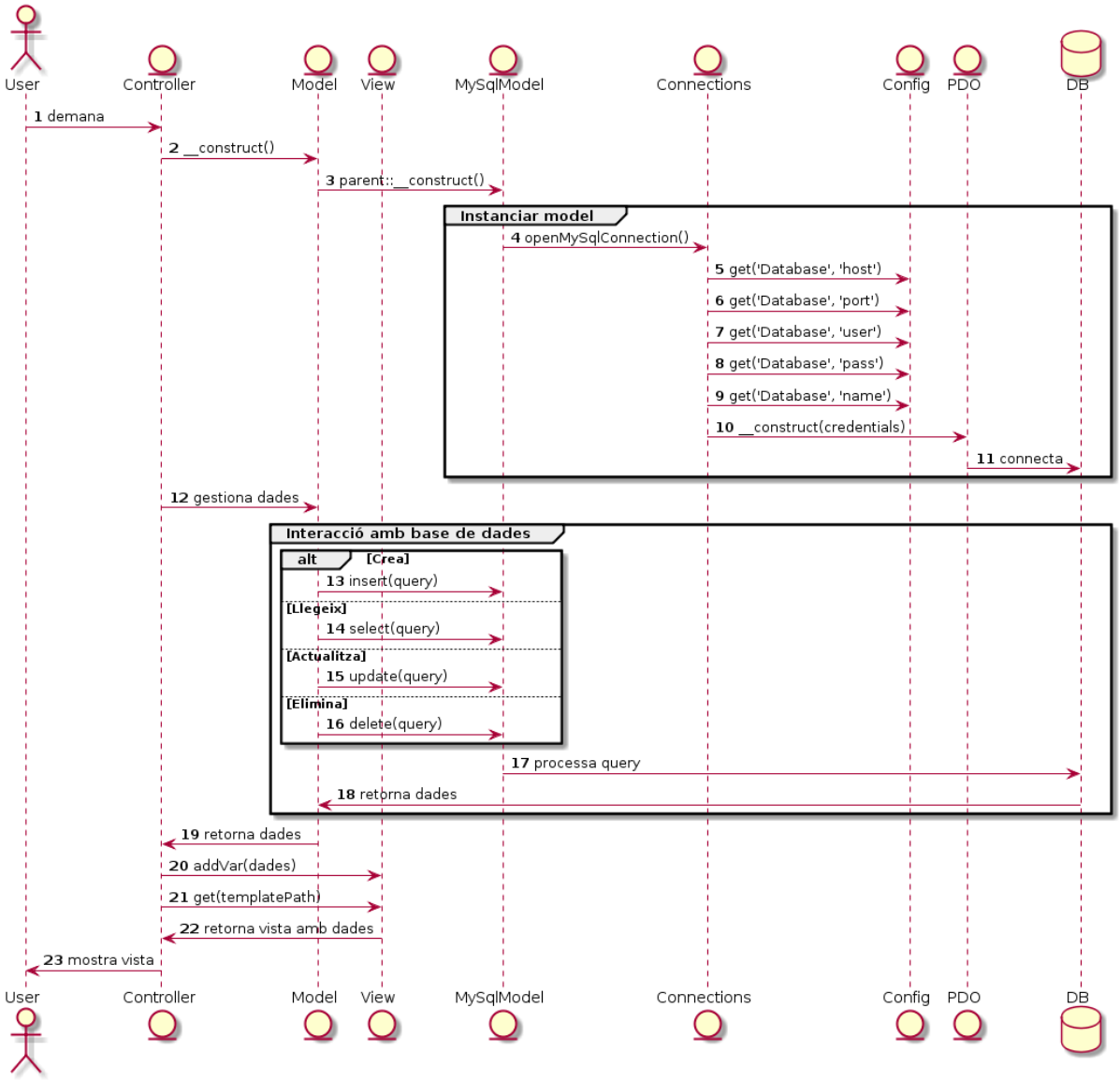

Figura 31 Interacció dels elements principals (controladors, models i vistes)

Es pot observar com un usuari fa una petició al controlador i aquest instancia els models amb què ha d'interactuar per actualitzar les dades. Aquests a la vegada extenen la classe MySqlModel, que mitjançant la classe Connections obra una connexió a la base de dades utilitzant una instància de la classe PDO, amb les credencials obtingudes amb l'objecte Config.

Una vegada es disposa dels models instanciats correctament, des del controlador ja es pot fer ús d'aquests per manipular les dades com es desitgi i, posteriorment, enviar-les a l'objecte View perquè quan el controlador li demani una vista, les hi incrusti prèviament a ser retornada com a resposta a la petició que ha realitzat l'usuari.

7.11 Llibreries

Podem agrupar les classes del *framework* en dos grups, les que estan intrínsecament relacionades amb el funcionament del mateix o bé les que són entitats externes al nucli i que, per tant, són més prescindibles. Això no vol dir que el *framework* no les necessiti per a funcionar, sinó que per si soles ja constitueixen una funcionalitat completa que pot utilitzar-se en molts altres llocs, sense necessitat del *framework*. Si pensem en una llibreria per a integrar-se amb qualsevol servei de tercers, com

podria ser, per exemple, el servei d'enviament d'emails Mailchimp; dita llibreria es connectaria a través d'una API probablement, i serviria d'interfície per a la comunicació entre una aplicació qualsevol i el servei en qüestió. Per tant, la llibreria pot funcionar perfectament sense necessitat de l'entorn on s'està utilitzant, i de la mateixa manera, aquest component Jepi.Fw.Libraries està pensat per incloure les funcionalitats que es necessiten en el *framework* perquè aquest funcioni sense problemes, però que a la vegada per si soles podrien utilitzar-se fora del mateix.

Veiem que per ara només hi ha una classe dins d'aquest component, que és la classe FileManager, que ens serveix per interactuar amb el sistema de carpetes i fitxers per tal de saber quins fitxers conté un directori o bé per conèixer quants i quins nivells té una carpeta concreta.

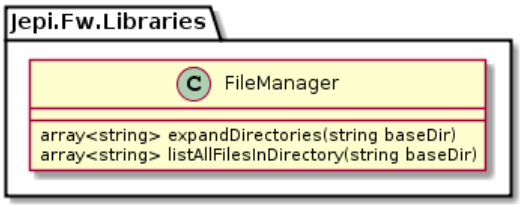

Figura 32 Diagrama de classes del component Jepi.Fw.Libraries

La classe FileManager s'utilitza dins la classe FrontController per a carregar tots els fitxers de configuració que es trobin dins la carpeta config, vegem-ho:

```
private function loadConfigFiles() {
    $this->config->loadFile(SYSTEM_ROOT . DIRECTORY_SEPARATOR . 'config.ini');
    $configFiles = $this->fileManager->listAllFilesInDirectory(APP_ROOT . DIRECTORY_SEPARATOR . 'Config');

    foreach ($configFiles as $configFile) {
        $this->config->loadFile($configFile);
    }
}
```

Figura 33 Mètode de la classe FrontController on s'utilitza la classe FileManager per llistar els fitxers d'un directori

Per defecte el FrontController carrega l'arxiu config.ini que està dins la carpeta System, i després procedeix a llistar tots els fitxers que hi ha dins la carpeta Config de l'aplicació. Aquí és on intervé la classe FileManager, que actua a mode d'interfície entre el *framework* i el sistema d'arxius del sistema i retorna un array amb les rutes corresponents a tots els fitxers dins el directori App/Config. Aleshores es carreguen tots a la classe Config perquè se n'actualitzin els paràmetres de configuració.

7.12 Excepcions

Finalment, el darrer mòdul o component que queda per definir és el que s'encarrega d'organitzar les excepcions. En el cas que ens ocupa, donat que hi ha diversos components interactuant entre ells, amb només una classe per a representar les excepcions, hauria quedat desorganitzat i massa generalitzat el concepte de gestió d'excepcions. Així que s'ha optat per a crear una excepció per a cada pas del processat d'una petició rebuda pel *framework*. D'aquesta manera, tenim una excepció per al paquet de configuració, una per al de models, una altra per al d'enrutament, una altra per a la gestió de dades (*cookies* i sessió), i una altra per al paquet de IO (*Input/Output*).

Veiem que totes les classes d'excepció extenen la classe base JepiException, que a la vegada exten la classe base Exception de PHP. Per tant, totes les excepcions del JepiFW tenen accés tant als mètodes de Exception com als tres mètodes que defineix JepiException:

- getProductionMessage: configura el missatge que es retornarà si l'entorn està configurat com a entorn de producció.
- getDevelopmentMessage: configura el missatge que es retornarà si l'entorn està configurat com a entorn de desenvolupament.
- getExceptionType: retorna la propietat *protected* `exceptionType`, que cada classe del paquet defineix diferent i per tant ens permet conèixer quin component ha generat l'error sense necessitat de saber la classe concreta d'excepció que ha generat l'error.

A més, veiem que la classe JepiException té dependència de l'objecte Response, i és que aquest, com s'explica a la descripció del paquet Jepi.Fw.IO, conté els diferents missatges de retorn de la petició HTTP que el *framework* pot retornar a les capçaleres, incloent els missatges pertinents als errors indexats segons codi de l'excepció.

Si s'executa una aplicació desenvolupada amb el JepiFW en un entorn de desenvolupament, el missatge d'error que retornarà l'excepció és un missatge que detallarà el fitxer que ha generat l'error, la línia concreta on aquest s'ha generat i el missatge concret que ha retornat l'execució fallida. Però si l'entorn és de producció, per tal de protegir el codi de l'aplicació, no es retornarà aquesta informació detallada relativa a la programació sinó un missatge d'error estàndard corresponent al codi d'excepció que s'hagi rebut (veure Taula 1 Missatges d'error gestionats pel JepiFW), mantenint d'aquesta manera l'estructura de l'aplicació invisible de cara als usuaris.

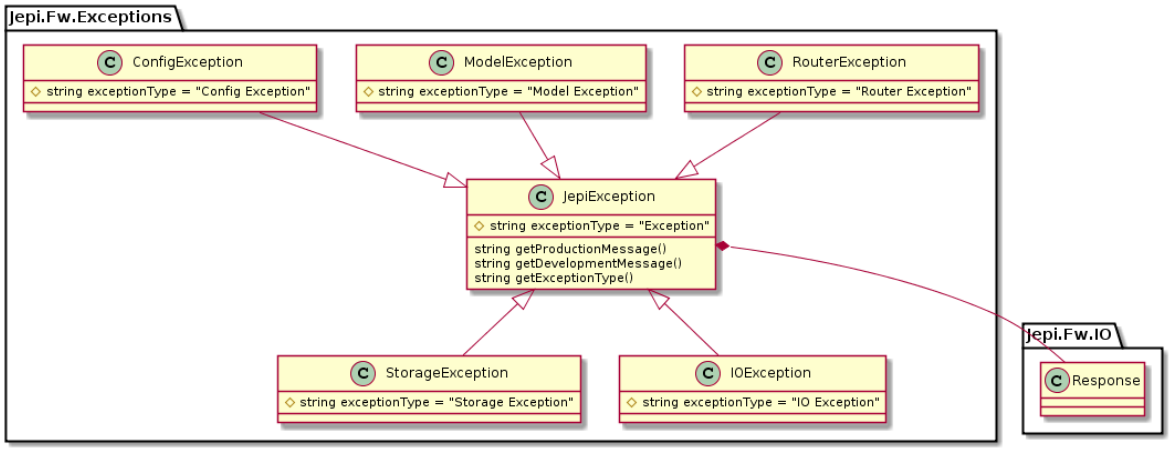

Figura 34 Diagrama de classes del component Jepi.Fw.Exceptions

La gestió d'excepcions es duu a terme dins del FrontController, que és qui captura les excepcions i s'encarrega de crear la resposta amb la informació que aquestes contenen (per més detalls, veure la Figura 12 Funció per gestionar la captura d'excepcions a FrontController).

8. Llibreries externes

Aquest *framework* s'ha nodrit de dues eines ja existents, que han estat integrades al nucli del mateix, i que en condicionen el funcionament. Aquestes dues llibreries són composer i PHP-DI, i a més, de composer se'n desprèn una altra que és el motor que fa funcionar composer, però cal tractar-la per separat, és l'autoloader.

8.1 Composer

Composer és una eina per a gestionar les dependències en PHP, permetent declarar les llibreries amb un format estàndard, de tal manera que cada llibreria consisteix en una línia dins d'un fitxer anomenat composer.json, i després mitjançant la comanda `composer update` l'aplicació actualitzarà totes les llibreries externes que estan declarades.

El què antigament s'havia de fer era anar a la web de cada una de les llibreries que volies utilitzar en el teu codi, on algunes de les pàgines web demanaven registrar-se; actualment es fa mitjançant composer, que a més també permet fer un manteniment de les mateixes d'una forma excel·lent. I un dels altres problemes que composer soluciona és el fet que abans quan s'actualitzava una llibreria la tasca d'actualitzar-la al teu codi era manual, fet que ha canviat, ja que ara pots actualitzar totes les llibreries amb una sola comanda.

I ara pensem a nivell d'escalabilitat, què pot implicar l'ús de composer? Si ens imaginem múltiples nivells de dependència, abans resultava costós d'organitzar l'arbre de fitxers de les llibreries externes perquè cadascuna portava dins tots els fitxers necessaris per al seu correcte funcionament, i per tant si dues llibreries utilitzaven una tercera, aquesta sovint apareixia duplicada i possiblement utilitzant versions diferents. Això a nivell de manteniment de codi és d'una eficiència pèssima, motiu de més per utilitzar un gestor de dependències de llibreries, que situa totes les llibreries al mateix nivell i amb la mateixa estructura. Totes les llibreries estan dins de la seva corresponent carpeta i totes elles dins la carpeta /vendor, que és la que aglutina les dependències del codi. I d'aquesta manera, en tot moment ets molt conscient de totes les dependències que té el teu codi, a més que t'assegures que totes elles estan utilitzant la última versió i a la vegada aquestes també utilitzen la última versió de les seves dependències, perquè al final és una instal·lació recurrent, les dependències de les teves dependències s'ubiquen a la carpeta /vendor com si fossin dependències directes del teu codi.

El fitxer composer.json conté tot un seguit de paràmetres de configuració que permeten declarar les propietats del projecte, a part de les dependències, per tal que aquest quedi definit amb el mateix format que la resta de paquets que administra composer. De tal manera que el paquet es pot pujar a Packagist (repositori de paquets de composer) i una vegada fet això, ja tothom podrà utilitzar el teu paquet com a dependència d'altres aplicacions.

En el cas del JepiFW s'ha utilitzat composer per gestionar la dependència de PHP-DI, llibreria que s'explicarà tot seguit. Aquesta a la vegada té múltiples dependències, i per poder utilitzar alguna funcionalitat avançada de la mateixa ens ha calgut afegir-ne encara més. Quedant-nos el fitxer composer.json de la següent manera:

Figura 35 Contingut del fitxer composer.json

Veiem que primerament es defineix el nom del paquet, que és el què s'hauria d'utilitzar per a crear un paquet que utilitzi el JepiFW com a dependència. Tot seguit trobem una descripció del paquet, la llicència que regeix el codi i els autors del mateix, la mínima versió estable del paquet; i més avall veiem dues seccions destinades a les dependències, una genèrica i una explícita per l'entorn de desenvolupament. Aquesta divisió ens permet afegir dependències necessàries per al desenvolupament, però que no s'utilitzen en l'entorn de producció. En el nostre cas s'ha afegit com a dependència en l'entorn de desenvolupament la versió 5.0 de phpunit, i un commit concret fet a la branca dev-master de la llibreria php-code-coverage, que ens serveix per poder analitzar el percentatge de cobertura de codi que donen els tests

I pel què fa a les dependències de producció, tenim la llibreria PHP-DI que per utilitzar unes funcionalitats concretes que d'aquesta ens demana afegir doctrine/annotations i ocramius/proxy-manager. I la darrera dependència és un motor de plantilles anomenat Twig que s'ha afegit perquè a la propera versió del *framework* es vol integrar aquest motor de plantilles com a sistema per defecte de gestió de vistes (per més informació, consultar l'apartat de Projecció a Futur).

Un cop instal·lades les dependències que s'acaben de descriure, podem observar com aquestes a la vegada també en tenen moltes d'altres, ja que el contingut de la carpeta vendor queda de la següent manera:

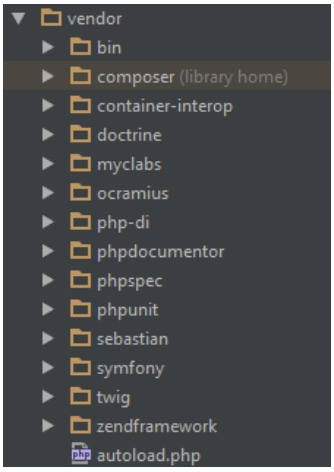

Figura 36 Carpeta vendor amb totes les dependències instal·lades per composer

Veiem com algunes de les carpetes es corresponen al paquet que hem definit a les dependències però moltes d'altres no, i aquestes són les que es podrien considerar dependències de segon nivell.

Per veure el detall de les diverses versions del JepiFW que es troben indexades al repositori de paquets de composer Packagist, cal dirigir-se al següent enllaç:

https://packagist.org/packages/jepi/fw

8.2 Autoloader

Al fitxer composer.json hi ha una secció que es diu autoload. Aquesta secció és la que permet definir quins són els paràmetres en què es basarà composer per dur a terme la càrrega de fitxers durant l'execució de l'aplicació. És imprescindible que se li indiqui on es troben les classes que ha de carregar, i això es fa mitjançant un protocol que pot ser PSR-0 o PSR-4. En el nostre cas hem escollit PSR-4 perquè és més modern i donat que composer ha definit com a objectiu que es migrin tots els paquets actuals a PSR-4, resultava absurd plantejar-se l'alternativa.

El protocol de suport a la càrrega automàtica de classes funciona com un traductor, si l'*autoloader* detecta que es demana una classe (que pertany a un espai de treball concret), i encara no s'ha importat l'arxiu que la conté, mirarà a quina ruta es troba a partir d'analitzar el *namespace* i comparar-lo amb els *namespaces* definits com a claus dins la definició del composer.json. Aleshores mirarà si hi ha un fitxer anomenat com la classe en qüestió a cadascuna de les rutes que inclou l'array associat al *namespace*. Un cop trobat el fitxer, l'inclourà i ja podrà ser instanciat.

Aquesta llibreria és extremadament útil perquè ens permet oblidar-nos d'utilitzar les funcions include, require, include_once, require_once, que s'utilitzaven abans. L'autoload afegeix un nivell d'abstracció i ens dóna la possibilitat de obviar la inclusió de fitxers al projecte en qualsevol punt de l'execució, ja que només ens cal incloure aquest fitxer i prou, ell ja farà la resta.

8.3 PHP-DI

Aquesta llibreria implementa un patró de disseny de tipus estructural anomenat Dependency Injection (d'aquí el DI del nom), el qual serveix per implementar una arquitectura amb acoblament flexible amb la finalitat d'obtenir un codi més testejable i fàcil de mantenir i extendre.

La idea de la injecció de dependències és que cada objecte en el constructor defineixi quins són els objectes de què depèn, per tal que aquests li siguin enviat al constructor. Si un objecte A depèn d'un objecte B, si instanciem l'objecte B dins l'objecte A estarem augmentant l'acoblament del nostre codi; en canvi si passem l'objecte B com a paràmetre al constructor de l'objecte A, aleshores estem desacoblant les dues classes tot i mantenir la dependència, a més no ens importa de on vingui l'objecte, simplement que contingui els mètodes que se sap que té l'objecte B. Aquest és un dels motius pels quals el JepiFW utilitza tantes interfícies, perquè les dependències entre classes s'han definit seguint aquesta norma, i així, no ens preocupem per la implementació concreta sinó només per la definició.

La idea del Dependency Injection prové dels principis de la inversió del control (IoC), que és un dels mecanismes de l'enginyeria del software que s'utilitza per a incrementar la modularitat d'un programa per tal de fer-lo més escalable, i les principals aplicacions s'ubiquen a la programació orientada a objectes així com a alguns altres paradigmes de programació. L'IoC avarca els següents propòsits de disseny:

- Desacoplament entre l'execució d'una tasca i la seva implementació.
- Es focalitza en la tasca per la qual està dissenyat un mòdul.
- No creu en assumpcions del què farà un codi, sinó que es beneficia dels contractes entre objectes (l'ús d'interfícies per dependències el podem entendre com a un compromís ferm dels propòsits de la classe).
- Prevé els *side effects* [7]al reemplaçar un mòdul per un altre.

Així doncs, seguint amb aquesta idea, s'ha utilitzat la injecció de dependències en el constructor com a sistema per a reduir l'acoblament. Però fins aquí només s'ha definit el perquè de l'ús del patró Dependency Injection, però encara no s'ha justificat el perquè utilitzar la llibreria PHP-DI. Bé doncs, la llibreria intervé en el moment en què volem instanciar una classe. Si volem instanciar la classe FrontController, per exemple, caldria instanciar prèviament les classes que aquesta requereix en el constructor.

[7] Es coneix com a *side effect* aquells errors que apareixen després de modificar un codi, per culpa de què a algun lloc s'està confiant amb què l'estat del codi és un, sense tenir en compte que els canvis han pogut alterar altres propietats. Si no hi ha un control molt atòmic dels propòsits concrets d'una funció, aquests canvis poden suposar un risc important.

```php
/**
 * @param ConfigInterface $config
 * @param RequestInterface $request
 * @param Container $container
 * @param FileManager $fileManager
 */
public function __construct(ConfigInterface $config,
        RequestInterface $request, Container $container,
        FileManager $fileManager) {
    $this->config = $config;
    $this->request = $request;
    $this->container = $container;
    $this->fileManager = $fileManager;
    $this->loadConfigFiles();
    $this->initErrorManagement();
}
```

Figura 37 Constructor de la classe FrontController

És en aquest punt on intervé la llibreria PHP-DI. Si observem el fitxer index.php (Figura 9 Fitxer index.php del JepiFW), veurem com s'instancia la classe FrontController sense necessitat de passar-li cap dependència com a paràmetre. Això és gràcies a què s'ha utilitzat el contenidor d'instàncies que gestionar la llibreria PHP-DI. Aquest contenidor el què fa és buscar l'objecte que se li està demanant i en cas que ja el tingui instanciat el retorna i en cas que no, el prova d'instanciar. Si en el constructor necessita objectes, els busca dins del contenidor i repeteix el procés, de manera que anirà instanciant una a una totes les dependències de totes les classes que requereixi la classe que se li està demanant i les que a la vegada requereixin les dependències d'aquest.

Composer i PHP-DI són molt semblants pel què fa a la manera que tenen de gestionar les dependències, la diferència està en què composer ho fa a nivell de llibreria i PHP-DI ho fa a nivell de classe. Sempre es mira si tens el què necessites, en cas que no ho tinguis, ho vas a buscar (o ho crees en el cas de PHP-DI), i si en procés de creació necessites més coses, repeteixes el procés.

I seguint amb l'exemple de l'obtenció d'una instància de la classe FrontController, s'entén que un cop el contenidor ens retorni l'objecte FrontController, al contenidor a part d'aquest objecte també contindrà el propi contenidor, i una instància de cadascuna de les dependències que determina el constructor. Però precisament en aquest cas, veiem com les dependències no són objectes instanciables sinó interfícies, que com sabem, no es poden instanciar perquè no són classes finals. I en aquest punt és on intervé el fitxer de configuració di.php. Aquest fitxer conté la definició de les diferents abstraccions, per tal que quan el contenidor es trobi amb una dependència que és una instància, sàpiga quina és la classe final que s'ha d'instanciar.

```php
return [
    \Jepi\Fw\Config\ConfigInterface::class => DI\object(\Jepi\Fw\Config\Config::class),
    \Jepi\Fw\IO\InputInterface::class => DI\object(\Jepi\Fw\IO\Input::class),
    \Jepi\Fw\IO\RequestInterface::class => DI\object(\Jepi\Fw\IO\Request::class)->lazy(),
    \Jepi\Fw\Router\RouterInterface::class => DI\object(\Jepi\Fw\Router\Router::class),
    \Jepi\Fw\Model\ModelInterface::class => DI\object(\Jepi\Fw\Model\MySqlModel::class),
    \Jepi\Fw\View\ViewInterface::class => DI\object(\Jepi\Fw\View\View::class)
];
```

Figura 38 Definició de les abstraccions al fitxer di.php

Veiem com aquest fitxer centralitza totes les interfícies i en defineix la classe final que s'haurà d'instanciar per a resoldre les dependències de la mateixa. En cas que un desenvolupador desitgés utilitzar una implementació diferent de la que el *framework* proposa, caldria crear la classe nova

assegurant que implementa la interfície en qüestió del mòdul que es vol extendre o sobreescriure, i després modificar l'equivalència de la interfície a la classe que es defineix a aquest fitxer.

A més, la llibreria PHP-DI té algunes particularitats interessants pel què fa a la injecció d'instàncies que van més enllà de la resolució de dependències al moment de construir els objectes. Una d'elles és la creació d'instàncies mandroses i l'altra és la possibilitat d'injectar variables mitjançant anotacions als comentaris del codi.

El primer dels avantatges que ofereix PHP-DI és el fet de instanciar objectes de forma mandrosa, que vol dir que ens retorna una instància d'un objecte que diu ser de la classe esperada però fins que no se'n crida cap mètode aquest no és instanciat realment. El què es retorna en realitat és un objecte que actua a mode de *proxy* de l'objecte, és a dir que és idèntic a l'objecte original en estructura i funcionalitat de tal manera que no es pot notar la diferència, però no és fins que no actues sobre aquest que no s'instancia l'objecte real, de tal manera que és el *proxy* qui instancia l'objecte abans de cridar-ne el mètode concret. Vegem que a l'anterior figura (Figura 38 Definició de les abstraccions al fitxer di.php) hem definit la classe Request per tal que s'instanciï de forma mandrosa. Per a poder fer servir aquesta funcionalitat s'ha hagut d'afegir la dependència a la llibreria ocramius/proxy-manager, com es pot veure a l'arxiu composer.json.

I finalment, l'altre avantatge que ofereix PHP-DI és un dels motius principals pels quals s'ha escollit aquesta llibreria per a dur a implementar la injecció de dependències, i és el fet de poder utilitzar anotacions per a injectar objectes al codi. Per a utilitzar aquesta funcionalitat ha calgut afegir la dependència a la llibreria doctrine/annotations però és realment útil poder incrustar objectes a la definició d'una classe sense haver-te de preocupar ni de construir-los ni de gestionar les dependències que aquests tenen. El funcionament és ben senzill. Únicament cal afegir una anotació amb el tipus d'una variable i afegir una anotació @Inject, com podem observar al següent fragment de codi:

```php
/**
 * @Inject
 * @var \App\Models\TaskModel
 */
private $model;

public function add($id_list, $name, $description = "") {
    $id_task = $this->model->createTask($id_list, $name, $description, 0);
    return $this->model->getTask($id_task);
}
```

Figura 39 Exemple de injecció de dependències mitjançant anotacions

La imatge anterior és un abstracte del codi d'un controlador, en el qual veiem com es declara la variable `$model` i dins dels mètodes del controlador s'utilitza sense preocupar-se de instanciar-la ni tant sols de si té dependències. Així doncs, és bastant evident que aquesta funcionalitat pot ajudar molt alhora de desenvolupar, ja que et redueix dràsticament la quantitat de línies de codi, i a més et redueix la quantitat d'objectes instanciats. Si és necessari gestionar múltiples instàncies a un objecte determinat, es farà mitjançant la creació d'instàncies clàssica, i si són elements que només cal que s'instanciïn una vegada podem fer ús d'aquesta funcionalitat que ja suporta el JepiFW.

Aquesta funcionalitat, però, té l'inconvenient de la penalització en quan a rendiment que porta associada. El fet d'haver d'interpretar les anotacions a cada execució és quelcom costós, tot i que en entorns de producció, on el codi no canviarà sovint, es poden utilitzar sistemes de *cache* que compensen aquests *delays* i incrementen el rendiment i velocitat d'execució gràcies a guardar en memòria gran part de la informació de les anotacions, per tal de poder donar respostes molt més immediates. I sigui com sigui, és important assegurar-nos que únicament utilitzem la injecció a través d'anotacions en els controladors.

Finalment, cal establir també, les pautes bàsiques per a la utilització d'aquesta funcionalitat, ja que és molt útil però també pot generar dependències de la pròpia llibreria PHP-DI a la vegada que per altres mitjans estem intentant reduir tant com podem el grau d'acoblament del codi. Per tant és important establir les bases del què és una bona pràctica i el què no, pel què fa a la injecció de dependències. Aquestes normes són extretes de la documentació de la llibreria i s'ha cregut convenient citar-les explícitament:

1. No s'han d'agafar mai les instàncies directament del contenidor des de dins d'una classe. Sempre s'ha d'utilitzar la injecció de dependències (és a dir definir les dependències en el constructor).
2. De forma general, escriure codi desacoblat del contenidor.
3. Al codi, definir les dependències a partir d'interfícies, i gestionar quines implementacions cal utilitzar a l'arxiu de configuració del contenidor.

En el cas de Symfony, la primera norma no es compleix ja que tots els controladors tenen el contenidor a disposició i s'utilitza precisament per instanciar els diversos objectes que s'hi necessiten. A la documentació de PHP-DI recomanen utilitzar la injecció a través de comentaris en els controladors perquè d'aquesta manera no s'estableix cap contracte entre la llibreria i el codi del controlador, enlloc es concreta que el codi necessiti aquesta propietat instanciada. A més, a l'utilitzar les anotacions per definir les dependències a injectar, la classe és independent del contenidor, com es recomana fer a la norma 2.

9. Perfils d'usuari

El llenguatge PHP es situa setè[8] en el *ranking* de llenguatges més utilitzats, segons IEEE Spectrum. Aquest *ranking* està elaborat a partir de les dades extretes de 10 fonts diferents (d'entre les quals es troba el servei de control de versions GitHub), i que permeten obtenir la popularitat dels 48 llenguatges utilitzats amb més freqüència. D'aquestes diverses fonts se n'obtenen 12 mètriques diferents, que ponderant-les segons diferents premisses com són la quantitat d'ofertes de feina dels diferents llenguatges, entre d'altres, s'ha arribat als resultats il·lustrats a la següent figura.

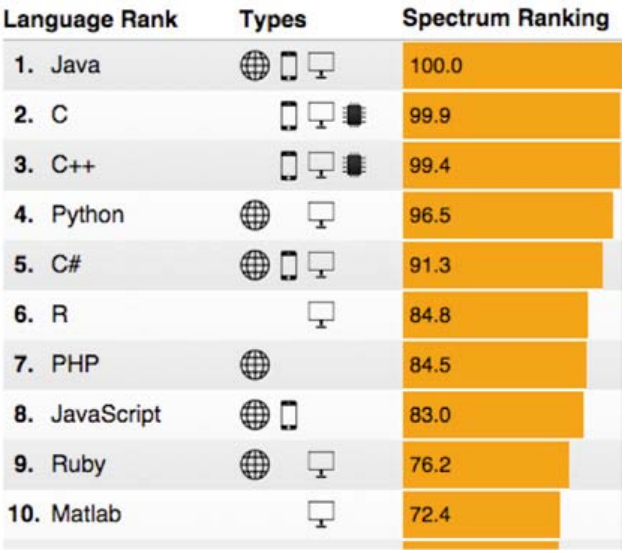

Figura 40 Ranking dels 10 llenguatges de programació més utilitzats el 2015 segons IEEE Spectrum.

Donat que la revolució més gran de les darreres dècades ha estat el naixement i creixement d'Internet, i que ens trobem en un moment en què, pràcticament, qualsevol sector de qualsevol indústria ja està connectat a Internet. Tenir presència *online* ja no és un caprici sinó una necessitat de qualsevol empresa o institució; per tant qualsevol eina que pugui facilitar el desenvolupament d'un portal, d'una aplicació, d'un servei o, en definitiva, de qualsevol solució requerida, és benvingut.

Per tant, doncs, aquest treball va dirigit a professionals del sector del desenvolupament web, que podran gaudir d'un conjunt de funcionalitats mínimes que li brindaran comoditat i versatilitat a l'hora de personalitzar el *framework* per acostar-lo al seu estil de programació, sempre partint d'una base sòlida.

I d'altra banda, també va dirigit a persones que tot just comencen a entrar al món de la programació i que tenen ganes d'entrar en el món web, ja que tot el projecte es fonamenta sobre el patró de disseny MVC, que possiblement sigui un dels patrons més fàcils d'entendre per aquelles persones que no tenen un perfil d'enginyeria del software, i que per tant els patrons de disseny els poden resultar més difícils de conceptualitzar.

[8] Dades extretes de http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages.

10. Seguretat

Pel què fa a seguretat, donat que tenim un sistema heterogeni hem de tenir en compte les diferents amenaces a què es pot haver d'enfrontar el sistema que contingui el *framework* en qüestió, no només el codi del *framework*. Així doncs, per una banda tindrem les amenaces que rebrà el servidor com a contenidor del *framework*, i per altra banda hem de preparar aquest per protegir-lo dels atacs més comuns.

Quan es parla de seguretat relativa a Internet el què és més important és seguir unes bones pràctiques a l'hora de tractar amb els diferents canals d'accés i les diferents vies des d'on es rebran dades d'entrada. Així doncs, els principals riscos a què ha d'enfrontar-se un *framework* sempre seran els que estiguin relacionats amb les dades d'entrada o els canals d'accés al mateix. Per tant, ja veiem que els punt més importants del *framework,* pel què fa a seguretat, són el paquet JepiFW\System\IO i el control d'accés a la informació continguda al servidor.

10.1 Possibles atacs

10.1.1 Atac XSS (Cross-Site Scripting)

Els atacs de Cross-Site Scripting són aquells mitjançant els quals s'insereix codi a una aplicació i se'n modifica el funcionament. Això pot ser més greu o menys segons a quin nivell arribi la injecció del codi.

Plantegem un exemple en el qual tenim un formulari com el següent:

```
<form action="/post/comment" method="post">
    <input type="text" name="comment" value="">
    <input type="submit" name="submit" value="Submit">
</form>
```

Com veiem, hi ha un camp anomenat `comment` que es rebrà dins la variable $_POST['comment']. Si aquesta variable no es processa ni es filtra de cap manera al ser enviada, una persona malintencionada podria introduir en el camp un script que pogués obrir una finestra emergent.

```
<script>alert("Hauries de filtrar les coses abans de guardar-les ;) ");</script>
```

Aquest exemple és senzill i pot semblar molt inofensiu, però no és important pel què fa sinó pel què suposa el fet que es pugui fer, ja que el què es pot aconseguir mitjançant un atac XSS és molt més que obrir un *popup*. Si volem prevenir que robin *cookies* o que es pugui extreure informació sensible dels usuaris, cal gestionar les dades d'entrada per tal d'evitar els atacs XSS.

10.1.2 Prevenir atacs XSS

La regla d'or en qualsevol entorn web és la següent:

"No confiar mai en les dades que venen de l'usuari o d'aplicacions de tercers."

Això és important tenir-ho clar perquè mai saps com t'arribarà la informació, així que tot ho has de tractar com si es tractés de dades que provenen d'un *hacker*. I per tant totes les dades han d'estar validades, corregides i escapades[9] abans d'afegir-les a la base de dades o bé abans de retornar-les a la resposta.

Al JepiFW s'ha utilitzat la següent funció per prevenir dels atacs XSS:

```php
public function xssPreventFilter($data){
    $trimmed = trim($data);
    $stripped = strip_tags($trimmed);
    $entities = htmlspecialchars($stripped);
    $sanitized = filter_var($entities, FILTER_SANITIZE_SPECIAL_CHARS);
    return $this->typedValue($sanitized);
}
```

Figura 41 Funció per prevenir els atacs XSS

Aquesta funció s'encarrega d'eliminar els espais del principi i final de la variable, després n'elimina els tags i finalment codifica els caràcters especials, a més d'aplicar un filtre estàndard de sanejament de les variables. Finalment es crida una funció que s'encarrega de forçar el tipus segons les dades que conté la variable, per tal que el retorn de la funció sigui una variable tipada.

10.1.3 SQL Injection

Un altre atac molt comú en entorns web és la injecció SQL. El què fa aquest atac és aprofitar les consultes de dades a la base de dades per a robar informació continguda a la mateixa mitjançant la modificació de dites consultes.

Per exemple, en un punt del nostra codi estem fent una consulta a la base de dades amb un paràmetre que es rep de la UI a través d'un formulari.

```
/* Variable correcte */
$name = "Manel";

/* Variable amb injecció SQL */
$name = "' OR 1 = 1";

/* Consulta que llista usuaris */
$query = "SELECT * FROM users WHERE name = '$name'";
```

[9] Escapar una variable en aquest entorn implica evitar que els caràcters propis dels tags (<, >) no es renderitzin com a tals i per tant es mostrin com als caràcters que són, sense processar.

Veiem com depenent del contingut de la variable que es fa servir per completar la consulta, podem tenir problemes perquè podran modificar la consulta al seu desig.

El primer pas per evitar aquest atac és utilitzar l'objecte PDO, que ja incorpora diversos elements de seguretat per protegir l'aplicació dels atacs més comuns. I a part d'utilitzar PDO per a dur a terme les connexions al servidor, a més és indispensable utilitzar la funció `prepare` ja que és l'encarregada de filtrar les dades que rep per evitar aquest atacs com aquest.

Això és un primer filtrat, però en molt recomanable assegurar-nos que filtrem les dades de l'aplicació abans d'enviar-les a la base de dades, independentment de si s'utilitza PDO o no.

10.2 Configuració del servidor

Un altre punt important, pel què fa a seguretat, és mantenir el servidor sempre actualitzat. En un servidor intervenen múltiples elements i cada un d'ells porta les seves vulnerabilitats associades, de tal manera que la millor manera de reduir els factors de risc és actualitzar tots els components a mesura que apareguin actualitzacions que involucrin millores de seguretat.

I també relatiu al servidor, hi ha un element que ens permet definir regles d'accés a la nostra aplicació. Aquest component és l'.htaccess, i en el nostre cas té definides algunes regles que eviten que es pugui accedir al contingut de determinades carpetes o fitxers.

El més interessant de l'.htaccess és que ens serveix per limitar i restringir qui pot veure què. Per defecte si accedim a una carpeta del nostre servidor, aquesta llistarà els fitxers que conté, de tal manera que si entrem a la carpeta que conté les imatges dels nostres usuaris, es podran obtenir totes les fotos dels usuaris de l'aplicació. Així que el què fem és negar-hi l'accés i accedir als fitxers des del codi, per tal de blindar les dades i només accedir-hi quan estem segurs que qui les demana n'és el legítim propietari.

11. Tests

S'ha optat per utilitzar una metodologia TDD així que els tests estan realitzats sobre cadascun dels diversos mòduls que componen el projecte i validen el funcionament a més baix nivell.

Els tests plantejats serveixen per validar els diferents mètodes de les classes de què consta el nucli del *framework*. I a més, per validar el correcte funcionament extrem a extrem, s'ha programat una aplicació utilitzant el JepiFW on es demostra que el funcionament és el desitjat.

Pels tests unitaris s'ha triat PHPUnit, que és un *framework* de testing orientat a objectes àmpliament estès i documentat. Gràcies al fet que PHPUnit obliga a treballar amb una estructura molt marcada, els tests són molt atòmics i per tant les funcionalitats que es validen amb cada test són molt concretes.

PHPUnit ofereix una classe per a crear els casos d'ús a validar, de tal manera que només s'han de definir els mètodes concrets de tests, i aleshores cada classe que extengui la classe de test PHPUnit_*Framework*_TestCase es considerarà un conjunt de tests que validen una funcionalitat. De tal manera que els fitxers de test que conté el JepiFW tots tenen una estructura similar a aquesta:

```
<?php

class ClasseExempleTest extends \PHPUnit_Framework_TestCase {
    /**
     * @var ClasseExemple
     */
    public static $object;

    public static function setUpBeforeClass() {
        self::$object = new ClasseExemple();
    }

    protected function tearDown() {
       unset(self::$object);
    }

    /**
     * @covers ClasseExemple::metodeExemple
     */
    public function testMetodeExemple() {
        $resultats = self::$object->metodeExemple();
        $this->assertEquals($resultatsEsperats, $resultats);
    }
}
?>
```

El primer que ens trobem en aquesta classe és que el nom és el mateix que el nom de la classe que es vol validar seguit de la paraula Test. El nom de la classe no tindria perquè condicionar en el nom de la classe de test, però és molt usual fer-ho d'aquesta maner ja que així es poden organitzar els

tests per classes on cada classe de test valida els mètodes d'una classe de l'aplicació; i el què si que és una restricció del *framework* de *testing* és el fet de què el nom de la classe de test acabi amb la paraula "Test". Això és així perquè quan s'executen els tests, es busquen totes les classes que compleixin una regla concreta en el nom (per defecte, com ja hem dit, es busquen els arxius amb el sufix "Test" al nom però podria canviar-se per seguir una altra regla).

Seguint amb l'exemple de codi d'una suposada classe de test del JepiFW, podem trobar com es defineix un objecte estàtic que s'inicialitza al mètode `setUpBeforeClass`. El motiu de fer-ho amb classes i mètodes estàtics és perquè PHPUnit crea una instància de la classe de test per a executar cada un dels test que conté, de tal manera que d'un mètode de test a l'altre no hi ha connexió i per tant no es conserva l'estat en què es troba la classe. Per això cal utilitzar una propietat estàtica a la classe de test, que contingui una instància de l'objecte que volem validar, per tal de poder conservar l'estat en què es trobi durant l'execució de tots els tests de la pròpia classe.

A mode d'exemple, donada una classe qualsevol amb dos mètode `get` i `set`, és evident que si volem escriure un test per validar aquesta classe crearem una classe de test amb dos tests com a mínim, un per testejar el mètode `set` i l'altre per testejar el mètode `get`. I el més normal seria voler definir un valor en un test, i després validar que et retorna el valor correcte en un altre. Per això ens resulta útil poder definir una propietat estàtica que contingui una instància de l'objecte que estem validant, perquè les classes de test no conserven estat de forma normal degut a què no s'executen tots els mètodes seguits sinó per separat a través de diverses instàncies, però les propietats estàtiques ens permeten donar-li un estat comú a totes les execucions.

Veiem que també es defineix un mètode `tearDown` que serveix per dur a terme tasques un cop s'hagin acabat d'executar tots els tests.

I finalment, el darrer mètode de l'exemple és un mètode de test que conté les comprovacions i validacions que determinen si la funcionalitat està correctament implementada. A l'interior d'aquest mètode s'utilitzen un seguit de funcions que ens proporciona PHPUnit per a dur a terme aquestes validacions.

Tot seguit es llistaran els diversos tests unitaris realitzats per a validar el correcte funcionament de l'aplicació, explicitant el resum de l'execució de cada fitxer, separat per paquets que conté el *framework*.

11.1 JepiFW\System\Config

Del mòdul de configuració se'n poden validar tant la classe abstracta ConfigAbstract com la classe final Config:

Classe	Nom test	Descripció
Config	`testLoadFile`	Valida que es guardin correctament les variables de configuració d'un fitxer amb extensió .ini dins la classe Config

Config	`testLoadArray`	Valida el mateix que `testLoadFile` però en comptes de recuperar les dades d'un fitxer les rep com a paràmetre
Config	`testGetData`	Valida que la funció `getData` retorni tota la informació de configuració continguda dins l'objecte Config
ConfigAbstract	`testSet`	Valida que el mètode `set` no falli a l'executar-se.
ConfigAbstract	`testGet`	Valida que el mètode `get` retorni la informació que conté un paràmetre d'una secció de configuració
ConfigAbstract	`testGetSection`	Valida que el mètode `getSection` retorni tot el contingut d'una secció en format d'array.

Taula 2 Llista de tests del paquet JepiFW\System\Config

I els resultats dels tests són els següents:

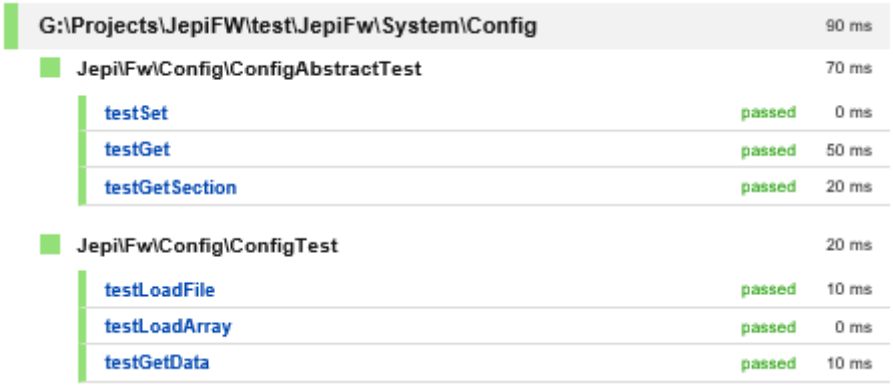

Figura 42 Resum de l'execució dels tests del paquet JepiFW\System\Config

11.2 JepiFW\System\FrontController

Per a definir un test per al FrontController necessitem primer assegurar-nos que l'execució es duu a terme en les mateixes condicions que quan s'executen els fitxers index.php i bootstrap.php al rebre's una petició, ja que la única cosa que fa el FrontController és iniciar el procés. Per crear aquest test hem hagut d'utilitzar la funció `setup` que permet definir tasques que s'executaran amb cada test, de manera que hem copiat el contingut del bootstrap en aquest mètode per tal de inicialitzar totes les coses necessàries per al correcte funcionament del *framework*.

Classe	Nom test	Descripció
FrontController	`testRun`	Valida que es retorni el contingut que retorna el mètode per defecte del controlador per defecte. El controlador per defecte és Home i el mètode per defecte és `index` i amb la configuració per defecte del *framework* ha de retornar "Hello World"

Taula 3 Llista de tests del paquet JepiFW\System\FrontController

I els resultats dels tests són els següents:

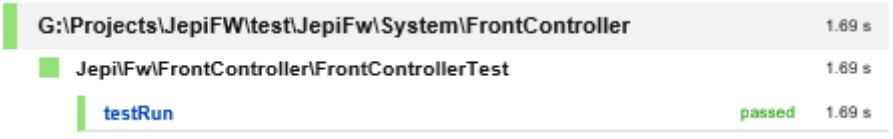

Figura 43 Resum de l'execució dels tests del paquet JepiFW\System\FrontController

11.3 JepiFW\System\Router

Del paquet de rutes només hi ha un objecte a provar que és la classe Router. Als test d'aquesta classe hem hagut d'inicialitzar-hi les dades simulant una petició, de manera que hem afegit variables com si es rebessin per GET o per POST, en pro de coincidir amb els paràmetres que espera rebre el mètode objectiu que es simula que s'està cridant. Tot això s'ha afegit al mètode `setUpBeforeClass` que consisteix en la inicialització de la variable Input que conté els paràmetres, la variable Router, que és l'objecte que es vol validar.

```
public static function setUpBeforeClass() {
    parent::setUpBeforeClass();

    $input = new Input();
    $input->setup(array('param1' => 1, 'param2' => 2),false);

    self::$object = new Router(self::$frontController->getConfig());
    self::$object->setInput($input);
    self::$object->checkRoute('/demo/testmethod');
}
```

Figura 44 Mètode setUpBeforeClass de la classe RouterTest

Els tests que es duen a terme del paquet de gestió de rutes són els següents:

Classe	Nom test	Descripció
Router	`testGetController`	Valida que es fa correctament la traducció de la ruta rebuda en un controlador. En aquest cas, veiem a la imatge anterior com la ruta és /demo/testmethod i per tant s'ha de comprovar que el controlador que va a buscar el Router és la classe Demo.
Router	`testGetAction`	Valida que el mètode que es cridi sigui el que defineix la ruta. En aquest cas, la ruta és /demo/testmethod i per tant hem d'assegurar-nos que el Router va a buscar el mètode anomenat `testmethod`.
Router	`testGetParameters`	Valida que els paràmetres rebuts són accessibles a través del mètode `getParameters`, ja que és la funció que s'utilitzarà per a llegir les dades que han de coincidir amb els paràmetres d'entrada del mètode que es cridarà.

Taula 4 Llista de tests del paquet JepiFW\System\Router

I els resultats dels tests són els següents:

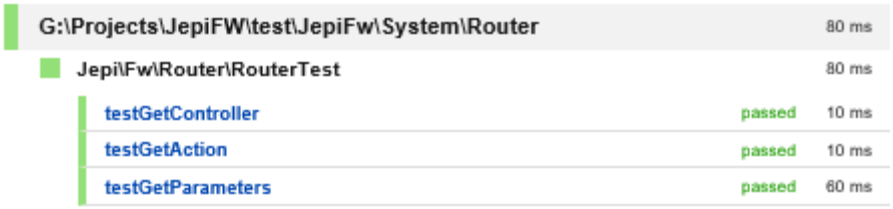

Figura 45 Resum de l'execució dels tests del paquet JepiFW\System\Router

11.4 JepiFW\System\IO

El mòdul d'entrada i sortida és el que més classes involucra, fent que el total de classes a testejar (excloent les interfícies) sigui de 5: DataCollection, Input, RequestData, Request i Response.

Classe	Nom test	Descripció
DataCollection	`testGet`	Valida que el mètode `get` de DataCollection retorna un objecte del tipus Input.
DataCollection	`testPost`	Valida que el mètode `post` de DataCollection retorna un objecte del tipus Input.
DataCollection	`testFiles`	Valida que el mètode `files` de DataCollection retorna un objecte del tipus Input.
Input	`testXssPreventFilter`	Valida que donat un paràmetre guardat a la classe Input, aquest és filtrat per evitar que es puguin dur a terme injeccions de codi maliciós.
Input	`testGet`	Valida que els valors continguts dins la variable súper global $_GET són retornats a través del mètode `get` de la classe Input. Assegurant que en cas d'afegir el modificador que permet retornar variables sense filtrar les injeccions de codi, retornin també els valors esperats. També es prova d'obtenir un valor que no existeixi per comprovar si retorna el valor per defecte.
Input	`testPost`	Valida que els valors continguts dins la variable súper global $_POST són retornats a través del mètode `get` de la classe Input. Assegurant que en cas d'afegir el modificador que permet retornar variables sense filtrar les injeccions de codi, retornin també els valors esperats. També es prova d'obtenir un valor que no existeixi per comprovar si retorna el valor per defecte.
RequestData	`testGetRemoteHost`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
RequestData	`testGetRemoteAddr`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
RequestData	`testGetRemotePort`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
RequestData	`testGetUri`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
RequestData	`testGetMethod`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
RequestData	`testGetTime`	Valida que el valor retornat és un nombre sencer.
RequestData	`testGetTimeFloat`	Valida que el valor retornat és un nombre decimal.

RequestData	`testGetUserAgent`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
RequestData	`testGetScriptFilename`	Valida que el valor retornat és una cadena de text.
RequestData	`testGetQueryString`	Valida que el valor retornat és del tipus correcte. A l'executar-se sense tractar-se d'una petició ha de retornar el valor per defecte .
Request	`testGetRequest`	Valida que el mètode `getRequest` retorna un objecte del tipus RequestData i que els mètodes retornen els valors del tipus esperats.
Request	`testGetHeader`	Valida que es poden obtenir correctament les capçaleres de la petició.
Request	`testValidateRequest`	Valida que la funció `validateRequest` no dóna una excepció i que, a més, retorna l'objecte Router.
Response	`testSend`	Valida que la funció `send` retorna la sortida de l'execució del controlador i mètode per defecte, i s'assegura que ho fa a través de la sortida estàndard (imprimint el contingut) i retornant-lo com a resposta a la crida de la funció.

Taula 5 Llista de tests del paquet JepiFW\System\IO

I els resultats dels tests són els següents:

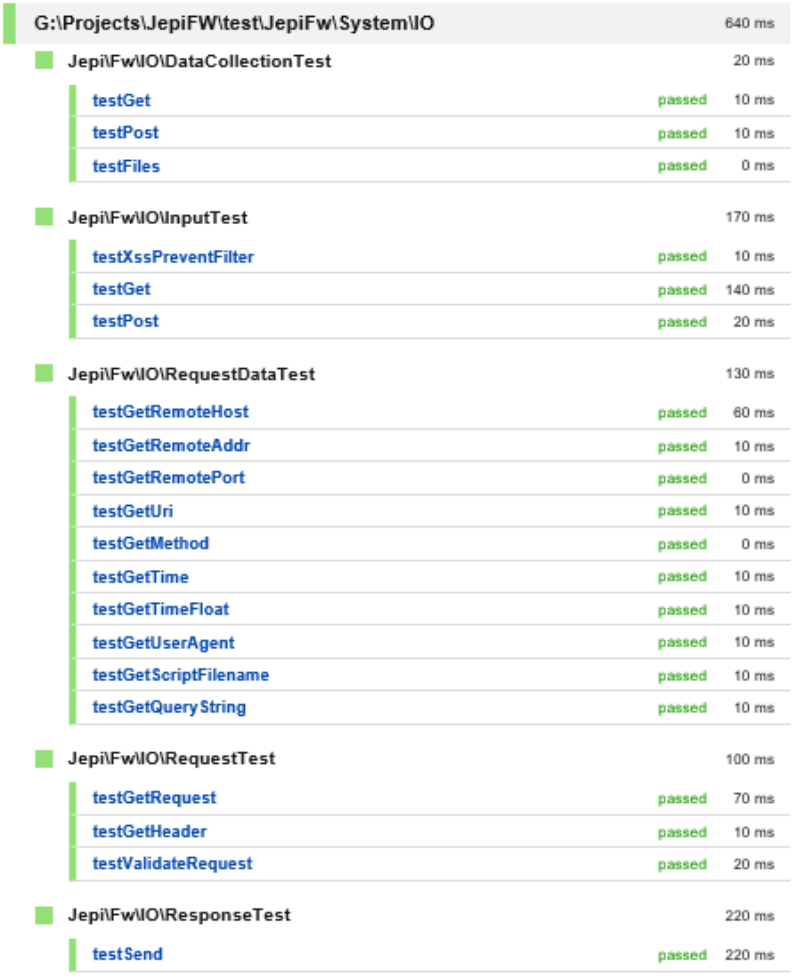

Figura 46 Resum de l'execució dels tests del paquet JepiFW\System\IO

11.5 JepiFW\System\Security

Del paquet de seguretat, com ja s'ha comentat, només es disposa d'una classe i per tant els tests d'aquest mòdul es redueixen a un:

Classe	Nom test	Descripció
XssFilter	`testXssPreventFilter`	Valida el filtre que evita que puguin incrustar codi al sistema.

Taula 6 Llista de tests del paquet JepiFW\System\Security

I els resultats dels tests són els següents:

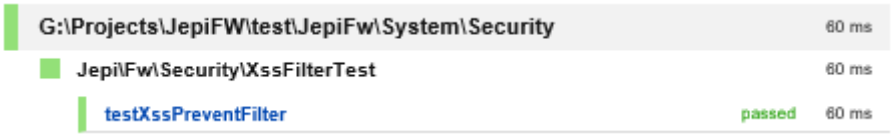

Figura 47 Resum de l'execució dels tests del paquet JepiFW\System\Security

11.6 JepiFW\System\Model

Els tests dels models han de validar el correcte funcionament de la gestió de les connexions a base de dades. Les classes a testejar són Connections i MySqlModel. Per a dur a terme els tests d'aquest mòdul s'ha hagut de definir una base de dades i un model que hi interactuï. L'esquema de la base de dades es troba al fitxer /test/schema.sql, i el model que ens ajudarà a valida el funcionament de MySqlModel es troba dins de /tests/JepiFW/System/Model/ModelExample. S'ha creat una classe que estengués de MySqlModel per poder simular un cas real, ja que no s'hauria d'utilitzar MySqlModel directament sinó sempre fer-ho a través d'una classe que l'extengui.

Classe	Nom test	Descripció
Connections	`testOpenMySqlConnection`	Valida que a l'obrir una connexió es retorna un objecte PDO a través del qual es podrà interactuar amb la base de dades, sense que es generi cap excepció en el procés.
Connections	`testCloseConnection`	Valida que la connexió s'hagi tancat correctament i no es generi cap excepció en el procés.
MySqlModel	`testSelect`	Valida que el llistat d'elements que ens retorna el model es correspon amb els que inicialment hi ha a la base de dades.
MySqlModel	`testInsert`	Valida que el model és capaç d'inserir nous objectes a la base de dades.
MySqlModel	`testUpdate`	Valida que el model actualitza amb èxit els objectes ja existents a la base de dades
MySqlModel	`testDelete`	Valida que el model elimina els objectes de la base de dades.

Taula 7 Llista de tests del paquet JepiFW\System\Model

I els resultats dels tests són els següents:

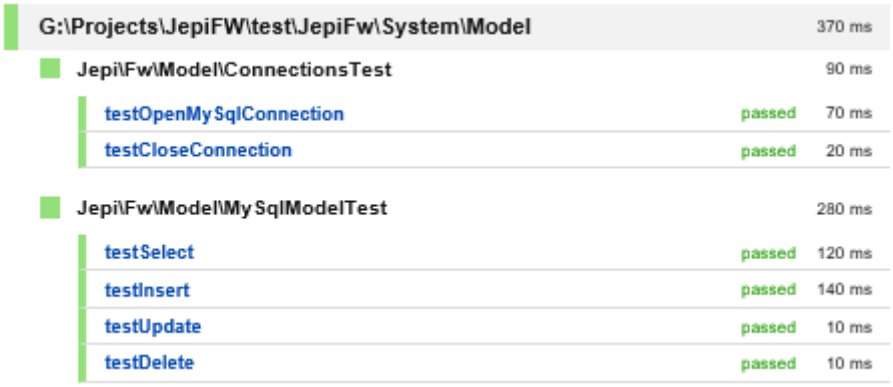

Figura 48 Resum de l'execució dels tests del paquet JepiFW\System\Model

11.7 JepiFW\System\View

Les vistes són imprescindibles però no porten associada una gran funcionalitat així que els tests de la classe View són molt limitats. Es descriuen tot seguit:

Classe	Nom test	Descripció
View	`testAddVar`	Valida que la vista rep i retén correctament les variables que s'afegeixen a la vista per tal de ser inserides posteriorment al moment de renderitzar la mateixa.
View	`testGet`	Valida que la vista és capaç d'executar el fitxer .php rebut per paràmetre i retornar-ne el contingut en format text.

Taula 8 Llista de tests del paquet JepiFW\System\View

I els resultats dels tests són els següents:

Figura 49 Resum de l'execució dels tests del paquet JepiFW\System\Model

12. Versions de l’aplicació/servei

Actualment el JepiFW es troba a la versió 1.2.3, i es pot fer el seguiment de les versions anteriors des de l’enllaç de Packagist del projecte: https://packagist.org/packages/jepi/fw.

Per a fer el versionat del codi s’està seguint el format X.Y.Z, on X és el número de la versió estable, la Y és per indicar canvis o millores funcionals, i la Z serveix per marcar els canvis o millores de menor importància. En anglès es fa referència a *major issue* o *minor issue*, que ho podríem entendre com a temes importants o temes menys importants. Per tant, quan hi ha canvis importants s’incrementa la Y, i quan hi ha canvis senzills o poc compromesos, la Z. D’aquesta manera guardem la X única i exclusivament per als canvis de versió, que poden portar associats problemes de retrocompatibilitat, etc. I que per tant, per a realitzar un canvi de versió s’ha de fer un bon estudi de la situació prèviament a realitzar-lo.

Cal recalcar que el projecte es troba en versió estable, però segueix viu i per tant seguirà tenint canvis (per més informació sobre els canvis, consultar l’apartat de Projecció a Futur).

13. Instruccions d'instal·lació

Perquè el *framework* funcioni correctament és necessari disposar dels recursos següents:

- *Software*:
  - PHP 5.6
  - PHPUnit 5.0
  - MySql 5.5
  - Composer

- *Hardware*: No hi ha requeriments específics.

Per a instal·lar el *framework* cal seguir els següents passos:

1. Instal·lar Composer
   a. Per instal·lar composer cal seguir les instruccions que hi ha a l'enllaç següent: https://getcomposer.org/doc/00-intro.md#system-requirements

2. Crear un projecte amb el jepi/fw

```
> composer create-project jepi/fw
```

3. Actualitzar les dependències del *framework* (això es fa automàticament al crear el projecte).

```
> composer update
```

4. Modificar el fitxer de configuració ubicat a JepiFW/App/Config/config.ini
   a. Modificar les credencials de la base de dades.
   b. Crear base de dades amb l'esquema contingut al fitxer JepiFW/test/schema.sql.
   c. Modificar el valor de configuració Routing > SiteUrl per indicar quina és la URL que conté la nostra aplicació.

5. Si tot ha anat bé, si et dirigeixes a la pàgina que has indicat al paràmetre SiteUrl, hauries de llegir "Hello World" al navegador.

6. Addicionalment, es pot comprovar que passen correctament tots els tests mitjançant l'execució de la següent comanda:

```
> cd [Ruta_Del_Projecte]
> php vendor/phpunit/phpunit/phpunit --bootstrap vendor/autoload.php –configuration phpunit.xml
```

I la sortida hauria de ser semblant a aquesta:

```
G:\Projects\JepiFW>php vendor/phpunit/phpunit/phpunit --bootstrap vendor\autoload.php
--configuration phpunit.xml
PHPUnit 5.0.10 by Sebastian Bergmann and contributors.

.....................................                             39 / 39 (100%)

Time: 11.89 seconds, Memory: 7.00MB

OK (39 tests, 67 assertions)
```

Figura 50 Resultats de l'execució dels tests a través de terminal

14. Instruccions d'ús

Una vegada instal·lat el *framework* ja es pot començar a utilitzar, i per a fer-ho cal saber com funciona. Tot seguit es descriu com utilitzar el JepiFW per a desenvolupar una aplicació web.

NOTA: Abans de continuar és important haver llegit l'apartat 16: Instruccions d'instal·lació, per tal de tenir tot l'entorn preparat.

14.1 Controladors

14.1.1 Controlador i acció per defecte

El controlador per defecte és la classe que s'anirà a buscar en cas que la ruta introduïda al navegador no coincideixi amb cap controlador existent. El nom del controlador per defecte depèn del config.ini, i si no s'especifica el contrari serà la classe Home.

Les propietats del config.ini que corresponen al controlador i acció per defecte estan dins las secció de *routing* i són les següents:

```
[Routing]
DefaultController = Home
DefaultAction = index
```

14.1.2 Crear un controlador

Per crear un controlador únicament ens cal crear una classe que extengui la classe \Jepi\Fw\Controller\Controller i estigui dins el namespace App\Controllers (aquest namespace s'ha especificat al config.ini i també es pot canviar si es desitja).

Els controladors tenen una peculiaritat interessant i és la possibilitat de inicialitzar classes únicament definint-les i afegint l'anotació @Inject al comentari. Per exemple:

```
<?php

namespace App\Controllers;
use Jepi\Fw\Model\ModelExample;

class Example extends \Jepi\Fw\Controller\Controller
{
    /**
     * @Inject
     * @var ModelExample
```

```
     */
    private $model;

    public function index(){
        //Utilitza $this->model aquí!
    }
}
```

El millor d'utilitzar la injecció de classes als controladors és que aquestes classes realment no s'instanciaran fins que no s'utilitzin, de tal manera que en aquest punt, almenys, no hi ha cap penalització de rendiment per utilitzar aquesta funcionalitat.

I això mateix es pot fer amb qualsevol classe de l'aplicació, ja sigui del nucli del *framework* com les classes definides per l'usuari.

14.2 Models

14.2.1 Configurar base de dades

Per poder gaudir de connexions a base de dades amb el JepiFW només cal que en definim les credencials al config.ini i que creem models que extenguin la classe \Jepi\Fw\Model\MySqlModel.

Primer afegim les credencials:

```
[Database]
defaultConnection = default

host[default] = localhost
port[default] = 3306
user[default] = foo
pass[default] = foobar
name[default] = foo
```

I creem un model:

```
class Example extends \Jepi\Fw\Model\MySqlModel {
[...]
```

14.2.2 Utilitzar múltiples connexions

El *framework* per defecte pot utilitzar múltiples connexions a la base de dades, però sempre n'hi haurà una que serà la que està guardada com a connexió per defecte. Aleshores, als models és

necessari definir el nom de la connexió que s'utilitzarà quan aquesta no sigui la connexió per defecte, tot i que és altament aconsellable definir-la sempre per reduir possibles errors.

Veiem un exemple de les modificacions necessàries:

- Pel què fa al config.ini hi ha un exemple als annexos on s'inclouen dues connexions a base de dades.
- Al model cal definir la propietat dbConnection amb el nom definit com a clau al config.ini. Seguint el cas de l'exemple que està als annexos, tenim una connexió a base de dades anomenada "default" i una altra anomenada "todo", doncs com es pot veure al TaskModel.php que també està als annexos, afegim la propietat dbConnection (que és protected) i li donem el valor "todo", perquè no és la connexió per defecte.

14.2.3 Creació d'un model

Els models com ja hem vist, han d'extendre la classe \Jepi\Fw\Model\MySqlModel, i un cop fet això podrem gaudir dels mètodes de gestió de consultes SQL, que es defineixen tot seguit:

- `select`: executarà consultes de lectura de informació de la base de dades i retornarà un array amb els elements que hagi trobat.
- `insert`: executarà consultes d'inserció d'informació a la base de dades i retornarà la ID de l'últim element inserit.
- `update`: executarà consultes d'actualització d'informació de la base de dades i retornarà el nombre de files actualitzades.
- `delete`: executarà consultes d'eliminació d'informació de la base de dades i retornarà el nombre d'elements eliminats.

14.3 Vistes

14.3.1 Afegir variables a una vista

Des dels controladors es pot interactuar amb l'objecte View, ja que aquest està dins del propi controlador. I per tant, l'únic que ens cal fer per definir variables per la vista és afegir-les a través del mètode `addVar` o bé enviant-les directament al moment de recuperar la vista amb el mètode `get`.

```
public function welcome(){
    $this->view->addVar('title', 'JepiFW Template');
    $this->view->addVar('author', 'Jepi Humet Alsius');

    return  $this->view->get('bootstrap-template.php',  array('content'  =>
'Hello, world!'));
}
```

14.3.2 Treballar amb vistes

A l'exemple anterior hem pogut veure com es recupera una vista, que és mitjançant el mètode `get` de l'objecte View. El què cal tenir en compte és que al moment de cridar aquesta funció, no cal passar-li la ruta absoluta del fitxer de la vista sinó la ruta relativa al *path* que està definit al config.ini

```
[Views]
ViewsDirectory = /Views
```

El contingut d'una vista ha de ser un fitxer HTML normal, amb extensió .php, i a l'interior podem utilitzar els tags "<?= $var ?>" (manera curta d'escriure "<?php echo $var ?>") per incrustar les variables a l'HTML. Es pot veure tot el contingut del fitxer bootstrap-template.php als annexos.

14.4 Variables de Sessió

14.4.1 Llegir i escriure variables de sessió

Per a treballar amb variables de sessió només ens cal accedir a la propietat "session" que ja tenen els controladors fruit de l'herència de la classe Controller. Aleshores per interactuar amb aquest objecte només ens cal fer-ho de la següent manera:

```
class Example extends \Jepi\Fw\Controller\Controller {

      public function listCookies(){
            $this->session->set("foo", "Bar");
            return  $this->session->get("foo");
      }
}
```

14.4.2 Variables flash

Les variables *flash* són variables de sessió d'un sol accés. Estan pensades per a poder passar informació d'una pàgina a la següent sense haver de preocupar-nos d'esborrar les dades un cop llegides.

Per a utilitzar les variables *flash* es pot fer a través dels mètodes `setFlash` i `flash`. Vegem-ho:

```
class Example extends \Jepi\Fw\Controller\Controller {

      public function listCookies(){
            $this->session->setFlash("foo", "Bar");
            return  $this->session->flash("foo");
```

```
	}
}
```

14.5 Cookies

14.5.1 Llegir i escriure cookies

Per a llegir i escriure *cookies* és tant senzill com injectar l'objecte de Cookies al controlador i després utilitzar-lo normalment al codi mitjançant els mètodes `get` i `set`, entre d'altres. Vegem un exemple de controlador:

```
class Example extends \Jepi\Fw\Controller\Controller {

	/**
	 * @var \Jepi\Fw\Storage\Cookies
	 */
	private $cookies;

	public function listCookies(){
		$this->cookies->set("foo", "Bar");
		return  $this->cookies->get("foo");
	}
}
```

14.6 Noms d'espai

14.6.1 Utilitzar noms d'espai diferent

Per a modificar els noms de espai que han de seguir les classes de l'aplicació es pot fer mitjançant les variables següents de l'arxiu de configuració:

```
[Namespaces]
App = \App
Controllers = \App\Controllers
Models = \App\Models
Extensions = \App\Core
```

14.7 Input

14.7.1 Valor per defecte

Quan no hi ha cap element que coincideixi amb una determinada clau d'un conjunt de dades d'entrada es retorna sempre un valor que correspon a retorn buit. Aquest valor podria ser 0, null, false, -1, etc. Al JepiFW s'ha optat per donar-li false com a valor per defecte, però es pot canviar afegint o modificant els següents paràmetres de configuració del fitxer JepiFW/App/Config/config.ini.

```
[Input]
UnsetValue = false
```

15. Projecció a futur

A data d'entregar el projecte, ja hi ha una versió 2.0 del *framework* en fase de desenvolupament que consta d'algunes peculiaritats interessants, però donat que no és una versió estable (ni tan sols acabada), no s'ha fet el treball al voltant d'aquesta sinó de la versió 1.2, que és la versió estable amb què s'ha desenvolupat l'aplicació de demostració.

Les millors que es preveuen incorporar a la versió 2.0 un cop estigui finalitzada son les següents:

- El què ja s'ha començat a fer per a la primera release de la versió 2.0 del JepiFW és aprofitar la divisió del *framework* en paquets per separar-los realment en diversos projectes, gestionant la interdependència entre ells a través de composer. Així cada paquet pot ser utilitzat per separat de la resta.

  Per tant, doncs, des del perfil de Packagist https://packagist.org/users/jepihumet/packages/ ja es pot accedir a tots els paquets per separat. Com ja s'ha dit anteriorment en aquest document, composer permet definir dependències entre paquets, així que tenint diversos paquets separats, ens faltaria ajuntar-los en un sol projecte. Per tant el JepiFW 2.0 serà un projecte que tindrà com a dependències els diversos paquets que conformen el *framework*, per deixar-los tots lligats en un únic paquet, tot i ser plenament funcionals individualment.
- L'altra millora que es voldria fer és un *script* d'inicialització a través del procés d'instal·lació del *framework.* Composer permet definir tasques a dur a terme abans i/o després dels diversos processos que pot fer, de tal manera que al moment de fer la crida de creació de projecte, es podria fer que et sol·licités les credencials de la base de dades, el nom del *host* on es troba el projecte, etc. Per tal que es configuri l'arxiu config.ini de manera automàtica, entre altres coses.
- Es vol incorporar la possibilitat d'utilitzar un sistema ORM com el que utilitza Symfony. La idea seria que fos compatible la possibilitat d'operar amb un sistema ORM com Doctrine i utilitzar la gestió de models que es fa actualment. Si bé és cert que al treballar amb un sistema ORM pots operar únicament amb entitats que s'emmirallen de la base de dades, i per tant no faria falta l'ús de SQL directament. Considero molt interessant la coexistència dels dos sistemes de gestió de dades per a permetre dur a terme optimitzacions, ja que la gran penalització que suposa treballar amb sistemes ORM és l'increment substancial del nombre de consultes que són necessàries per a dur a terme qualsevol tasca a la base de dades.
- A més de les bases de dades relacionals, darrerament s'estan posant de moda les bases de dades NoSQL, que són un altra paradigma de base de dades basat en objectes, i no en taules relacionades com passa amb SQL. Així que també seria útil integrar la possibilitat de gestionar bases de dades amb sistemes no relacionals.
- Es vol implementar Twig com a motor de plantilles. Actualment la gestió de vistes és bastant senzilla, ja que no es fa un procés del contingut de les mateixes sinó que simplement es mostren i res més. Per tant seria interessant incorporar un post-processat de la informació continguda en les vistes. Per més detalls sobre Twig es pot consultar l'apartat 3: Marc Teòric, que presenta les bases comparatives sobre les quals s'ha fonamentat aquest treball.
- Per incrementar la velocitat del *framework* en termes generals seria molt útil incorporar sistemes de *cache* per guardar la informació recurrent, com les anotacions del codi, per exemple; o les vistes, ja que a vegades hi ha dades que es mostren moltes vegades i cada

vegada s'han d'anar a buscar a la base de dades, processar, etc., i si tinguéssim les vistes en *cache* ja podríem retornar el contingut processat i així reduir substancialment el temps de resposta. Aquest sistema de *cache* de vistes és el mateix que utilitza CodeIgniter.

- Un altre tema important, sobretot quan una aplicació comença a créixer, és el fet de poder conèixer quan passen les coses durant l'execució per a poder incrustar codi a mode de *hooks*. Així que el què es voldria és afegir un sistema de gestió d'esdeveniments perquè els mòduls puguin interferir en l'execució d'una tasca determinada. Quelcom semblant a un sistema d'alertes internes, amb possibles subscripcions a les mateixes, per tal d'executar un determinat fragment de codi en el moment en què salti dita alerta.

16. Conclusions

Després de molta feina, s'ha acabat el projecte en tant que exercici acadèmic, però ha nascut un projecte que tinc ganes de continuar tirant endavant i fent créixer. Com es pot veure, tinc moltes idees en ment per integrar noves funcionalitats i noves característiques al què considero que és una bona base.

Crec que l'aprenentatge més bo que he tret d'aquesta experiència és l'adonar-me que fer les coses ben fetes no és fàcil, s'ha de treballar molt, i al ser meticulós amb la feina feta. Això no vol dir que estigui tot perfecte, ja que considero que hi ha moltes coses que podrien polir-se molt, però el propòsit era obtenir un *framework* funcional, que pogués competir amb els actuals *frameworks* del mercat en quan a rendiment i, sobretot, que fos útil. I el què he descobert després de crear una petita aplicació, és que el JepiFW és útil, fàcil d'utilitzar i ràpid de desenvolupar-hi. I a més, constitueix una base compacte sobre la qual es pot seguir fent créixer tant com es vulgui el projecte.

El fet que el JepiFW hagi estat el meu projecte final de carrera ha estat una motivació extra per a dur a terme de la millor manera possible un projecte que feia temps que tenia pendent. Ja que el meu dia a dia es caracteritza pel desenvolupament web en PHP, i per tant, crear una eina així m'ha permès obtenir un coneixement molt més tangencial de tot el què implica la gestió i procés de dades des que es rep la petició fins que es proporciona una resposta.

Pel què fa a rendiment, s'ha dut a terme un petit *benchmarking* basat en proves d'estrés a tres espais web que contenien cadascú una aplicació: una amb CodeIgniter, una altra amb Symfony i una darrera amb JepiFW:

```
jepi@vps143632:~$ ab -c 20 -n 500 http://prbam.jepihumet.com/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking prbam.jepihumet.com (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Finished 500 requests


Server Software:        Apache/2.4.10
Server Hostname:        prbam.jepihumet.com
Server Port:            80

Document Path:          /
Document Length:        7151 bytes

Concurrency Level:      20
Time taken for tests:   10.667 seconds
Complete requests:      500
Failed requests:        0
Total transferred:      4325000 bytes
HTML transferred:       3575500 bytes
Requests per second:    46.87 [#/sec] (mean)
Time per request:       426.672 [ms] (mean)
Time per request:       21.334 [ms] (mean, across all concurrent requests)
Transfer rate:          395.96 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    7  41.3      0     614
Processing:    15  400 879.5    153    8645
Waiting:        0  397 879.2    151    8635
Total:         22  407 880.5    154    8648

Percentage of the requests served within a certain time (ms)
  50%    154
  66%    220
  75%    265
  80%    303
  90%    699
  95%   2088
  98%   4294
  99%   5163
 100%   8648 (longest request)
jepi@vps143632:~$
```

Figura 51 Test d'estrés d'una web desenvolupada amb CodeIgniter

```
jepi@vps143632:~$ ab -c 20 -n 500 http://127.0.0.1:8000/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Finished 500 requests


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8000

Document Path:          /
Document Length:        26027 bytes

Concurrency Level:      20
Time taken for tests:   39.860 seconds
Complete requests:      500
Failed requests:        0
Total transferred:      13177000 bytes
HTML transferred:       13013500 bytes
Requests per second:    12.54 [#/sec] (mean)
Time per request:       1594.396 [ms] (mean)
Time per request:       79.720 [ms] (mean, across all concurrent requests)
Transfer rate:          322.83 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0       2
Processing:   351 1559 750.4   1317    3531
Waiting:      344 1546 745.0   1306    3464
Total:        352 1559 750.4   1317    3531

Percentage of the requests served within a certain time (ms)
  50%   1317
  66%   1862
  75%   2133
  80%   2334
  90%   2646
  95%   3135
  98%   3312
  99%   3377
 100%   3531 (longest request)
jepi@vps143632:~$
```

Figura 52 Test d'estrés d'una web desenvolupada amb Symfony

```
jepi@vps143632:~$ ab -c 20 -n 500 http://jepifw.jepihumet.com/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking jepifw.jepihumet.com (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Finished 500 requests


Server Software:        Apache/2.4.10
Server Hostname:        jepifw.jepihumet.com
Server Port:            80

Document Path:          /
Document Length:        4320 bytes

Concurrency Level:      20
Time taken for tests:   31.395 seconds
Complete requests:      500
Failed requests:        0
Total transferred:      2342500 bytes
HTML transferred:       2160000 bytes
Requests per second:    15.93 [#/sec] (mean)
Time per request:       1255.787 [ms] (mean)
Time per request:       62.789 [ms] (mean, across all concurrent requests)
Transfer rate:          72.87 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    4  19.7      0     124
Processing:   447 1243 490.7   1074    3883
Waiting:      439 1242 491.0   1073    3883
Total:        515 1246 490.8   1078    3883

Percentage of the requests served within a certain time (ms)
  50%   1078
  66%   1208
  75%   1385
  80%   1514
  90%   1924
  95%   2315
  98%   2742
  99%   3241
 100%   3883 (longest request)
jepi@vps143632:~$
```

Figura 53 Test d'estrés d'una web desenvolupada amb JepiFW

En tots tres tests s'ha fet el mateix, un total de 500 peticions amb una concurrència de 20 consultes simultànies. I veiem com els resultats són bastant prometedors. El resum de resultats és el següent:

Framework	Temps total	Peticions/segon	Temps/petició
CodeIgniter	10,667 s.	46,87	21,334 ms.
Symfony	39,860 s.	12,54	79,720 ms.
JepiFW	31,395 s.	15,93	62,789 ms.

Taula 9 Resultats dels tests d'estrés passat als diferents *frameworks*

Veiem doncs, que els resultats no són gens dolents, ja que s'ha aconseguit ubicar el rendiment entre els dos *frameworks* que s'havien agafat com a referents. Tot i que també és important tenir en compte que Symfony incorpora moltes més funcionalitats que el JepiFW i per tant és important fer cura d'humilitat i acceptar que tot i que els resultats són bons, no es poden quedar en resultats sinó que se n'ha de fer una lectura adequada. Així que, en aquest cas, tot i que els resultats són favorables al JepiFW per sobre de Symfony es creu que un dels punts que s'haurà de mirar de millorar en un futur és l'eficiència i el rendiment.

Annex 1. Lliurables del projecte

El projecte lliurat consta de tres documents principals, que són els següents:

- Memòria del projecte
- Vídeo de defensa on s'explica el projecte
- Codi font del *framework*
    - Inclou el codi font de l'exemple d'aplicació programada amb el *framework*

Tant el *framework*, com l'aplicació de demostració que inclou el mateix (gestor de tasques), es poden descarregar a través de GitHub:

- https://github.com/jepihumet/JepiFW

o a través de Packagist:

- https://packagist.org/packages/jepi/fw.

Annex 2. Codi font (extractes)

Tot seguit es presenten un conjunt de fragments rellevants del codi font del *framework*, per brindar més grau de detall a alguna de es parts ja comentades al llarg del document.

ModelExample.php

Aquesta classe és utilitzada en els tests dels models i ens serveix d'exemple per com crear un model al JepiFW. Com veiem, únicament cal extendre de la classe MySqlModel i a partir d'aquí es poden utilitzar els quatre mètodes que ja estan preparats per a interactuar amb la base de dades:

- `select`: per llegir informació de la BD.
- `insert`: per inserir informació a la BD.
- `update`: per actualitzar la informació de la BD.
- `delete`: per eliminar informació de la BD

```
<?php
namespace Jepi\Fw\Model;
class ModelExample extends MySqlModel
{
    /**
     * @return mixed
     */
    public function listUsers(){
        return $this->select("SELECT id, name FROM user");
    }

    /**
     * @param $id
     * @return mixed
     */
    public function getUser($id){
        $data = $this->select("SELECT id, name FROM user WHERE id = $id");
        return $data[0];
    }

    /**
     * @param $name
     * @return int
     */
    public function createUser($name){
        return $this->insert("INSERT INTO user (name) VALUES ('$name')");
    }

    /**
     * @param $id
     * @param $name
     * @return int
     */
    public function updateUser($id, $name){
        return $this->update("UPDATE user SET name = '$name' WHERE id = $id");
    }
```

```
    /**
     * @param $id
     * @return int
     */
    public function deleteUser($id){
        return $this->delete("DELETE FROM user WHERE id = $id");
    }
}
```

Schema.sql

L'anterior script SQL conté l'esquema que presenta la base de dades que utilitzen els tests del JepiFW.

```
CREATE DATABASE  IF NOT EXISTS `foo` /*!40100 DEFAULT CHARACTER SET utf8
USE `foo`;

--
-- Table structure for table `user`
--

DROP TABLE IF EXISTS `user`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
```

Config.ini

Al següent exemple es mostren totes les possibles variables que pot contenir el fitxer config.ini

```
[Namespaces]
App = \App
Controllers = \App\Controllers
Models = \App\Models
Extensions = \App\Core

[Input]
UnsetValue = false

[Routing]
SiteUrl = http://localhost/
DefaultController = Home
DefaultAction = index
```

```
AutoRedirect = false

[Database]
defaultConnection = default

host[default] = localhost
port[default] = 3306
user[default] = foo
pass[default] = foobar
name[default] = foo

host[todo] = localhost
port[todo] = 3306
user[todo] = todo
pass[todo] = CaMFAsyzeb9D7nyL
name[todo] = todo

[Storage]
Encrypt = true
EncryptionKey = blablabla

[Cookies]
DefaultExpiration = 60000
DefaultPrefix = ''
DefaultDomain = ''
DefaultPath = '/'
DefaultSecure = false
DefaultHttpOnly = false

[Views]
ViewsDirectory = /Views
```

Todo.php

El següent codi és el codi que correspon al controlador que llista les tasques d'una llista de tasques, de l'aplicació d'exemple:

```
<?php

namespace App\Controllers;

/**
 * Todo.php
 *
 * @package     JepiFW
 * @author      Jepi Humet Alsius <jepihumet@gmail.com>
 * @link        http://jepihumet.com
 */
class Todo extends \Jepi\Fw\Controller\Controller {

    /**
     * @Inject
     * @var \App\Models\ListModel
```

```
     */
    private $lists;

    /**
     * @Inject
     * @var \App\Models\TaskModel
     */
    private $tasks;

    /**
     * @var \Jepi\Fw\Storage\Cookies
     */
    private $cookies;

    public function tasks($id = 0) {
        $lists = $this->lists->allLists();

        if (count($lists) > 0) {
            $selectedListIndex = 0;
            for ($i = 0; $i < count($lists); $i++) {
                if ($lists[$i]['id_list'] == $id) {
                    $lists[$i]['active'] = true;
                    $selectedListIndex = $i;
                    break;
                }
            }
            $lists[$selectedListIndex]['active'] = true;
            $currentList = $lists[$selectedListIndex];
            $tasks = $this->tasks->getListTasks($currentList['id_list']);
        } else {
            $tasks = array();
            $currentList  = false;
        }

        $this->view->addVar('lists', $lists);
        $this->view->addVar('currentList', $currentList);
        $listsView = $this->view->get('lists.php');

        $this->view->addVar('tasks', $tasks);
        $tasksList = $this->view->get('tasks-list.php');

        $this->view->addVar('title', 'ToDo manager');
        $this->view->addVar('listsView', $listsView);
        $this->view->addVar('tasksView', $tasksList);
        return $this->view->get('bootstrap-template.php');
    }
    /**
     * Default method will return tasks list for first ToDo list.
     * @return string
     */
    public function index() {
        return $this->tasks();
    }
}
```

TaskModel.php

El següent codi és el codi que correspon al model que gestiona les tasques d'una llista de tasques, de l'aplicació d'exemple:

```
<?php

namespace App\Models;

/**
 * TaskModel.php
 *
 * @author      Jepi Humet Alsius <jepihumet@gmail.com>
 * @link        http://jepihumet.com
 */
class TaskModel extends \Jepi\Fw\Model\MySqlModel {

    protected $dbConnection = 'todo';

    public function allTasks() {
        $sql = "SELECT id_task, id_list, name, description, status FROM tasks";
        return $this->select($sql);
    }
    public function getTask($idTask) {
        $sql = "SELECT id_task, name, description, status FROM tasks WHERE id_task = '$idTask' LIMIT 1";
        $data = $this->select($sql);
        if (count($data) > 0) {
            return $data[0];
        } else {
            return false;
        }
    }
    public function getListTasks($idList) {
        $sql = "SELECT id_task, name, description, status FROM tasks WHERE id_list = '$idList'";
        return $this->select($sql);
    }
    public function createTask($idList, $name, $description, $status) {
        $sql = "INSERT INTO tasks (id_list, name, description, status) VALUES ('$idList', '$name', '$description', '$status')";
        return $this->insert($sql);
    }
    public function updateTask($idTask, $name, $description, $status) {
        $sql = "UPDATE tasks SET name = '$name', description = '$description', status = '$status' WHERE id_task = '$idTask' LIMIT 1";
        return $this->update($sql);
    }
    public function deleteTask($idTask) {
        $sql = "DELETE FROM tasks WHERE id_task = '$idTask' LIMIT 1";
        return $this->delete($sql);
    }
}
```

bootstrap-template.php

Aquest fitxer és la vista que utilitza el JepiFW com a template de HTML per a l'aplicació de gestió de tasques.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <!-- The above 3 meta tags *must* come first in the head; any other head content must
come *after* these tags -->
        <title><?= $title ?></title>

        <!-- Bootstrap -->
        <link href="/public/assets/css/bootstrap.min.css" rel="stylesheet">
        <link href="/public/assets/css/template.css" rel="stylesheet">

        <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries --
>
        <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
        <!--[if lt IE 9]>
          <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
          <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
        <![endif]-->
    </head>
    <body>
        <nav class="navbar navbar-inverse navbar-fixed-top">
            <div class="container">
                <div class="navbar-header">
                    <button     type="button"     class="navbar-toggle     collapsed"     data-
toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
                        <span class="sr-only">Toggle navigation</span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" href="#">To-Do Manager</a>
                </div>
            </div>
        </nav>

        <div class="container content-container">
            <?= $listsView; ?>
            <?= $tasksView; ?>
        </div><!-- /.container -->



        <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
        <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
        <!-- Include all compiled plugins (below), or include individual files as needed -->
```

```
        <script src="/public/assets/js/bootstrap.min.js"></script>
        <script src="/public/assets/js/scripts.js"></script>
    </body>
</html>
```

Annex 3. Captures de pantalla

Tot seguit es mostren algunes captures de pantalla relacionades amb l'aplicació que s'ha desenvolupat per exemplificar una aplicació desenvolupada amb el JepiFW.

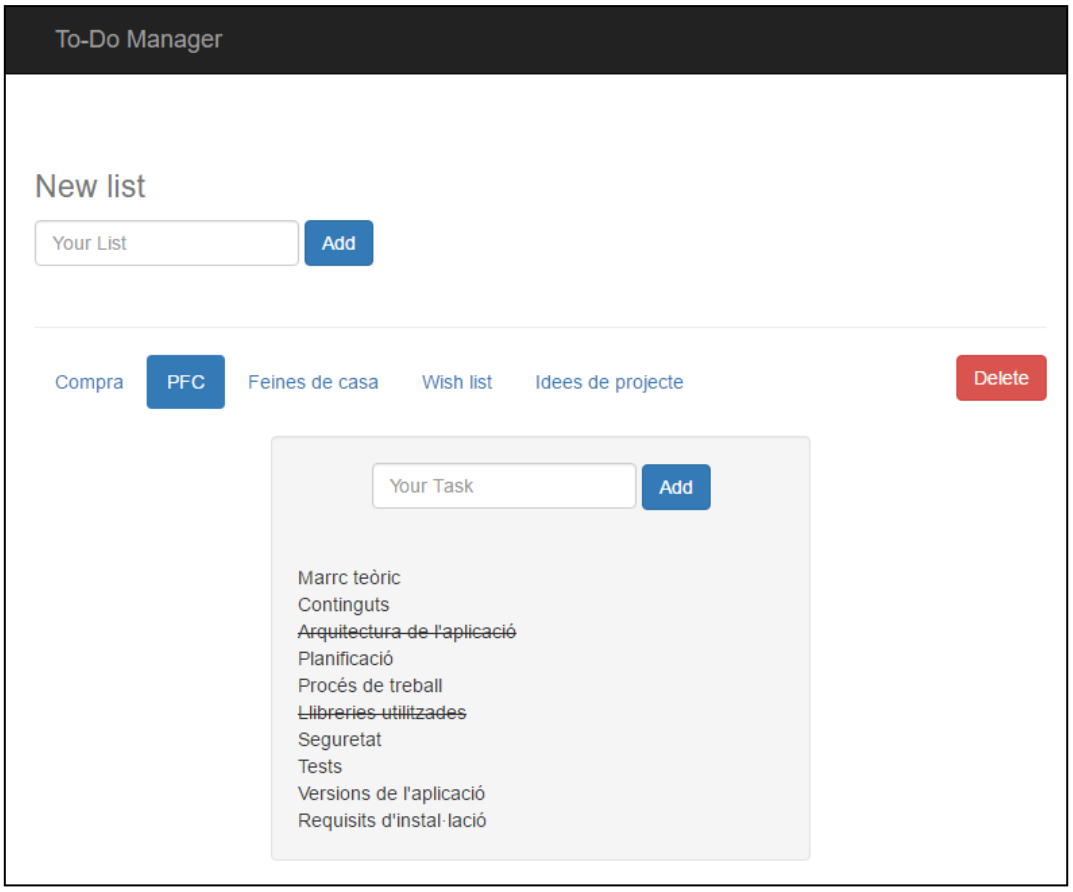

Figura 54 Vista general de l'aplicació de gestió de tasques desenvolupada amb el JepiFW.

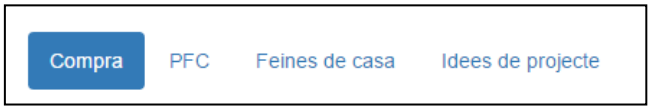

Figura 55 Selector de llistes de tasques ens permet saltar d'un llistat a un altre.

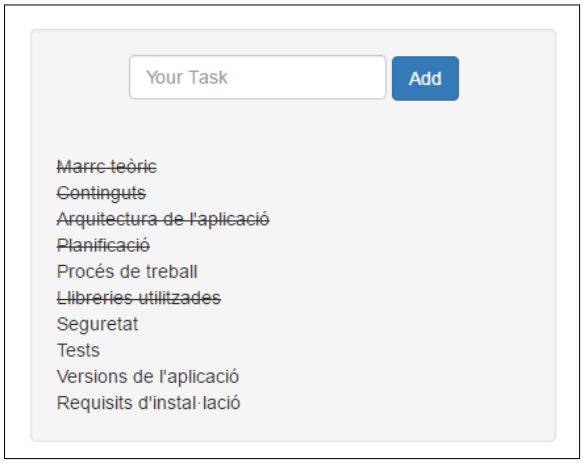

Figura 56 Tasques d'una llista amb input per afegir-ne més.

Annex 4. Glossari

Tot seguit es mostren els diversos acrònims utilitzats en el present document:

FW: *Framework.* És l'anglicisme utilitzat per referir-nos a un marc de treball. En el context d'aquest document es refereix al conjunt de codi que constitueix una eina completa per al desenvolupament.
MVC: Patró de disseny model-vista-controlador.
BD: Base de dades.
API: de l'anglès Application Programming Interface, és una interfície a través de la qual dues aplicacions poden interactuar entre elles.
IO: de l'anglès Input/Output, significa entrada/sortida i fa referència als canals d'entrada i sortida de dades d'una aplicació.
DI: de l'anglès Dependency Injection, és un patró de disseny de caràcter estructural que permet gestionar les dependències dels objectes d'una aplicació de forma automàtica.
XSS: de l'anglès Cross-Site Scripting. És un atac amb el qual s'altera el funcionament d'una aplicació mitjançant la inserció de codi a través dels canals d'entrada de la mateixa.
UI: de l'anglès User Interface, en català és la interfície d'usuari.
ORM: de l'anglès, Object Relation Mapping, en català és el mapeig objectes-relacional i és una tècnica de programació que converteix les dades de la base de dades en objectes. Consisteix en afegir una capa d'abstracció a la base de dades.

Annex 5. Bibliografia

Tot seguit es llisten totes les referències bibliogràfiques consultades per a dur a terme el projecte que aquest document explica. Donat que és un projecte de programació, tots els documents consultats són pàgines web especialitzades, blogs de professionals del món web, pàgines web d'empreses desenvolupadores de software o bé fòrums especialitzats.

Per organitzar aquest llistat s'ha fet seguint una agrupació segons el tema amb què està relacionat.

Frameworks existents i comparatives

The Best PHP *Framework* for 2015: SitePoint Survey Results
http://www.sitepoint.com/best-php-*framework*-2015-sitepoint-survey-results/

Performance benchmark of popular PHP *frameworks* – Systems Architect
https://systemsarchitect.net/2013/04/23/performance-benchmark-of-popular-php-*frameworks*/

Diagrama del flujo de una aplicación con CodeIgniter, por @davidvalverde
http://www.davidvalverde.com/blog/diagrama-del-flujo-de-una-aplicacion-con-codeigniter/

Hakeemmazinalanezikhaled.pdf
https://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/hakeemmazinalanezikhaled.pdf

Instalación de CodeIgniter, por @davidvalverde
http://www.davidvalverde.com/blog/instalacion-de-codeigniter/

CodeIgniter / EllisLab
https://ellislab.com/codeigniter

Design and Architectural Goals — CodeIgniter 3.0.6 documentation
http://www.codeigniter.com/user_guide/overview/goals.html

Homepage - Twig - The flexible, fast, and secure PHP template engine
http://twig.sensiolabs.org/

Symfony at a Glance
http://symfony.com/at-a-glance

PHP Standards Recommendations - PHP-FIG
http://www.php-fig.org/psr/

Symfony
http://symfony.com/legacy/doc/jobeet/1_2/es/01?orm=Doctrine

Framework - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/*Framework*

The 2015 Top Ten Programming Languages - IEEE Spectrum
http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

Diagrames UML

PlantUML : Open-source tool that uses simple textual descriptions to draw UML diagrams.
http://plantuml.com/

PlantUML - NetBeans Plugin detail
http://plugins.netbeans.org/plugin/49069/plantuml

PHP UML Generator - Stack Overflow
http://stackoverflow.com/questions/393603/php-uml-generator

UML basics: The class diagram
http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

Composer

Composer
https://getcomposer.org/

Los scripts de Composer
http://librosweb.es/tutorial/los-scripts-de-composer/

Building better project skeletons with Composer - Tutorial - Binpress
https://www.binpress.com/tutorial/better-project-skeletons-with-composer/157

Satis (repositori privat de paquets composer)

Setting Up a Local Mirror for Composer Packages With Satis - Envato Tuts+ Code Tutorial
http://code.tutsplus.com/tutorials/setting-up-a-local-mirror-for-composer-packages-with-satis--net-36726

Handling private packages with Satis or Toran Proxy - Composer
https://getcomposer.org/doc/articles/handling-private-packages-with-satis.md

Configuració de l'*autoloader* de composer

The composer.json Schema - Composer
https://getcomposer.org/doc/04-schema.md#autoload

PHP - Using Composers Autoload - Stack Overflow
http://stackoverflow.com/questions/12818690/using-composers-autoload

PHP: Autoloading Classes - Manual
http://php.net/manual/en/language.oop5.autoload.php

Views Management

Presentation Model
http://martinfowler.com/eaaDev/PresentationModel.html

Model-View-Confusion: In MVC the View gets its own data from the Model
https://r.je/views-are-not-templates.html

Homepage - Twig - The flexible, fast, and secure PHP template engine
http://twig.sensiolabs.org/

Integrating Twig in your legacy PHP code — Alessandro Nadalin
http://odino.org/integrating-twig-in-your-legacy-php-code/

Patrons de disseny

PHP Master | The MVC Pattern and PHP, Part 1
http://www.sitepoint.com/the-mvc-pattern-and-php-1/

Design Patterns - PHP: The Right Way
http://www.phptherightway.com/pages/Design-Patterns.html

DesignPatternsPHP — DesignPatternsPHP 1.0 documentation
http://designpatternsphp.readthedocs.org/en/latest/

PHP-DI - The Dependency Injection Container for humans
http://php-di.org/doc/*frameworks*/zf1.html

Observer (patrón de diseño) - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)

Modelo–vista–controlador - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

Ejemplo PHP + POO + MVC - Victor Robles | Victor Robles
http://victorroblesweb.es/2014/07/15/ejemplo-php-poo-mvc/

Design Patterns - PHP: The Right Way
http://www.phptherightway.com/pages/Design-Patterns.html

What is Dependency Injection? | Articles - Fabien Potencier
http://fabien.potencier.org/what-is-dependency-injection.html

Practical PHP Patterns: Front Controller - DZone Web Dev
https://dzone.com/articles/practical-php-patterns/practical-php-patterns-front

MVC Applications — documentación de Phalcon - 2.0.8
https://docs.phalconphp.com/es/latest/reference/applications.html

Design patterns used in symfony2 - Stack Overflow
http://stackoverflow.com/questions/13696059/design-patterns-used-in-symfony2

Design Patterns in PHP
http://forum.codeigniter.com/thread-507.html

Domnikl/DesignPatternsPHP · GitHub
https://github.com/domnikl/DesignPatternsPHP

MVC In PHP, A real world example - Tom Butler
https://r.je/mvc-tutorial-real-application-example.html

Start your own MVC *Framework* with PHP ~ Elvis is still in the building
http://www.elvishsu66.com/2014/01/start-your-own-mvc-*framework*-with-php.html#.VjOxza4vckg

Controlador frontal d'un *framework*

An Introduction to the Front Controller Pattern, Part 2
http://www.sitepoint.com/front-controller-pattern-2/

The Front Controller - Zend_Controller - Zend *Framework*
http://*framework*.zend.com/manual/1.12/en/zend.controller.front.html

Write your own PHP MVC *Framework* (Part 1) | anant garg
http://anantgarg.com/2009/03/13/write-your-own-php-mvc-*framework*-part-1/

PHP Master | An Introduction to the Front Controller Pattern, Part 1
http://www.sitepoint.com/front-controller-pattern-1/

Injecció de dependències

PHP-DI - The Dependency Injection Container for humans
http://php-di.org/doc/best-practices.html

Dice - PHP Dependency Injection Container
https://r.je/dice.html

PHP Dependency Injection Container Performance Benchmarks
http://www.sitepoint.com/php-dependency-injection-container-performance-benchmarks/

Dependency Injection in PHP. Create your own DI container.
http://krasimirtsonev.com/blog/article/Dependency-Injection-in-PHP-example-how-to-DI-create-your-own-dependency-injection-container

Dependency Injection in PHP - Tuts+ Code Tutorial
http://code.tutsplus.com/tutorials/dependency-injection-in-php--net-28146

PHP-Dependency/library/Pd at master · ryanto/PHP-Dependency · GitHub
https://github.com/ryanto/PHP-Dependency/tree/master/library/Pd

PHP Master | Dependency Injection with Pimple
http://www.sitepoint.com/dependency-injection-with-pimple/

Level-2/Dice · GitHub
https://github.com/Level-2/Dice

Aura for PHP : Aura.Di
http://auraphp.com/packages/Aura.Di/

Dependency Injection Containers with PHP. When Pimple is not enough. | Gonzalo Ayuso | Web Architect
http://gonzalo123.com/2012/09/03/dependency-injection-containers-with-php-when-pimple-is-not-enough/

PHP-DI - The Dependency Injection Container for humans
http://php-di.org/

Inversion of control - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Inversion_of_control

Playing with dependency injection in PHP
http://coderoncode.com/dependency-injection/design-patterns/programming/php/development/2014/01/06/dependency-injection-php.html

PhpUnit

PHPUnit Manual – Chapter 1. Installing PHPUnit
https://phpunit.de/manual/current/en/installation.html#installation.composer

Gestió d'entrada/sortida de dades i seguretat

http-foundation/Response.php at master · symfony/http-foundation · GitHub
https://github.com/symfony/http-foundation/blob/master/Response.php

Output Class : CodeIgniter User Guide
https://ellislab.com/codeigniter/user-guide/libraries/output.html

Symfony2 Application Security Guidelines | Arts & Sciences Computing
https://www.sas.upenn.edu/computing/infosec_symfony2

Cross-Site Scripting Attacks (XSS)
http://www.sitepoint.com/php-security-cross-site-scripting-attacks-xss/

Mysql - How can I prevent SQL-injection in PHP? - Stack Overflow
http://stackoverflow.com/questions/60174/how-can-i-prevent-sql-injection-in-php

Altres

How To Setup PhpStorm, Xdebug, and MAMP for Debugging
http://manovotny.com/setup-phpstorm-xdebug-mamp-debugging/

Draft-fielding-http-spec-01 - Hypertext Transfer Protocol -- HTTP/1.0
https://tools.ietf.org/html/draft-fielding-http-spec-01

Y 6 *frameworks* PHP más que te harán la vida más simple
http://www.genbetadev.com/*frameworks*/y-6-*frameworks*-php-mas-que-te-haran-la-vida-mas-simple

Building an ORM in PHP - devshed
http://www.devshed.com/c/a/mysql/building-an-orm-in-php/

PHP - Listing all directories and sub-directories recursively in drop down menu - Stack Overflow
http://stackoverflow.com/questions/14304935/php-listing-all-directories-and-sub-directories-recursively-in-drop-down-menu

Building a Custom PHP *Framework* with a custom template caching engine using Output Control functions | Code with Music
http://abhinavsingh.com/building-a-custom-php-*framework*-with-a-custom-template-caching-engine-using-output-control-functions/

TDD
https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas