SIB VISIONS
simplify IT business

Java Framework comparison

Table of contents

Introduction

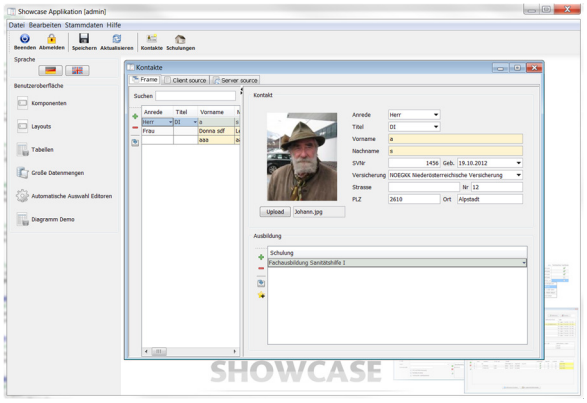
In the last month we've started a big research project. It is a comparison of Java frameworks for backend and frontend development. We knew that there are many different frameworks for the same job, but which one is the best? Is there one framework that would outpace all others? What are the pros and cons of each framework and how fast could we develop with them?

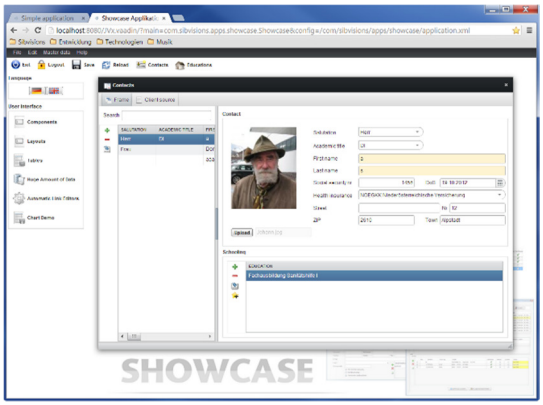
It was important for us to start our research project from scratch, without templates and without prior knowledge of the specific frameworks. We planned a two iteration process to get a simple learning curve. The first iteration should show the progress without prior knowledge of the used technology and the second iteration, by the same developer, should show the result with experience from the first iteration.

But first, we outlined the requirements for the framework, application and process:

- Create a (relational) database application with prespecified edit page/screen
- Use a Master/Detail relation (n:m)
- Direct editing in tables
- A detail form for the Master (edit/insert functionality)
- Show/Edit an image
- CSV Export of Master table records
- Standard CRUD operations for Master and Detail
- Use the best tool-chain for development (IDE, ORM, JPA, AddOns, …)
- Default layouting, no specific styling
- Fremwork must be Free Software/Open Source or free for commercial use (without license fees)

We planned to create a simple Contacts management UI as we knew it from our showcase or demo applications. Here are two screenshots from Swing and Vaadin UI:





It wasn't important for our project, to have a full application frame with menu and toolbar or authentication. Only the edit screen was subject of our project.

The screen has two grids. The left one shows all available contacts and the right one all available educations for the selected contact. The detail form contains editors for all contact information with

date editor, comboboxes and an image. It is possible to export all contacts as CSV file. Both grids support CRUD operations and inline editing. The screen itself is not too complex and doesn't have much business logic, but it has all elements which are relevant for our research project. Of course, it could be a screen from a real-world ERP system.

After the general project description, we tried to find the right frameworks for US. We found some nice reports about „top web frameworks" and general technology trends:

http://vitalflux.com/java-top-10-java-based-web-development-frameworks-2014-2015/

http://blog.spec-india.com/top-java-frameworks-today

and older:

http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/

http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/

We made our own list as well. The important frameworks for us were:

- PrimeFaces
- ADF/ Essentials
- Apache Wicket
- Play Framework
- AngularJS
- Vaadin
- GWT
- GXT
- jquery/ui
- JVx

We didn't choose all frameworks and our final candidates were:

- JSF with PrimeFaces
- JSF with ADF/ Essentials
- Vaadin
- JHipster
- JVx

Why not Wicket, Play and all the others?

We tried to use the best candidates from the list of „top web frameworks" and JVx, because it's the only cross-platform, single-sourcing solution with Swing, JavaFX, vaadin and mobile support that we know of. Sure, it's developed from SIB Visions and we were interested in direct comparison with keyplayers.

JHipster was cool because it does create applications based on Spring Boot and AngularJS and it does create a real application frame. It's more an application builder than a framework, but it was perfect for our project.

GWT is included in Vaadin and we thought Vaadin is the better choice for us. GXT didn't meet the license restrictions.

We knew many other frameworks like Struts 2, Flex, JSpresso, Pivot, pure Swing, pure JavaFX but some are covered by JVx and Struts 2 was not top anymore.

The project start

The start wasn't as easy as expected because we needed some „fresh" developers without prior knowledge of the selected technologies, but with similar developer know how. Not every developer has the same development speed, so we tried to find a mix of different developers.
The final team of developers was composed of some of our own and students. We were finally ready to start.

Each developer had to do accurate time recording and documenting exactly what has been done. It was important to know the tasks that took the most effort. We didn't define anything for a specific technology because the developer should start learning as practical as possible.

After all tasks were finished for one technology, the developer gave a presentation and talked about pros, cons, problems and their personal opinion.

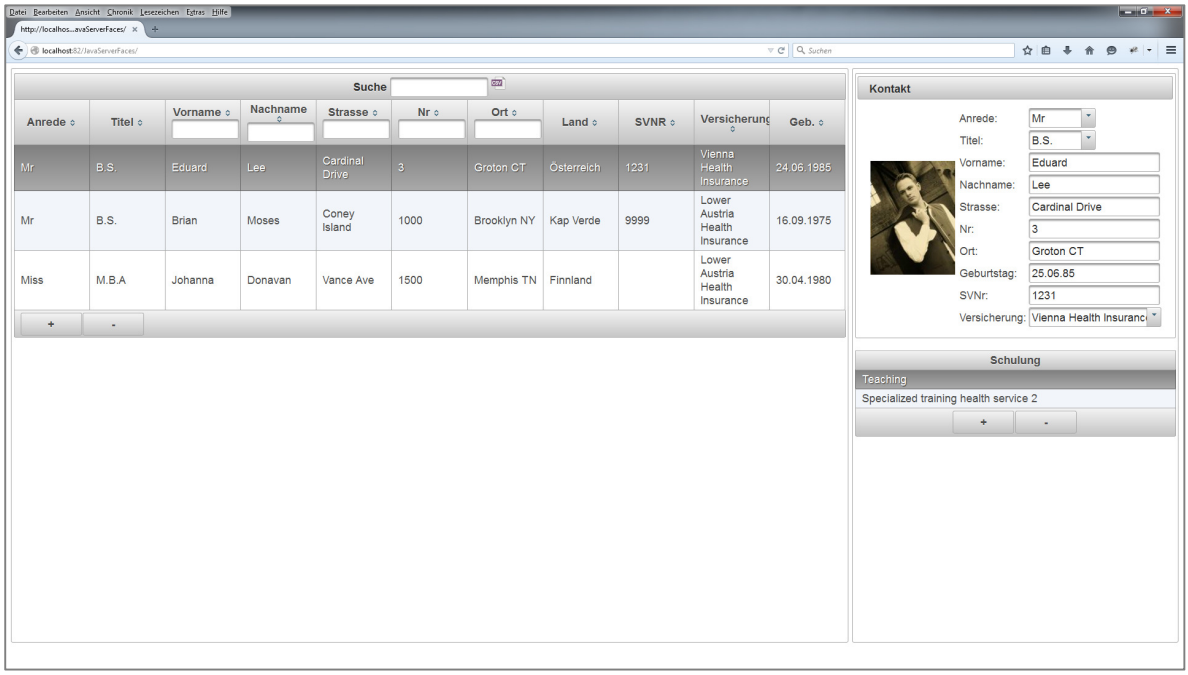
The result

It was awesome to see so many different frameworks in action and to get a presentation from different developers with different knowledge and different experience. The personal opinion was cool as well because it wasn't a marketing message and we didn't tell anyone what to say.

So, let's start with the summary for every technology.

Framework comparison

JSF (PrimeFaces)



Used Tools: Eclipse Luna for Java EE
Used Libraries: PrimeFaces, Hibernate, Spring

Required Time
First run: 20 hours
Second run: 11 hours, 5 minutes

Tasks

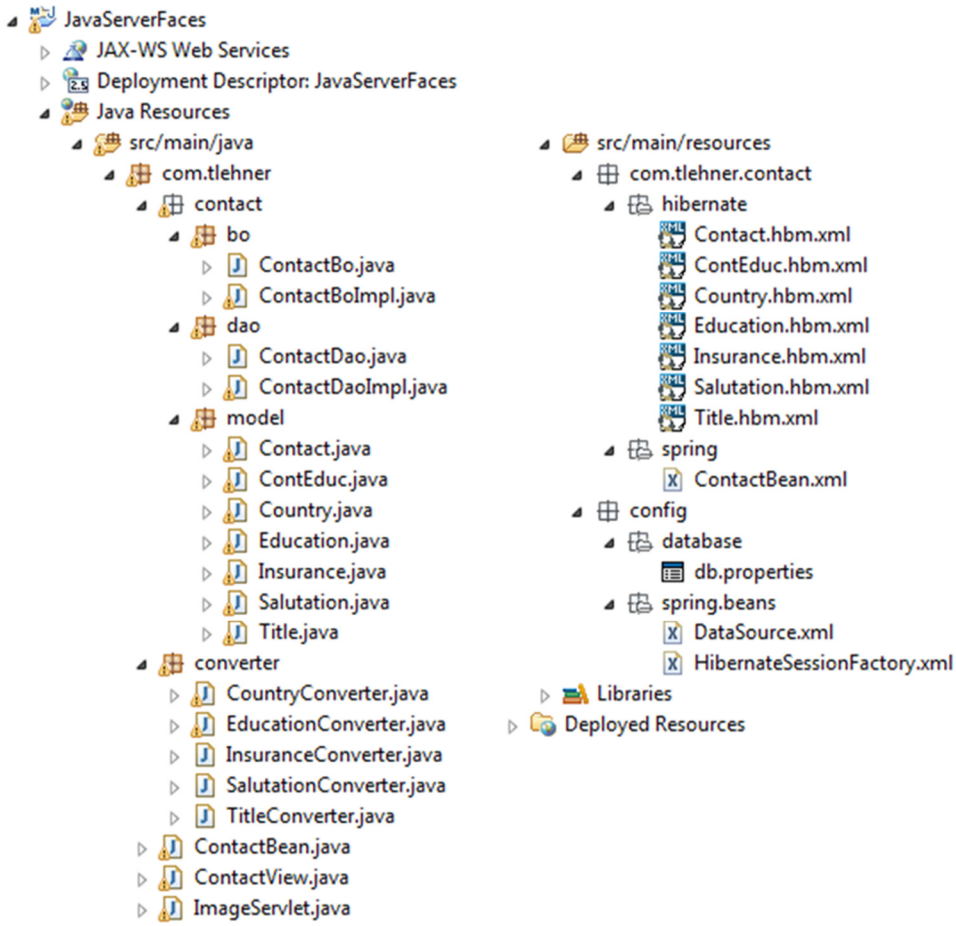
- Create database, tables
- Insert test data
- Configure maven
- Create a simple demo project (only for first run)
- Create and Configure Contacts project
- Create Beans, Mappings, Converter
- Implementation of sort, filter, CRUD, Servlet for Image

The first run took 20 hours because the developer didn't know the technology before and started with everything from scratch, with much help from various search engines and technology communities on the web. The demo project was used for experiments with the technology and as playground for testing different features.

The second run was started five days after the first run and represents the result of an experienced developer (it was the same developer as in the first run). The developer didn't re-use anything from the first run and started from scratch again.

The big difference was that the developer could skip a demo project and working was overall faster because there was no need to search the Internet for help. We're sure that a developer with many years experience can solve all tasks faster, but our guess is that it will be around one working day.

Implementation overview



To implement the screen, the developer had to use 19 source files. The programming model needs source files for: Model class, Business Object Interface (for Dependendy Injection), Business Object Interface implementation, DAO Interface (for Dependency Injection), DAO Interface implementation, Contact bean, Contact view.
And also model classes for relations and combobox editors: ContEduc, Country, Education, Insurance, Salutation, Title.
For every combobox editor, a converter was needed: CountryConverter, EducationConverter, InsuranceConverter, SalutationConverter, TitleConverter.
And last but not least, a Servlet was needed for the contact image because it wasn't possible to show an image as simple byte[]. Only URLs were supported from the technology.

Conclusion from the developer

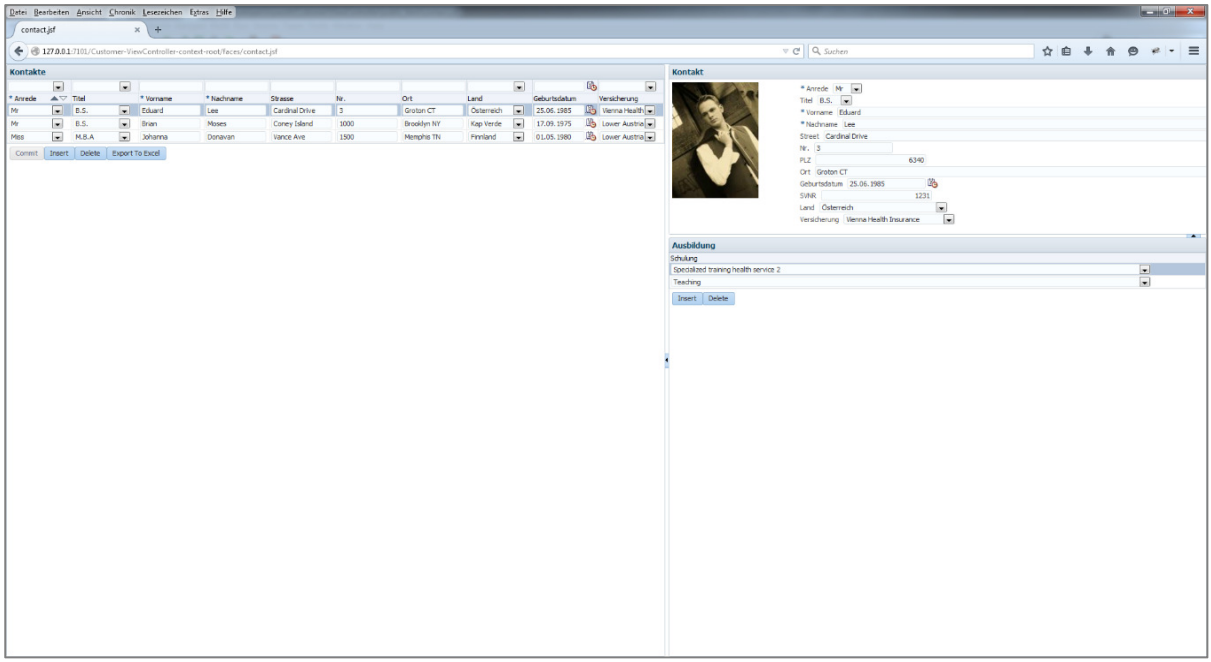
The start with hibernate was hard because of complex mappings and I didn't use annotations instead of XML mapping files. But after some hours it was easy to handle.
PrimeFaces has a lot of ready-to-use components with many features but this makes the handling and usage relatively complex because the html tag definition grows with every option. There are

many showcases and examples for PrimeFaces. If you need help for Hibernate or PrimeFaces, it's no problem because most problems were already answered on stackoverflow or similar services.

Pros: Many components,
Model classes are easy to re-use,
Components are easy to configure

Cons: GUI becomes complex fast,
Many files for just one screen,
Manual mapping is boring,
Many things aren't included: Converter, Image servlet

JSF (ADF/ Essentials)



Used Tools: Oracle JDeveloper 12c
Used Libraries: Oracle ADF 12

Required time
First run: 16 hours
Second run: 10 hours, 15 minutes

Tasks

- Create database, tables
- Insert test data
- Create a simple demo project (only for first run)
- Create and Configure Contacts project
- Create Models and Views
- Implementation of sort, filter, CRUD, Servlet for Image

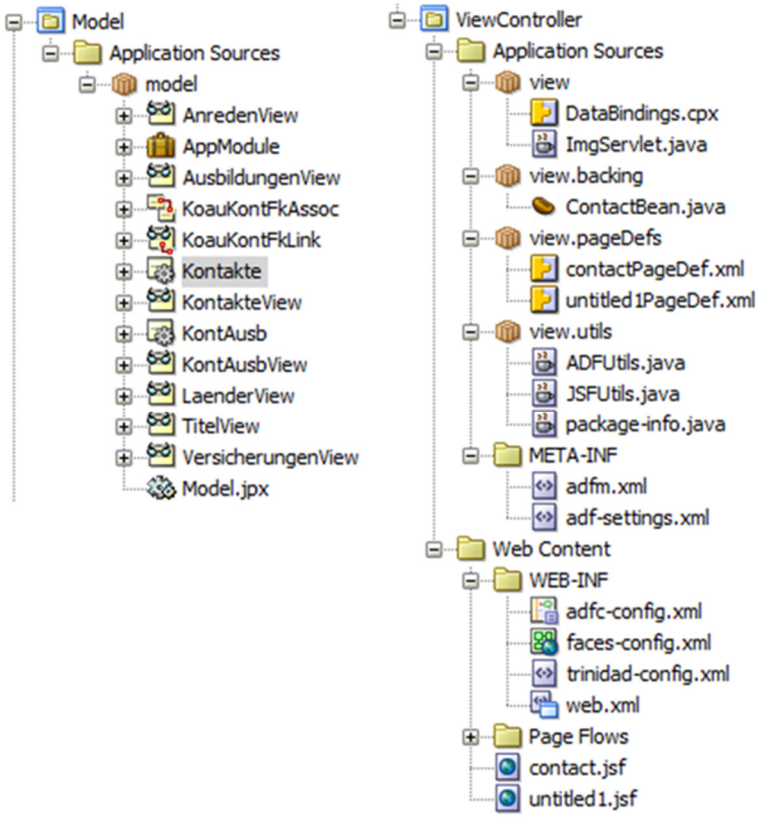
The result of the first run was comparable to the first run with JSF (PrimeFaces) because the developer didn't know the technology before and had started with a demo project to learn some basics.

The second run was started five days after the first run and represents the result of an experienced developer (it was the same developer as in the first run). The developer didn't re-use anything from the first run and started from scratch again.

So the development time of one and a half day is a good result for our contact screen. It was faster than JSF with PrimeFaces. But why? It was because of JDeveloper and the better integration of ADF.

Also ADF doesn't need mapping files or converters. It has more built-in features, but lets see some implementation details.

Implementation overview



To implement the screen, the developer had to use only 5 source files but 14 resource/configuration files. All files were created through JDeveloper Wizards. There's no need for mapping files or converters. ADF has many utilities out-of-the-box.
ADF also needs special handling for images. A servlet was created that returns the contact image for an URL.

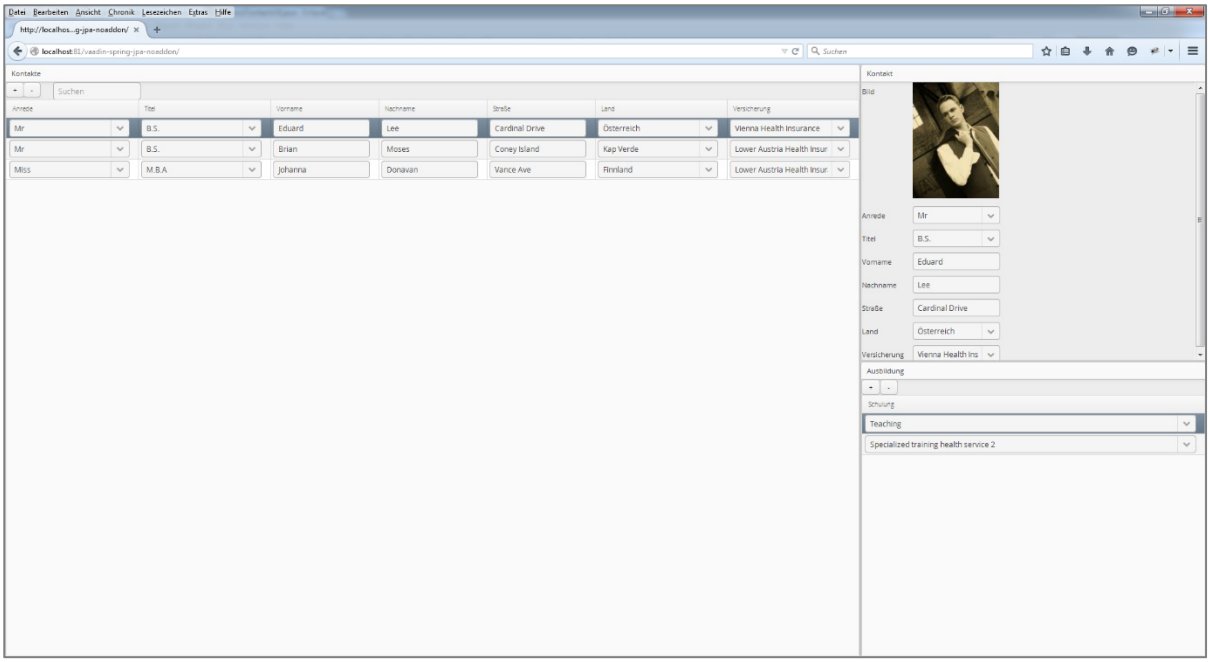
Conclusion from the developer

It's very easy to create a screen with ADF and the JDeveloper' ADF integration is great. But I didn't hear about JDeveloper before. It's not one of the top IDEs, isn't it?
By the way, the ADF handles everything and there was no need for manual database handling. ADF comes with many components and JDeveloper has a very useful GUI Designer. All standard features worked without problems: Master/Detail, Form view, CRUD. It was tricky to configure dropdowns because there were 5 different places with configuration options but only one was right. The documentation is available but mostly videos. It's sometimes very hard to find specific help because all non trivial problems aren't solved and aren't answered with docmentation. I found similar things after long search.
I had a great start but customizing wasn't fun because of too many resource/xml files. I prefer Java source code instead of configuration files.

Pros: Automatic database handling,
Fast start,
JDeveloper works great with ADF

Cons: High complexity after fast start and hard to learn,
Missing documentation and available Videos weren't helpful,
ADF itself is complex and you don't know what's happening behind the scene

Vaadin



Used Tools: Eclipse Luna for Java EE
Used Libraries: Vaadin 7.4, Spring, Hibernate

Required time
First run: 14 hours, 15 minutes
Second run: 9 hours, 50 minutes

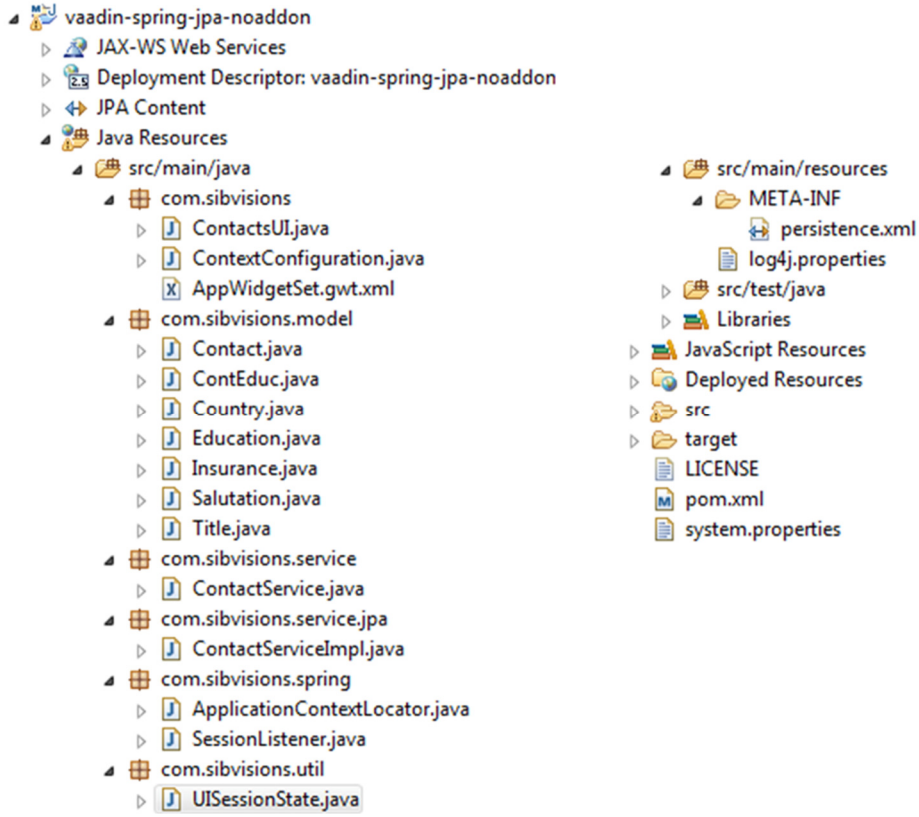
Tasks

- Create database, tables
- Insert test data
- Create a simple demo project (only for first run)
- Create and Configure Contacts project
- Create Beans and Mappings
- Create Models and Views

The first run was faster than all other first runs. This might be because vaadin is component based and not like other web frameworks. It's comparable to swing or JavaFX. Sure, the difference of 2 hours does not sound serious, but a 10% time saving can be important in production environments.

The second run shows that 10 hours are a good time for the screen. We don't know if it's possible in less than one day even for a Vaadin expert.

Implementation overview



The solution has 14 source files and 7 are model classes. One file for every entity. All model files were created manually. There's a service interface ContactService (for Dependency Injection) and an implementation (ContectServiceImpl) for the interface. And also Spring needs additional files for session handling (SessionListener) and for managing the spring context (ContextConfiguration). The screen itself is only one source file (ContactsUI) and that's it.

Conclusion from the developer

Vaadin feels simple and is easy to use. Simply use an archetype and you're ready to go. No html/javascript code was involved, only Java. This was nice. I had to use Spring and Hibernate for accessing the database. This wasn't a real problem because there's much documentation for vaadin, spring and hibernate. The documentation, the book, of vaadin is great. The amount of components is manageable but enough for my tasks. I saw a long list of AddOns for vaadin, but didn't use one. The support for vaadin is great because there's a forum and many Q&A on stackoverflow.

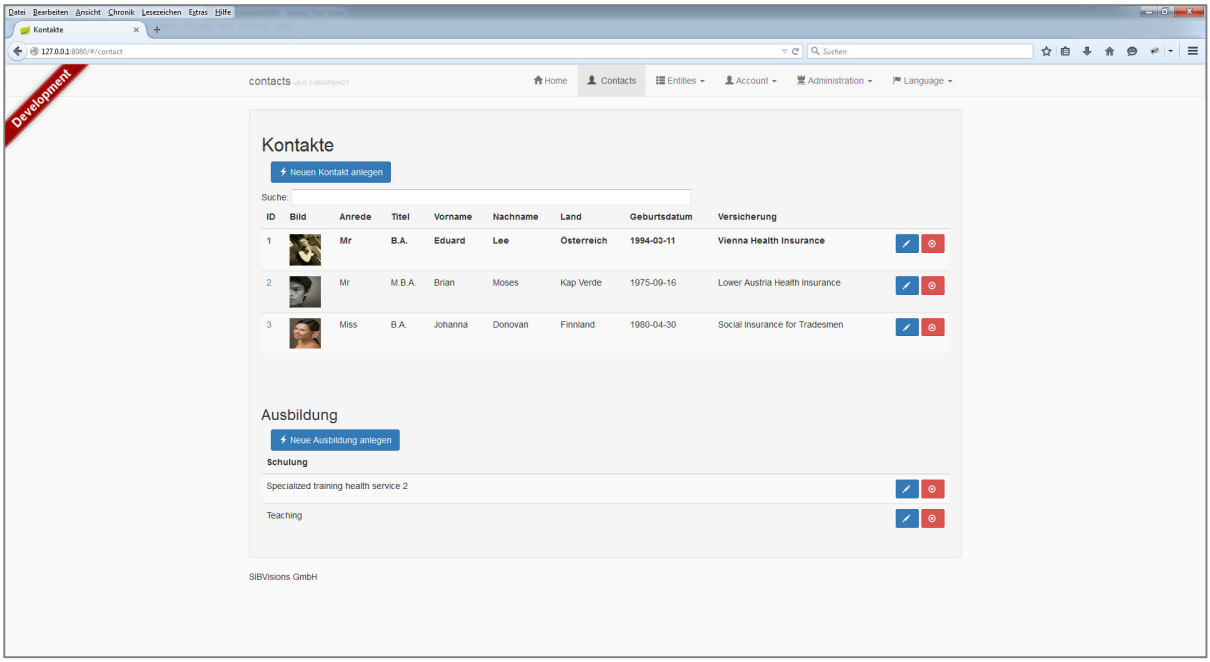
Pros: Flat learning curve,
Pure Java,
Very polished,
Great documentation,
Only one file for the GUI code

Cons: Much boilerplate code,

No support for detail forms or master/detail handling,
Implementation of custom row editors

JHipster

JHipster is different to all previous frameworks because it's an application generator and uses different open source frameworks to generate an application.



Used Tools: Spring Tool Suite, Yeoman, Node.js, Grunt, Bower
Used Libraries: Spring Boot, AngularJS, many others

Required time
First run: 18 hours, 50 minutes
Second run: 11 hours, 20 minutes

Tasks

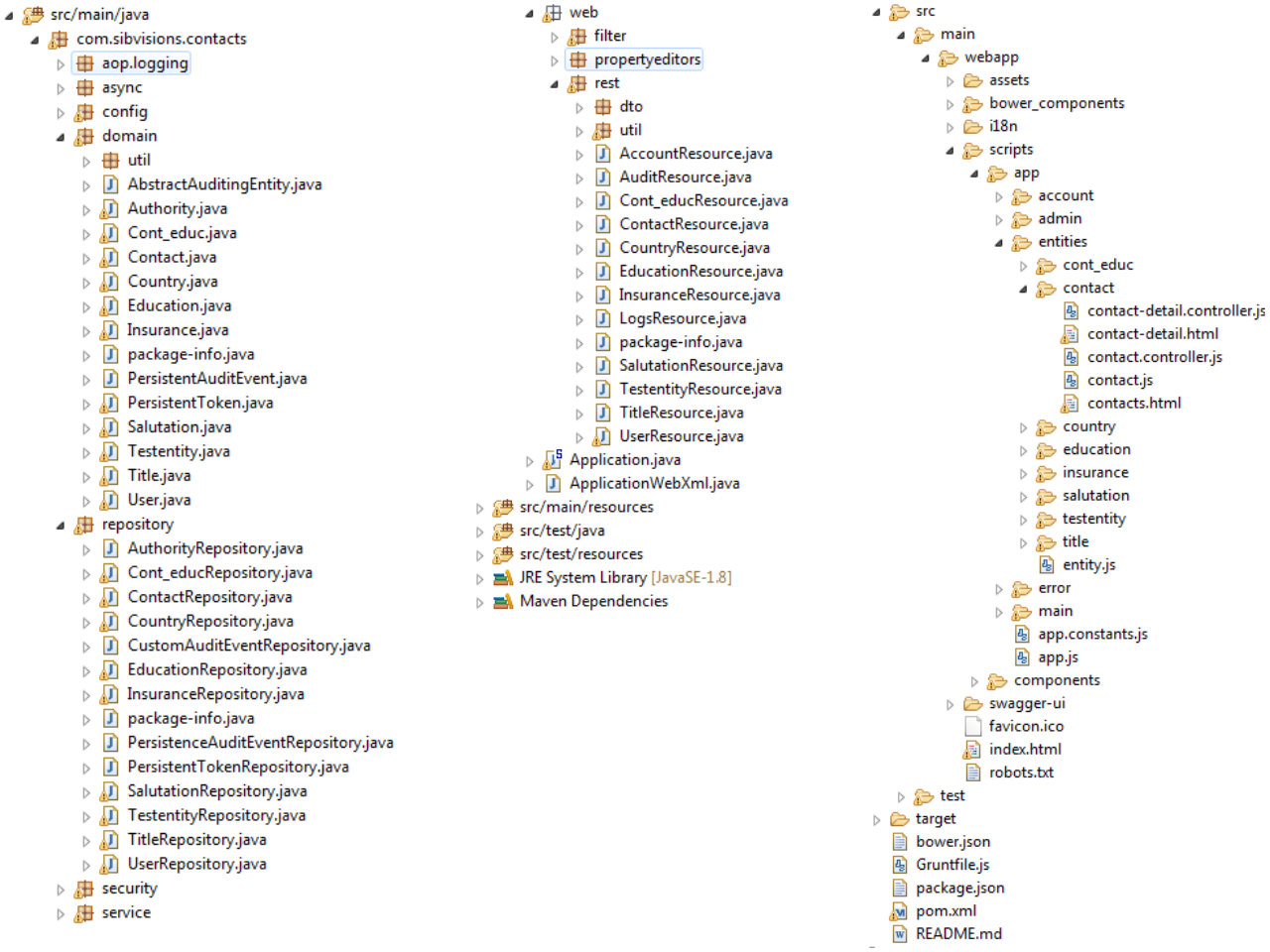
- Create database, tables
- Insert test data
- Create and Configure Contacts project
- Generate Models and Views

The first run took round about 19 hours. This is much for an application generator. The problem was the technology mix. The IDE is Eclipse based and also not standard because it's fully customized for Spring development. The developer didn't create a test application! He did use only one application for tests and development. The time was lost during development of defined features, because the generated code was a lot. The amount of tasks was not as much because the generator did everything for the developer. Also the tool mix was a problem because the command line was used for model generation. It was a lot of work to enter all relevant information.

The second run saved about one day. The difference was that the developer knew the directory structure and it was easier to find the right file and to use the command line tools without problems.

So the result is in the range of all other second runs.

Implementation overview



There are 63 source files, but only 21 were relevant, and 5 important configuration files. The whole project had more than 28.000 files (grand total). Every model file like Contact.java had an additional repository file (ContactsRepository.java) and a resource file (ContactResource.java). The resource file handled REST calls.
The view was generated too, but one view had different javascript files (contact.js, contact.controller.js) and the html template (contacts.html).

There are many files for one screen, but JHipster generates a full application with user authentication, a menu and different views for lists and edit forms.

Conclusion from the developer

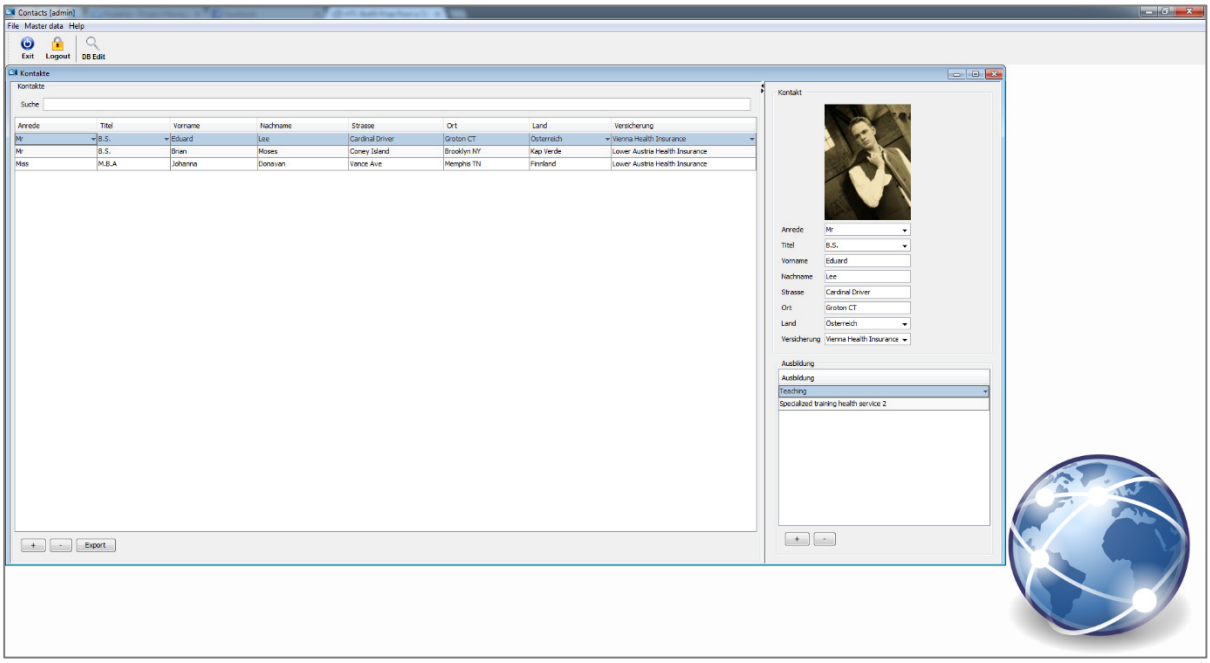
JHipster is hip but horrible because it generates a new Internet and not a simple application. The application itself is cool because it has everything out-of-the-box. It has user authentication, a menu, supports navigation and offers a nice standard style. The dropdowns work without additional work but only in detail forms.

It becomes complicated if you need additional features or the default settings aren‘t right. The directory structure is complex and you heavily use the search function of the IDE.
This solution is useful if you’re happy with „standard“ but if you want a customized application, hands off. I had problems to find the right files after 5 days break, so what will happen after 1 month?

Pros: Generates a complete and ready-to-use backend application,
Useful tools for model generation,
Nice standard look and feel,
Full application frame (authentication, menu, …)

Cons: Installation needs much time because of many different tools (yeoman, bower, node),
Reverse engineering of database tables is not possible,
A lot of work to apply changes,
It’s hard to find the right file(s),
Limited to MySQL and PostgreSQL,
Complex application structure,
Not easy to unterstand the different folders,
Customization is horrible

JVx



Used Tools: Eclipse Luna for Java EE
Used Libraries: JVx

Required time
First run: 8 hours
Second run: 2 hours, 30 minutes

Tasks

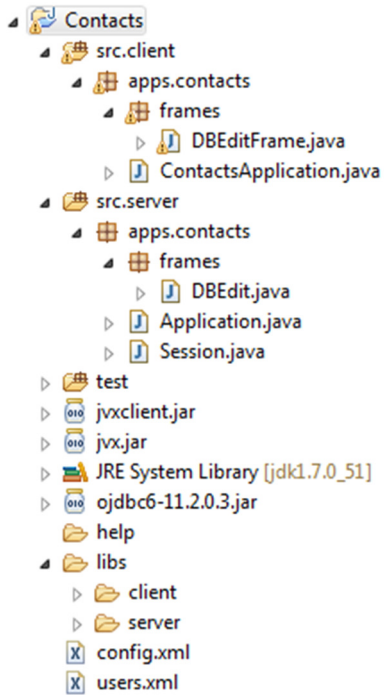
- Create database, tables
- Insert test data
- Create and Configure Contacts project
- Create Screen

The developer didn't create a demo application because the documentation of JVx' FirstApp had a screen with one table and this was the base for the Contacts project. All missing features were available in the documentation and so the first run was used for database creation and reading documentation.

The second run was super fast because the developer didn't have to read much documentation and database creation was trivial. The database creation wasn't faster than in the second run of the other frameworks.

If the database would have been ready, the developer would have been ready in 1 hour and 30 minutes.

Implementation overview



A bare minimum. The hierarchy is very flat. The project contains 5 source files and 2 xml files. No specific model classes.

Conclusion from the developer

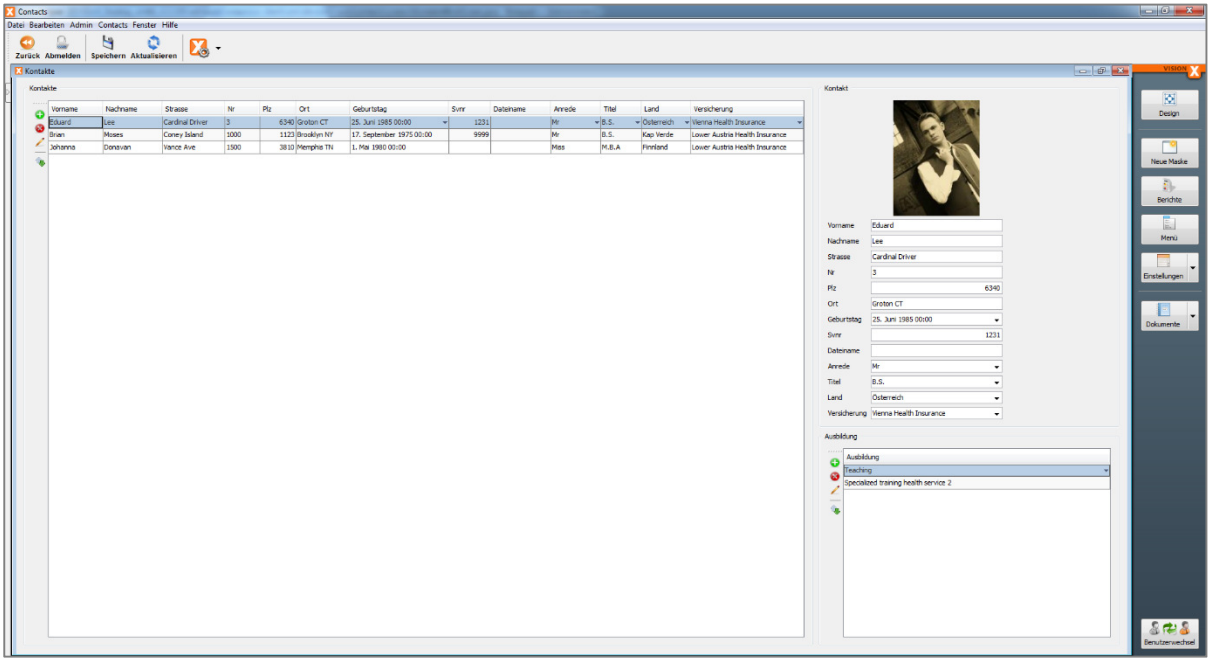
JVx handles GUI and database without additional dependencies. Two jar files are enough to start the application. The documentation for this use-case was great but I didn't find anything on stackoverflow. It was not trivial to work with the Forum as documentation system, but after some hours it was useful. There are many demo applicatons but the FirstApp was enough for the Contacts screen. The framework has an application frame but I didn't use it, only standard. There's an authentication system but I did use a simple xml file with user credentials – very simple. I was so fast, really cool!

Pros: Flat learning curve,
Same programming model for GUI and Server-side,
No specific database code,
Few files (it's minimalistic),
Different UI technologies (but I didn't try it, only watched youtube videos)

Cons: Documentation system is strange,
Swing UI doesn't look cool

VisionX (out of competition)

VisionX is a SIB Visions tool and it creates JVx applications. We thought it is a good comparison and to see what's possible. It's not Open Source and not free-to-use!



Used Tools: VisionX Desktop 2.2
Used Libraries: all included

Required time
First run: 1 hour 15 minutes
Second run: -

Tasks

- Create database, tables
- Insert test data
- Create application
- Create screen

One test run was enough because most time was spent for database creation and inserting test data. The application was ready after 15 minutes, with all required features. It could be faster if the database had been created automatically via VisionX, but we tried to compare VisionX with all other technologies and database creation always was a manual task in all other tests.

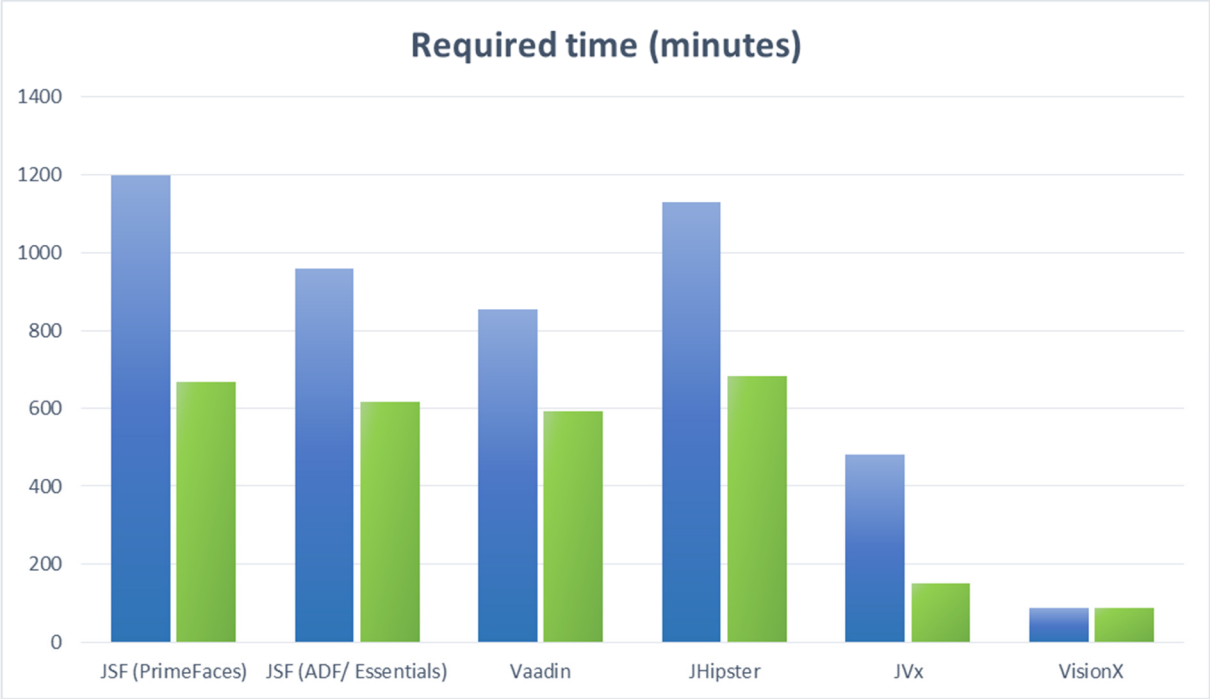
It won't be possible to create the screen faster than in 15 minutes. Another advantage was that the Vaadin application was ready in the same time, so we got two UI technologies in only 15 minutes.

Conclusion from the developer

This tool is rocket science. I've never seen such a tool before and it was super easy to use.

Statistics

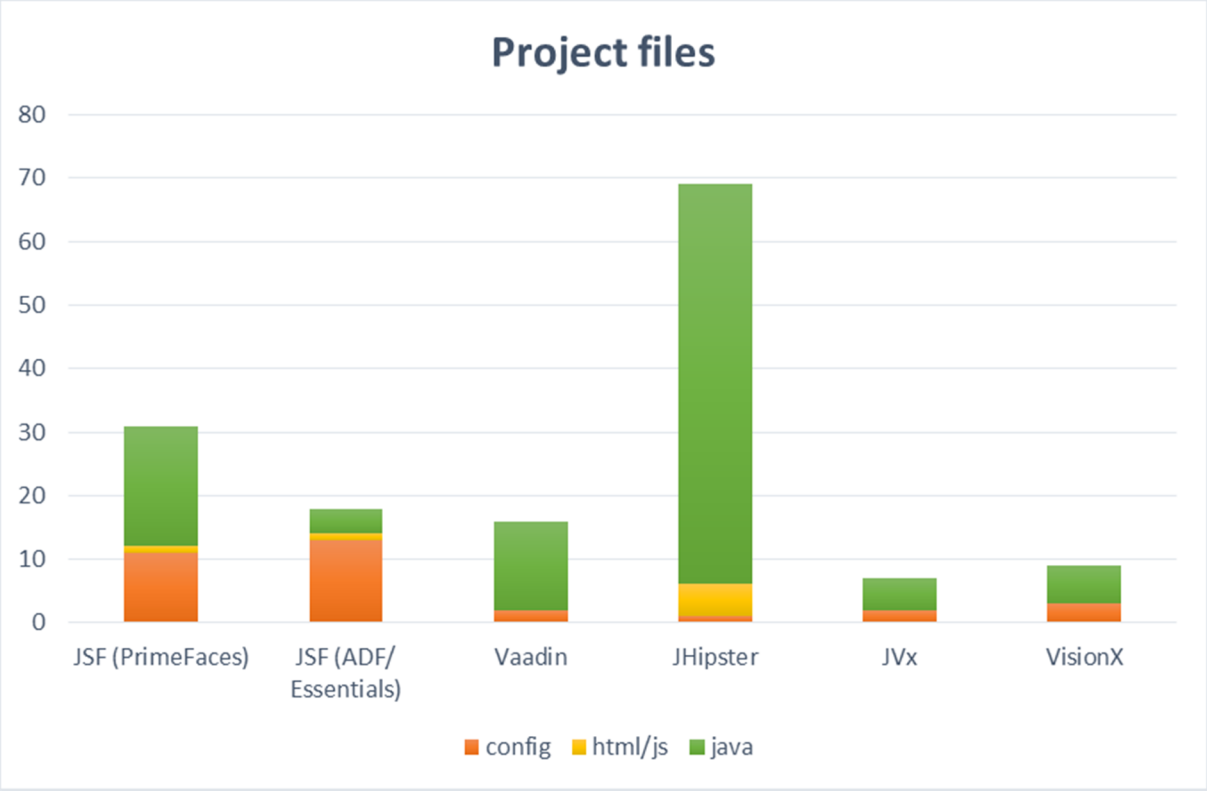
The following chart represents the direct comparison of „Required time“. The effort was measured in minutes.



The effort for the first runs were similar for JSF, Vaadin and JHipster. The technology didn’t make a big difference for a beginner. Also the second run was in the same range. The big difference was JVx in both runs. It was easy to start and after the developer had the know how, the time saving of 69% was better as with all other technologies. Only JSF (PrimeFaces) with a time saving of 45% had a comparable learning curve.

The next chart contains relevant Project files. Whe have three categories:

- Configuration (xml, properties, mappings)
- Html and Javascript
- Java Source Code



Especially JHipster had an extreme high number of project files. It was because JHipster is an application generator and the framework creates a full application frame and not only one screen.

ADF only had 4 Java source files, but 13 configuration files and 1 html page.

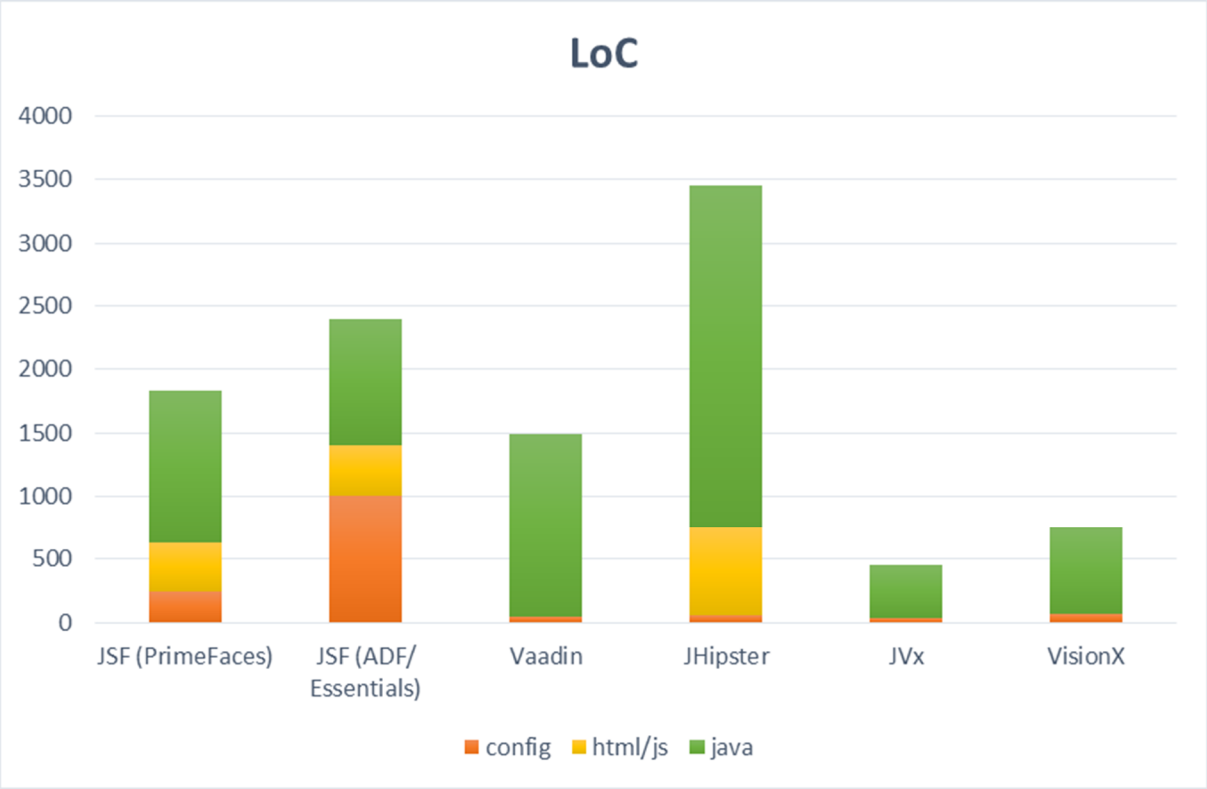
Vaadin had 14 Java source files (1 File was the GUI, all others were needed for JPA) and only 2 configuration files.

JVx had 5 Java source files and 2 configuration files. It didn't use a html page.

JSF (PrimeFaces) had 19 Java source files, 11 configuration files and one html page. This wasn't more than JHipster, but more than any other framework in our project.

It's easier to learn a framework and to maintain a project with a handful of files than a project with hundred files.

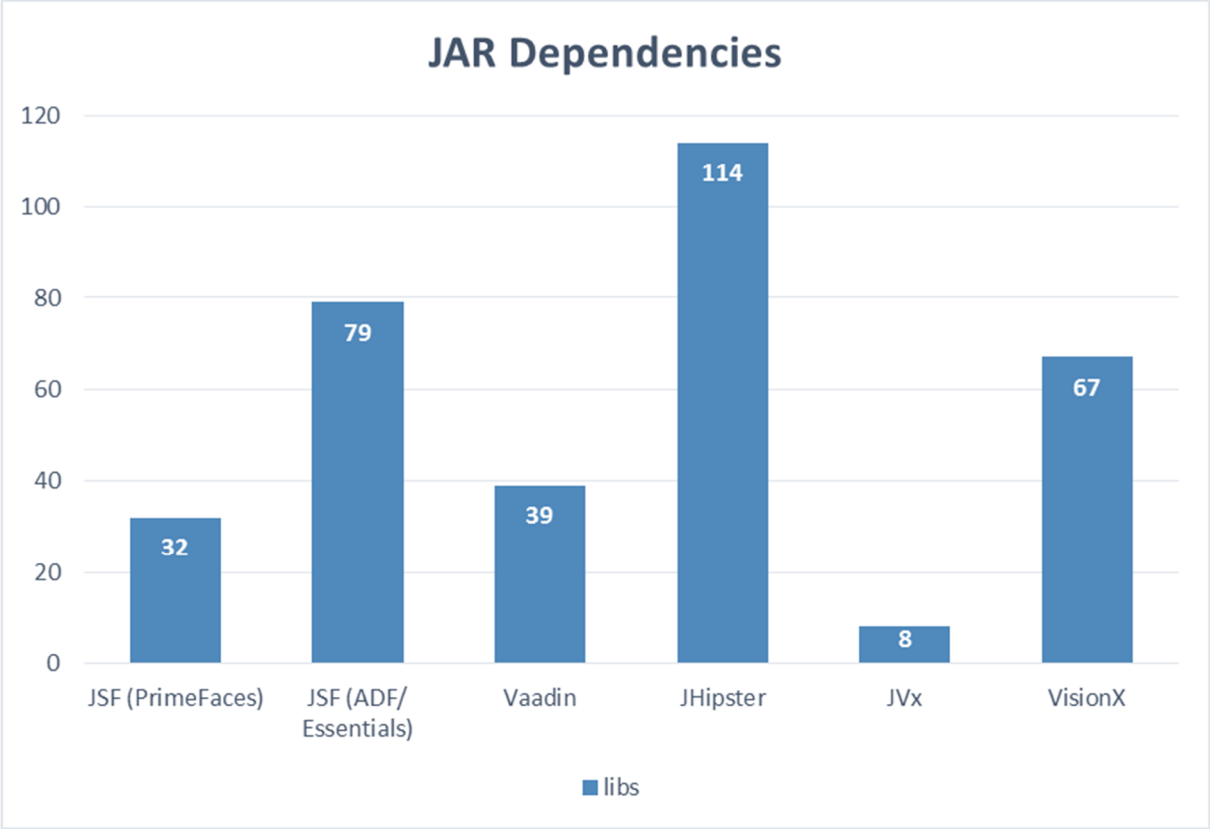
Next chart is an overview of used „Lines of Code“ which were needed to implement all defined features.



It’s clear that JHipster leads the group because the number of files was the highest of all projects.
Both JSF solutions look similar but ADF has a lot of configuration code and also a lot of Java source code with only 4 Java source files. PrimeFaces has 19 Java source files with similar LoC.
Vaadin looks nice and the configuration is small and simple. The rest is Java source code.
But again, JVx has a slim configuration and only 423 Lines of Java code.

It’s easier to understand projects with fewer LoC. This isn’t a big secret.

And finally we have a chart with project dependencies. It's good to know how many jar files a project needs and how many different APIs or frameworks are part of a project.



In general, fewer jar dependencies are better but one dependency can be as complex than five other dependencies. The number of jar files isn't always a good indiciator for complexity but it's also not wrong.

JVx has 8 jar dependencies and JSF with PrimeFaces has 32. JHipster leads the group with 114.

Conclusion

Our framework comparison confirmed that all used frameworks can handle complex tasks and that it's not hard to start with a new framework. The learning curve is steep enough and the documentation is most important thing if you're starting out with a (new) framework.

We discovered one big difference between all used frameworks: Required time.

It's very important for a developer to implement requirements as fast as possible and with the highest quality possible. Only JVx had a real advantage in this area.

The average development time of all other frameworks, in their 2nd run, was 10 hours and 38 minutes (638 minutes). The 2nd run of JVx was 2 hours and 30 minutes (150 minutes) which is around 76% faster than the average of all other frameworks.
Such a time saving is awesome and helps companies to earn more money in less time, because projects can be delivered faster and it's possible to do more projects or features in the same time.

One final information is very important. All developed solutions need additional time before they can be used in a productive environment because they aren't user friendly, they weren't styled and not well tested. It wasn't important for us in this project to create a good test coverage, but it's very important for a production ready system – for sure. Styling and usability wasn't important for our comparison because it's project specific and subjectively. Our focus was on functionality and development time.