# A UI for the Open Movie Database

The purpose of this document is to provide a brief overview of the application, to explain how the code works and how it was written, and to give instructions on how to deploy it.

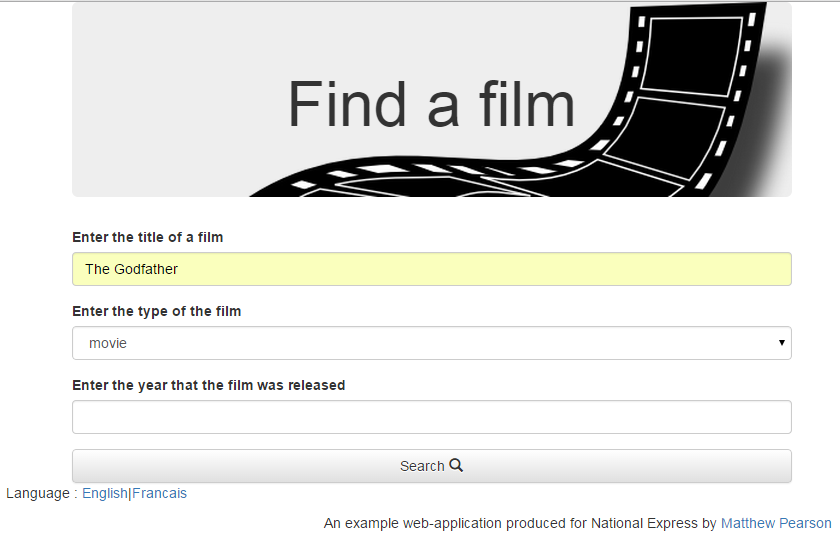
The application is currently accessible at

[http://ec2-52-16-64-134.eu-west-1.compute.amazonaws.com:8080/filmfinder/](http://ec2-52-16-64-134.eu-west-1.compute.amazonaws.com:8080/filmfinder/%20)

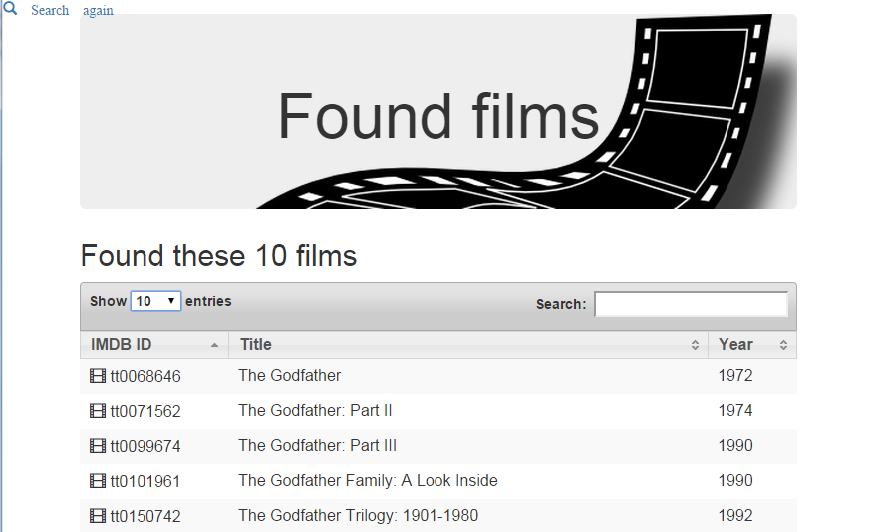
deployed on Amazon web services.

## Functional description

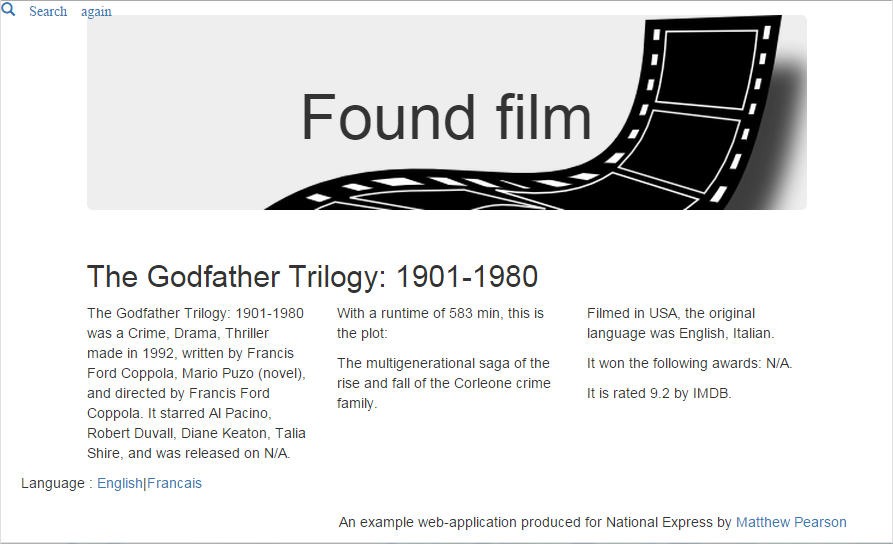
The intention is that its use should be intuitive. The opening page presents the user with a form containing just three fields: title, type and year; and a search button.



Clicking the search button will, assuming the form is valid, take the user to the second page, displaying the search results. These are shown in a table. If there are many results, then they are paged. You’re unlikely to see this behaviour, though, because OMDb only returns a maximum of ten results.



On double-clicking a row in the table, the third and final page is shown. This details various facts about the film, as provided by OMDb, and displays the poster if one is available, as well as the linking to the Wikipedia page for that film.



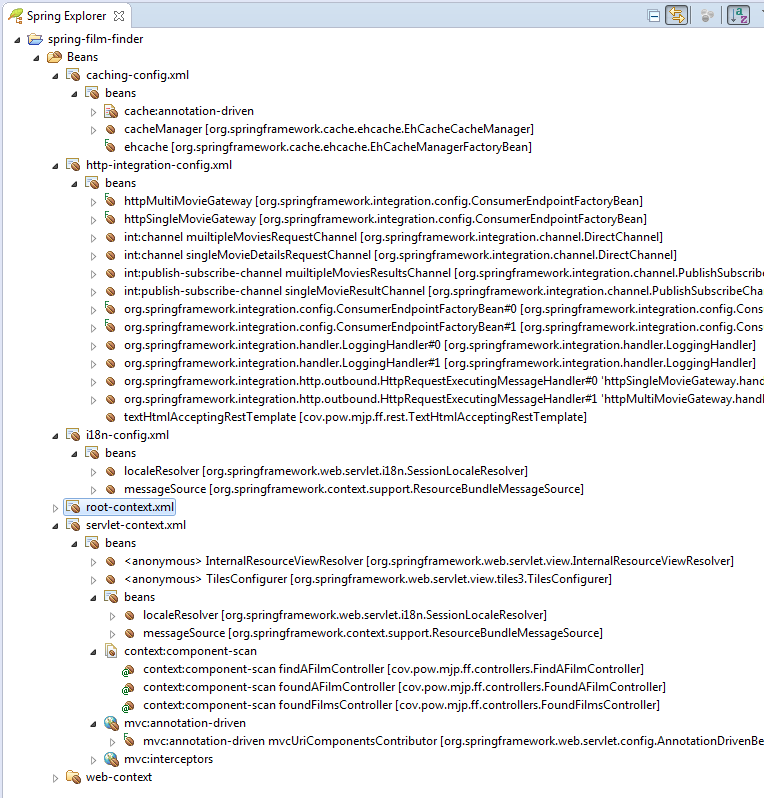
## Frameworks used

Most processing is done on the server in Java using the Spring framework. Client-side, Jquery and Bootstrap are used.

As a general rule, I believe that programming frameworks should be used wherever practicable, because they reduce the costs of both development and maintenance. The less code you write, the fewer bugs you can write. Using well-known frameworks, as well as well-known programming and design patterns, should make code written by one developer more easily understood by another. Using frameworks to provide standardised views makes the UI components of the application more instantly recognisable and familiar to end-users, so reducing the time and effort they have to expend in learning to use it.

### Server-side: Spring modules used

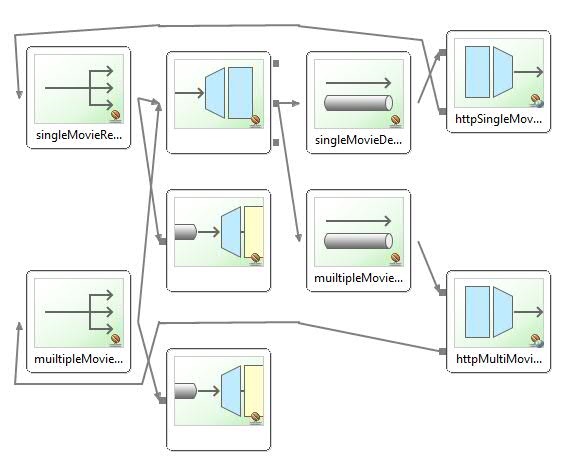
The screenshot below show the 5 spring-config files, and the beans each of them defines.



Tomcat serves web-pages generated dynamically by Spring using the JSP and Tiles view technologies. The requests are mapped to the views at system start-up, as defined by annotations on the controllers (component-scanning). This concern is mainly configured in servlet-context.xml, which also has an interceptor to discover what type of device the client is – desktop, tablet, or mobile. Most of the form validation is done using HTML5, but Spring validates it too. This provides a fall-back on older browsers.

The pages come in a choice of two languages: English (the default) and French. French will be used if the browser has set the request header accept-language to *fr*. The user can also opt to change language by clicking on a hyper-link in the page. This is configured in *i18n-config.xml*.

Communication between this web-app and the OMDb web-app is handled by beans defined in *http-integration-config.xml*. This wires up components provided by Spring that have been designed according to standard EAI patterns (Enterprise Application Integration Patterns).



Eclipse didn’t make a very good job of this integration bean graph, so I’ll explain what’s happening: When the user clicks to search, a request comes into tomcat, which passes it on to the dispatcher servlet which in turn decides which controller should handle it (the findAFilmController). The controller asks the movieDbService for the search results. The service interfaces with the EAI module, requesting the results from a gateway, and then an http message handler actually communicates with the external web-app. The result then makes the reverse journey, and is published both to the original service and to a logger. Everything is linked together with channels, and the messages are seamlessly converted from Java objects to JSON and back again.

Finally, *caching-config.xml* wires up an instance of EH Cache, so that if the same terms are entered for a search, or the same movie details requested, even by different clients, then only one request for the data is made to OMDb.

### Client-side: JQuery and Bootstrap

JQuery abstracts away much of the low-level logic, and in my experience, encourages developers to write cleaner and more modular Javascript programs. Bootstrap, which I am less familiar with, appears to extend these advantages to CSS.

The form on page one is laid out according to Bootstrap conventions; page two’s table is a *datatable* combining both Jquery and Bootstrap; and page three is built with Bootstraps grid template.

Bootstrap makes writing a UI that will be used on different types of device, with different screen sizes, much easier. Spring can also determine the client device, and amend JSPs accordingly. I have included some different css and javascript files in the JSPs depending on the client screen-size.

## Build

The build is done by Gradle. This takes away much of the pain of sourcing the application’s dependencies, and was particularly useful in resolving clashes where multiple dependencies depended in turn on different version of the same jar. By assuming build conventions, gradle scripts tend to be much smaller, cleaner, and easier to understand than those written with ant of maven.

## Deployment

Simply put the war file onto your file system, and point to it from a *filmfinder.xml* file like this, placed in CATALINA\_HOME/conf/Catalina/{your\_host}

<Context docBase="{path\_on\_your\_file\_system}/filmfinder-0.1.war" path="/filmfinder" />

Substituting for {your\_host} and {path\_on\_your\_file\_system} with the appropriate values for your system.

Then start up tomcat using its start-up script in its bin directory.

Alternatively, you can deploy the war file to a live system once tomcat is running by using the tomcat manager GUI web-app.

In the unlikely event that anything should go wrong, log files are written to the usual directory, CATALINA\_HOME/logs. The application uses log4j, with the configuration file sitting in the source tree under main/resources.

## Testing

As I’ve managed to delegate most of the logic to 3rd party libraries, there isn’t much left to unit test. Checking that everything has been correctly wired together is best done with integration tests and by a live tester.

I had to extend Spring’s RestTemplate, which translates http JSON responses into Java objects, because OMDb declares its JSON responses to be text/html in the header. So I wrote unit tests for that. They could be more lightweight if they mocked the http response. They actually request data from OMDb, so perhaps they are integration tests really.

I did initially start writing unit tests for the controllers (which again, are pseudo integration tests, because they use Spring to test the wiring), but when I introduced mobile device detection I gave up. I don’t think Spring MVC Test has the capability to mock this yet.

## Known bugs

Some poster images (maybe all) fail to display when the app is running on Amazon Web Services:

Referral denied. You don't have permission to access XXX on this server.

When running in small-screen mode, the page redirecting doesn’t work properly (at least sometimes), and the JQuery document.ready() function isn’t invoked. This means that the onClick() function doesn’t get added to the table rows on page 2, so it’s impossible to advance to page 3.

## Source code

The source is available for your perusal on GitHub:

<https://github.com/matthew-pearson/National-Express-OMDb-Demo>

but it’s probably easier to read it if you import it as a project into Eclipse (preferably a Spring Tools Suite version). If you don’t load it into an IDE, then this diagram of the file structure may make it easier to figure out.

