jHipster Lite

Generate an application with only a few words

I have used jHipster to generate the “Admin” for my applications. So we get a user administration, CRUD user interface of all tables in the system etc.

Another use case was that I used jHipster to generate a full Repository to an application that mirrors data from SAP into a local sales application.
The UI part —> I never used. I copied the DTO’s, JPA entity classes and Liquibase definitions to my target Spring Boot Application. My client never really understood all this.

In this document I have tried to:

- express the power of jLite - ***Strong***
- give some concerns I have - ***Concern***
- and some suggestions for the future – ***Suggestion***

Conclusion

My conclusion is that jLite (as ***jhipster lite*** is called here) is a superiour tool. It is defining use of Hexagonal architecture. jLite defines where to place files in Hexagonal architecture like Maven did earlier for sources and test files. jLite remains a constant assistant to the developers.

jHipster is heavy. I understand why you made jLite.

Therefore I believe that jLite must take over features that classic can do like CRUD applications. And take over in a new way where the developer remains a dialog with ***jLite assistant***.

Custom fitting
jHipster classic and current jLite does not handle custom fitting of sources that has been generated by the generators. This paper gives a suggestion on how to obtain that.
See: *Custom fitting + Three way merge*.

JDL in jLite
This paper suggest a way to use JDL with an anticorruption layer that decouples classic from jLite.

Ensure jLite can grow
This paper suggests an internal generator to generate skeleton for a generators in jLite.

Developers consistent options
This paper suggests to seperate selection of technology from feature selecting

Using many small classes with single responsibility is in high demand. *Generating them is even better*.

/Anders K. Andersen, October 2023.

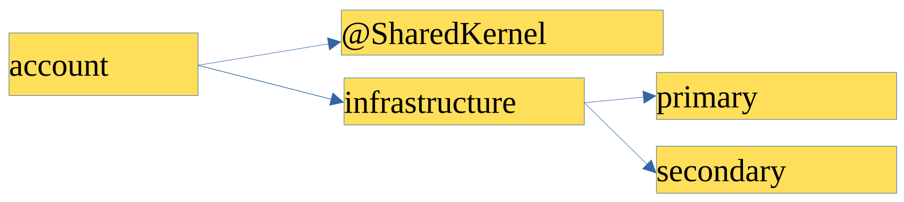
Table of Contents

Strong jLite: Hexagonal architecture

It is super that jLite declares the hexagonal architecture. I have used hexagonal architecture in projects with good results.

Maven gave us src/main/java + src/test/java (Thanks van zyl + team ❤️).

jLite gives us the structure for a Hexagonal application (Account is a sample)



This means that our industry has progressed.

Thanks to jLite ❤️

Strong jLite: Project assistant tool

Running jLite port 7471 together with a project is a super cool feature.

jLite makes it possible to maintain and extend a hexagonal architecture based application.

It is really useful.

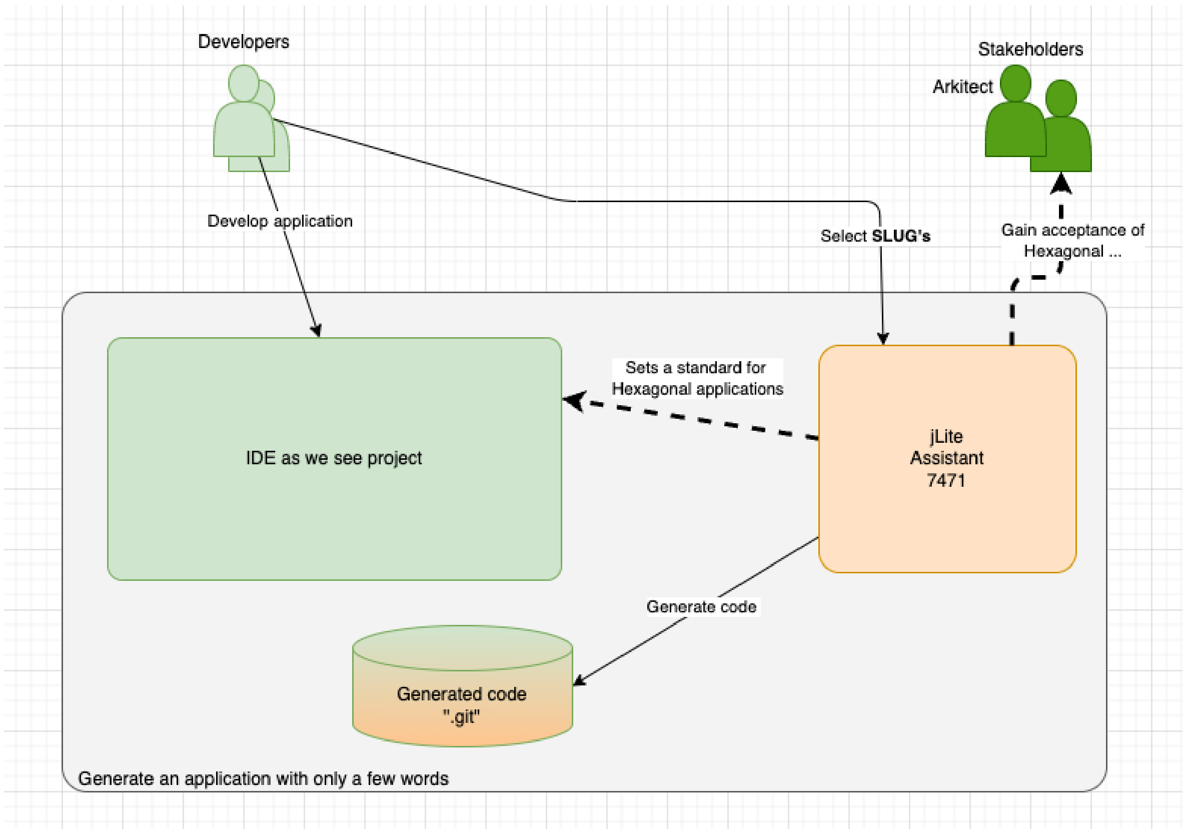


Figure: Developer experience with jLite

Quality that answers worries

- What is Hexagonal architecture? -- jLite explains very precise. ***Even document it***.
- Hexagonal is good, but let’s stay layered we know that! -- jLite generates the argument away.
- Hexagonal architecture requires more interfaces (the ports) It is overkill! -- jLite generates that argumet away as well. ... Or at least *must do* that in future 🐥

I feel that you made ***jLite Assistant*** was a milestone for our industri (nothing less 👍)

Concern: Being in control? 📣

When users are facing a complex product like ***jHipster*** and ***jLite,*** then people tends **not** to understand what is going on.

Therefore it is very important that jLite describes exact what it is doing. And when the user changes parameters jLite must specify what is changed in a safe way.

Users must understand:

- What generators are being used?
- What parameters are used in each generator?
- What output each generator produces?
- What change a parameter change produces?

jLite must answer questions like (se bullits right below) ***without doubt!!***

- I have added SLUG ***Neo4j***. Now I want to remove it?
  - In general all SLUG should be reversable.
  - Excluding ***init*** I assume. ***Init*** is not reversable.
  - That means a test can be: *Given* a SLUG was added. *When* removed again. *Then* it must be gone.
- I'm upgrading from jLite version V1 to V2
  - This means that Spring upgrades from X1 to X2
    - We must make sure that X1 Spring artifacts and generated code is replaced with what ever X2 has come up with
  - That also means that SLUG ***salando-problems*** has been deleted.
    - We must make sure that everything from that one will be removed
- I'm upgrading from jLite version V2 to V3
  - It turns out that V3 has a different view on SLUG ***consul***.
  - SLUG ***consul*** is no longer there
    - Customers must expect jLite to signal to the developer that ***consul*** is unknown and the developer must read the migration instructions.
      - In jLite ***consult*** might still be in SLUG list and marked as obsolete?

If we don't answer situations like above, then we just get a lot of fustrated developers that ends up giving jLite lots of troubles.

Objective: *Being in control*

Strong: Each module generation in own Git commit

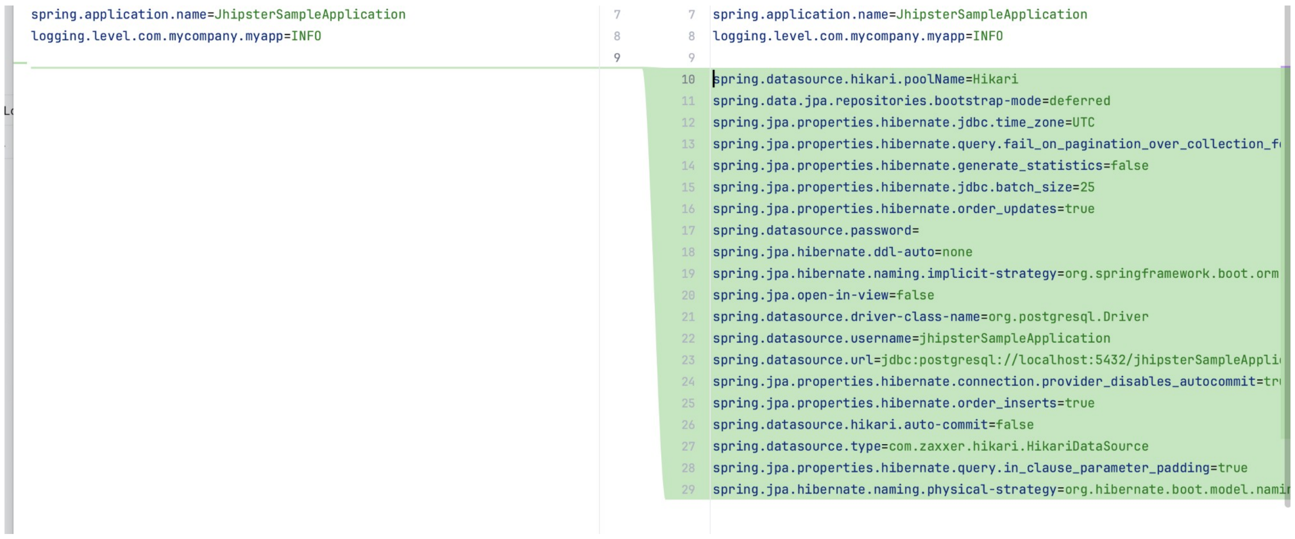
jLite store each module generation update in own Git commit.

That means that each code generation is separated and easy to identify.

This is very strong.

Concern: jLite removes custom fitting

The experience came when I added SLUG ***postgresql***



```
spring.application.name=JhipsterSampleApplication
logging.level.com.mycompany.myapp=INFO
```

```
spring.application.name=JhipsterSampleApplication
logging.level.com.mycompany.myapp=INFO

spring.datasource.hikari.poolName=Hikari
spring.data.jpa.repositories.bootstrap-mode=deferred
spring.jpa.properties.hibernate.jdbc.time_zone=UTC
spring.jpa.properties.hibernate.query.fail_on_pagination_over_collection_f
spring.jpa.properties.hibernate.generate_statistics=false
spring.jpa.properties.hibernate.jdbc.batch_size=25
spring.jpa.properties.hibernate.order_updates=true
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=none
spring.jpa.hibernate.naming.implicit-strategy=org.springframework.boot.orm
spring.jpa.open-in-view=false
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=jhipsterSampleApplication
spring.datasource.url=jdbc:postgresql://localhost:5432/jhipsterSampleAppli
spring.jpa.properties.hibernate.connection.provider_disables_autocommit=tr
spring.jpa.properties.hibernate.order_inserts=true
spring.datasource.hikari.auto-commit=false
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
spring.jpa.properties.hibernate.query.in_clause_parameter_padding=true
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.nami
```

Figure: The SLUG adds environment parameters ***application.properties***.

The generator sets ***spring.datasource.password*** to nothing. That means that the ***username*** becomes password as well. This is perfectly okay (***strong***).

Later I decide to set the password as a ***custom fitting***.

Later I applied the SLUG postgresql again.
Result ***spring.datasource.password*** was set back to nothing.
I conclude that jLite removes my ***custom fitting***.
I find that that behaviour is not 100 ***being in control*** 🦖.

This means that it is unclear in jLite what generated files has been custom fitted. Or at least it takes a much deaper understanding to maintain custom fittings.

My advice is to add ***custom fitting*** as a primary concern in jLite.

Concern: What generated files has been custom fitted?

Right now I might be swimming in deep water? I ***assume*** that an ***advanced query in git*** could conclude that changes to application.properties was generated by jLite and what changes the developer has intriduced in the file? And hereby deduct ***custom fittings?***
My concern is related to ***being in control***.

Suggestion: Separation of responsibility

jLite's *generator "department*" gets responsibility of maintain generated code in ***genlog***.

jLite's *patch "department*" together offers superiour tooling for patching project/customer code base. The developers have responsibility of ***patching*** the application and maintain customer fitting.

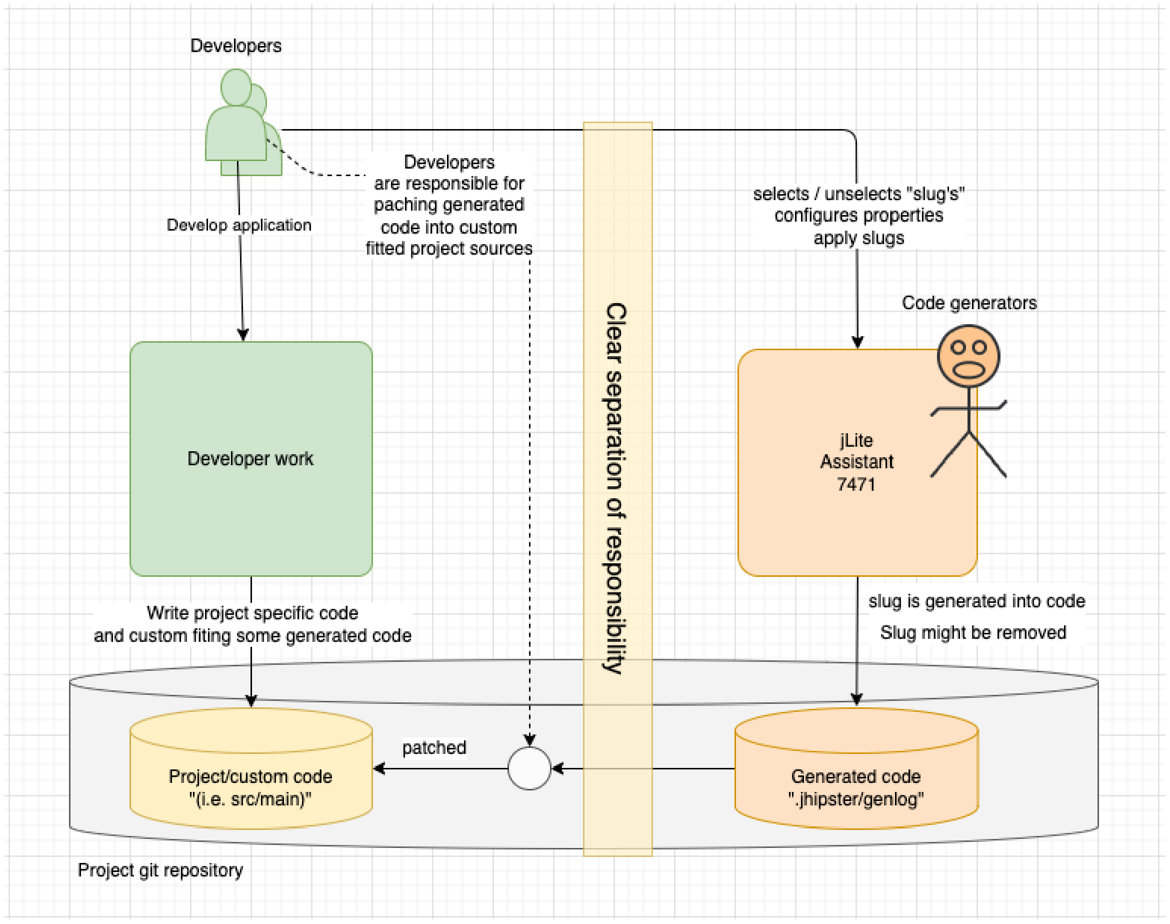

Developers
Develop application
Developers are responsible for paching generated code into custom fitted project sources
Clear separation of responsibility
selects / unselects "slug's" configures properties apply slugs
Code generators
Developer work
jLite Assistant 7471
Write project specific code and custom fiting some generated code
slug is generated into code
Slug might be removed
patched
Project/custom code "(i.e. src/main)"
Generated code ".jhipster/genlog"
Project git repository

Figure: Separation of responsitility. ***Genlog*** is suggested later in this document.

Advantage:

- Who do what is **simple** kept in separated files. Understanding files is easy to learn.
- jLite generator taking resonsibility for generating can be understood and trusted by developers.
- jLite patching of customers source code must be superviced by developers and ***owned by developers***. jLite offers superiour tooling for patching.
- Responsibilities matches physical reality.
- It is not a .git feature that maintains ***custom fitting***. It is explicit in files.
- Copy/cloning git and losing history information does not matter here.

Suggestion: genlog

Generator will send all output into **<<project-root>>.jhipster/*genlog*** directory structure that mirrors all generated sources.

Genlog details

1. All generated files like **src/main/java/com/mycompany/myapp/**JhipsterSampleApplicationApp.java will also be updated in **.jhipster/genlog/src/main/java/com/mycompany/myapp/**
2. genlog contains a 1:1 directory structure and file content of all generated files.

This enables jLite to 100% document what files are generarted. And it also enables the user 100% to understand what files are generated. They can see something similary in git, the mirror enables the project to custom-fit.

The mirror will not contain irrelevant directory clone like ***.git*** etc.

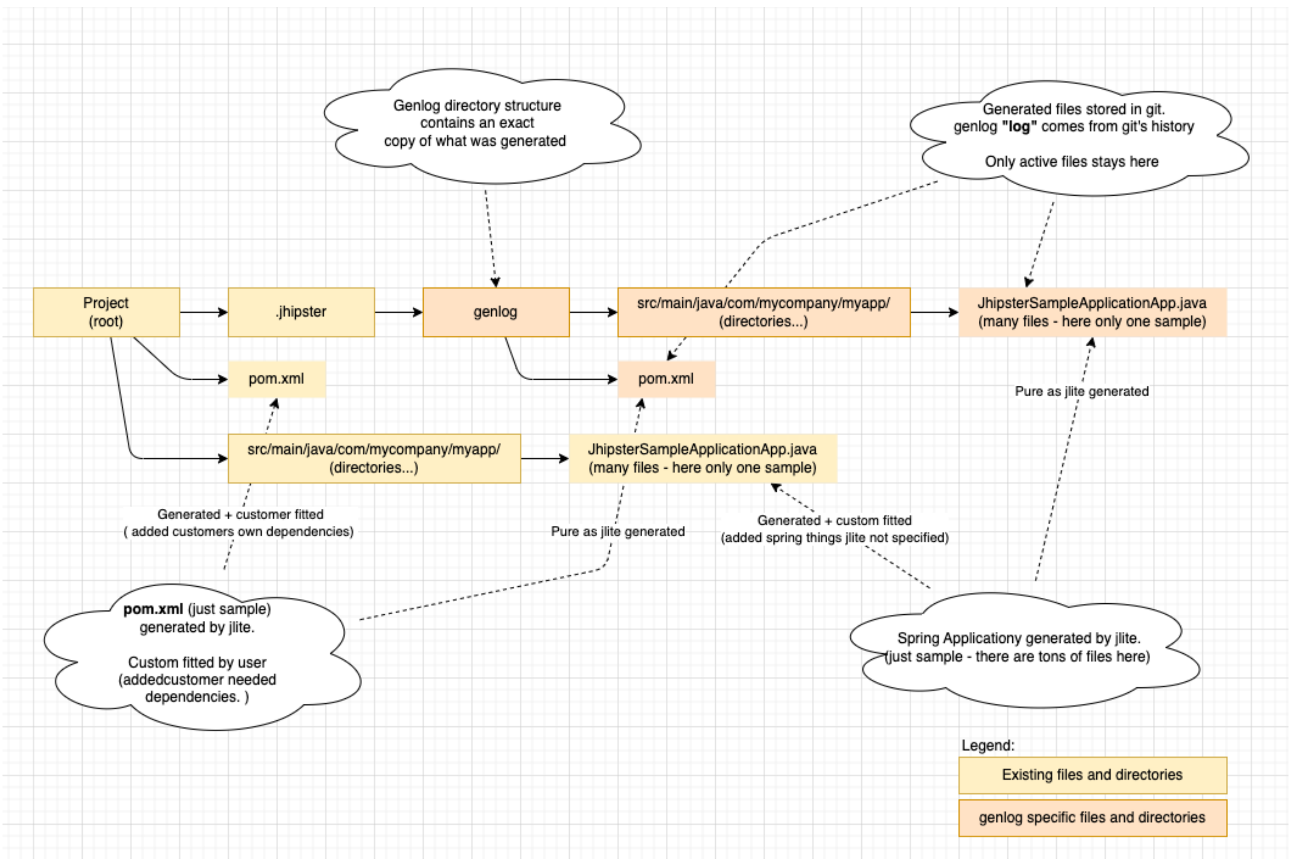

Genlog directory structure contains an exact copy of what was generated
Generated files stored in git. genlog "log" comes from git's history
Only active files stays here
Project (root)
.jhipster
genlog
src/main/java/com/mycompany/myapp/ (directories...)
JhipsterSampleApplicationApp.java (many files - here only one sample)
pom.xml
pom.xml
Pure as jlite generated
src/main/java/com/mycompany/myapp/ (directories...)
JhipsterSampleApplicationApp.java (many files - here only one sample)
Generated + customer fitted (added customers own dependencies)
Pure as jlite generated
Generated + custom fitted (added spring things jlite not specified)
pom.xml (just sample) generated by jlite.
Custom fitted by user (addedcustomer needed dependencies.)
Spring Applicationy generated by jlite. (just sample - there are tons of files here)
Legend:
Existing files and directories
genlog specific files and directories

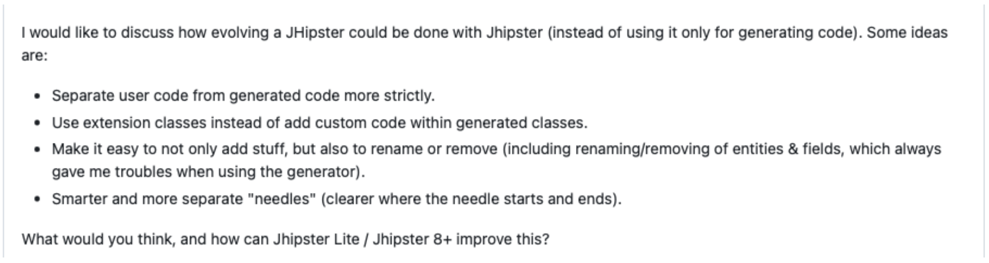
Figure: Genlog mirror directories and files.

Advantage:

- Understanding files is simple to learn.
- Separation responsibility is separated into two different directory trees.
- It is easy to explain. How do I see custom fitting? Answer: “*Just diff **project** vs **genlog**!*”

Strong: jLite team considers custom fitting

jLite team members are concerned about endusers


I would like to discuss how evolving a JHipster could be done with Jhipster (instead of using it only for generating code). Some ideas are:

- Separate user code from generated code more strictly.
- Use extension classes instead of add custom code within generated classes.
- Make it easy to not only add stuff, but also to rename or remove (including renaming/removing of entities & fields, which always gave me troubles when using the generator).
- Smarter and more separate "needles" (clearer where the needle starts and ends).

What would you think, and how can Jhipster Lite / Jhipster 8+ improve this?

Figure: Note from https://github.com/jhipster/jhipster-lite/discussions/512 about custom fitting.

Strong: The jLite team signals support for generated code must be custom fitted.

Concern: Custom fitting not primary in jLite

A good design principle is to separate generated sources from custom sources.

The problem is that several sources will need a mix of custom fitting and generarted code.
This is in example:

application.properties
We know will be a mix of jLite generated settings and customers own setup.

pom.xml
Customer must be able to add dependencies. Maven is designed that way.

JhisterSampleApplicationApp.java
Spring main class will often be custom fitted.

And many other samples.

Problem when jLite not answers this custom fitting

My example on Postgresql database password is a problem of this type.

Suggested solution for custom fitting

By adding ***genlog*** where all generated sources are stord 1:1 we know exactly what has been generated.

Compare the ***genlog/src/***... tree to ***<<project>>/src/***... we can see what code has been custom fitted. And we can see what the custom fit is all about. IntelliJ can make that compare out of the box. Severeal other tools can do the same thing. Ie. WinMerge.

See: ***Feature: Custom fitting + three-way merge***

Suggestion: Custom fitting + three way merge

The end user project must be able to custom fit the application after it is generated.

This requirement is a fundamental foundation for ***staying in control***.

Custom fitting limitations

1. There can be files that jLite cannot tolerate that customer add customer fitting into. jLite will document that in the generated files.
    1. jLite will be able to verify user has violated the constraint (see ***genlog***)

Generator update can use three-way merge

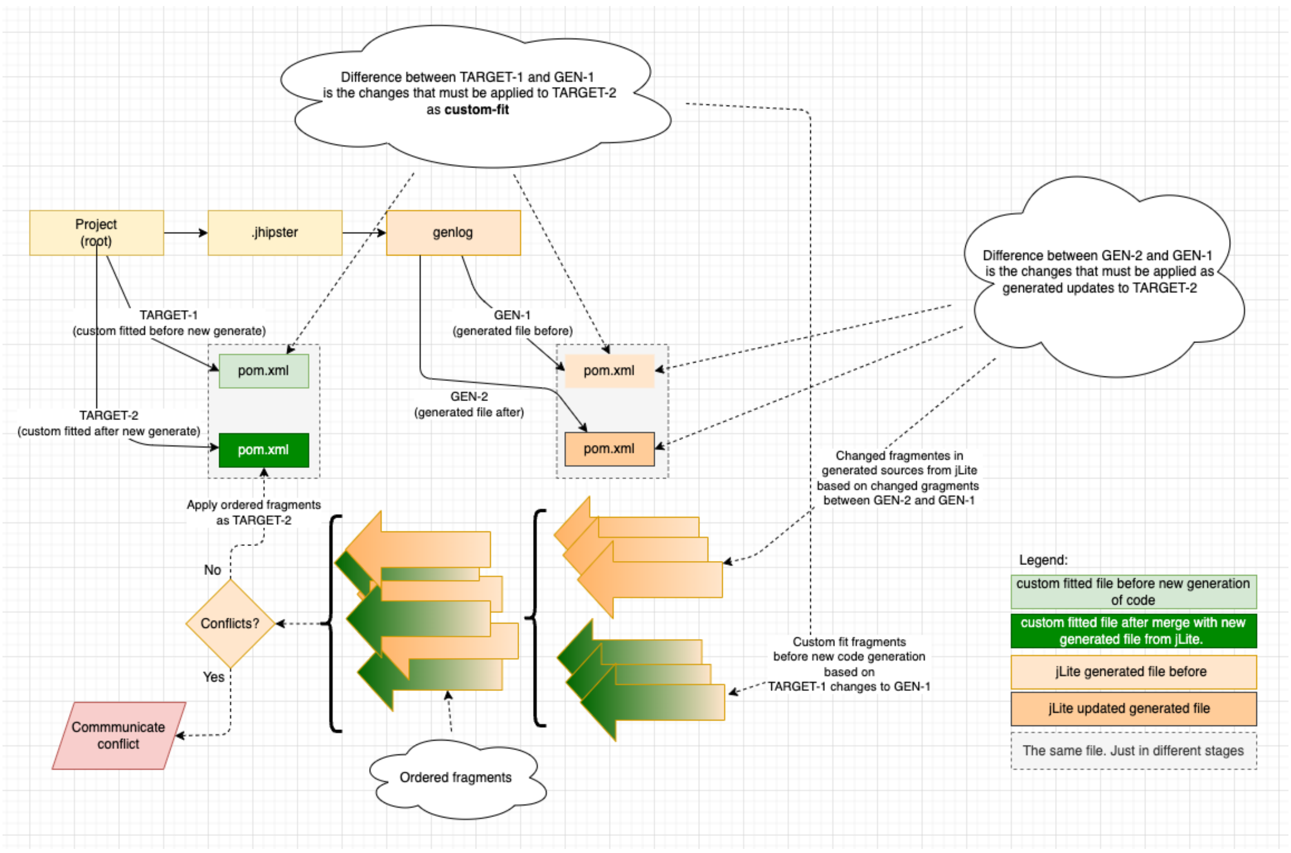

Difference between TARGET-1 and GEN-1 is the changes that must be applied to TARGET-2 as custom-fit
Project (root)
.jhipster
genlog
Difference between GEN-2 and GEN-1 is the changes that must be applied as generated updates to TARGET-2
TARGET-1 (custom fitted before new generate)
GEN-1 (generated file before)
pom.xml
pom.xml
TARGET-2 (custom fitted after new generate)
GEN-2 (generated file after)
pom.xml
pom.xml
Changed fragmentes in generated sources from jLite based on changed gragments between GEN-2 and GEN-1
Apply ordered fragments as TARGET-2
No
Conflicts?
Yes
Commmunicate conflict
Ordered fragments
Custom fit fragments before new code generation based on TARGET-1 changes to GEN-1
Legend:
custom fitted file before new generation of code
custom fitted file after merge with new generated file from jLite.
jLite generated file before
jLite updated generated file
The same file. Just in different stages

Figure. Three-way merge

https://en.wikipedia.org/wiki/Merge_(version_control)#Three-way_merge

Strong: jLite development process

jLite team has started jLite because jHipster “classic” got problems. Especially developers do not understand ***classics*** process, upgrades are difficult , developing classic internal is difficult and Jeoman is not maintained any longer.

Strong: jLite can add modules 1 by one and the developer can follow whats added for each step.

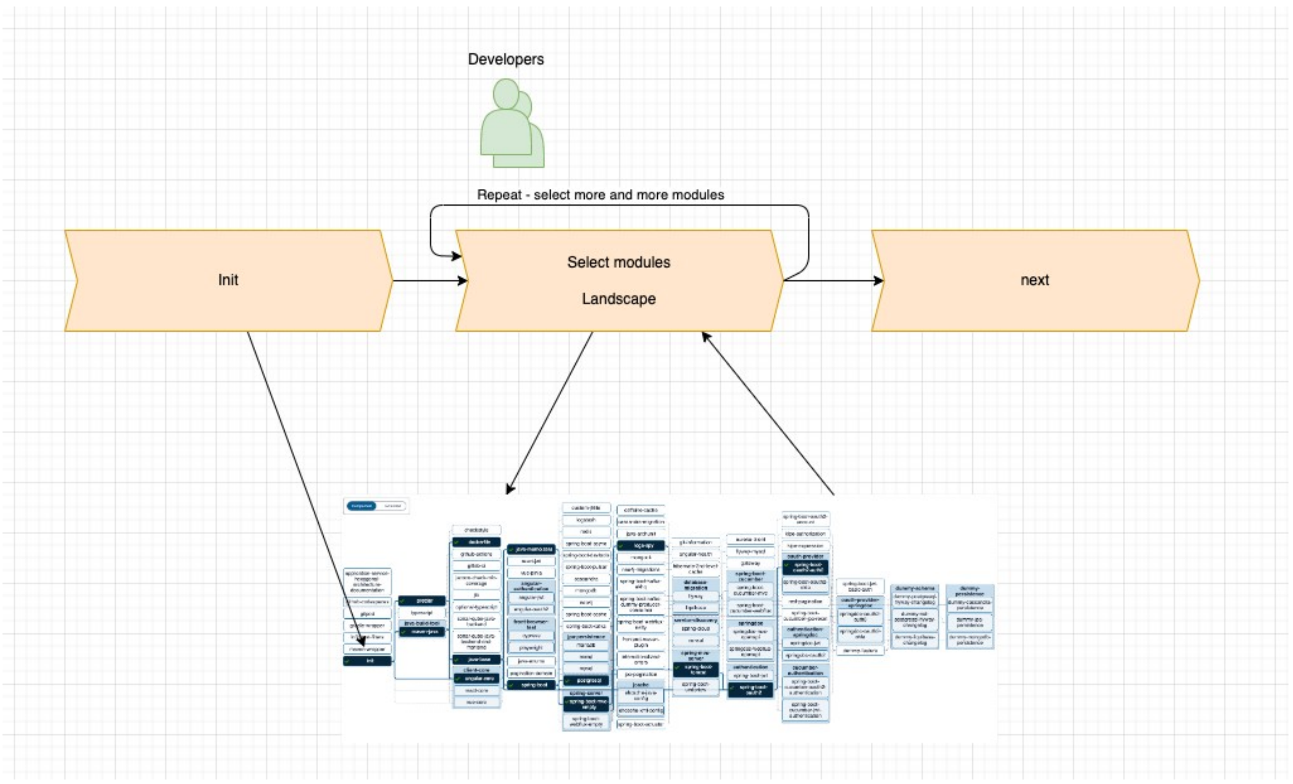
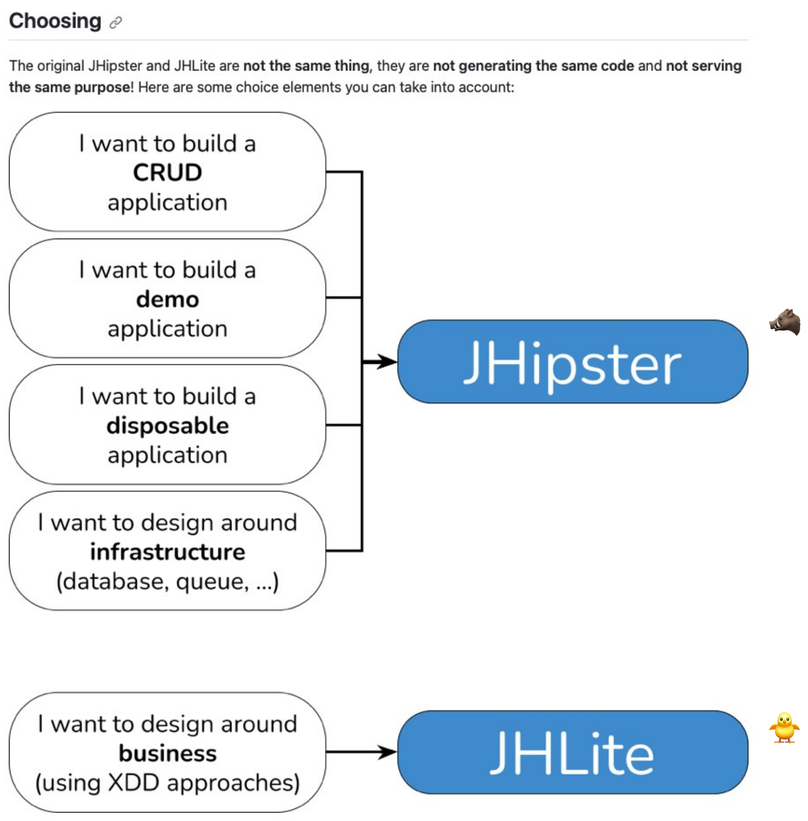


Figure: jLite developement process consists of an init + iterative

Strong: It is beautiful that developer can add modules in a speed that the developer can absorb.

Concern: jLite use-cases can make jHipster irrelevant



Choosing

The original JHipster and JHLite are **not the same thing**, they are **not generating the same code** and **not serving the same purpose**! Here are some choice elements you can take into account:

I want to build a **CRUD** application
I want to build a **demo** application
I want to build a **disposable** application
I want to design around **infrastructure** (database, queue, ...)
JHipster
🐗
I want to design around **business** (using XDD approaches)
JHLite
🐥

Figure: jLite's Github readme main page.

The figure shows guidance on when to use jHipster and when to use jLite.

According to https://docs.google.com/presentation/d/1i0LOJ0GSWNG2-x0zY220IbQc0PVQ2pndQWEuQKGu8n0/edit#slide=id.g5b8c73bad6_2_90 ***JHipster Lite Presentation*** I get the concern that JHipster classic will no longer be maintained.

- This means that jHipster (classic) will stop existing as a super and very powerfull application builder. Not nice reading 🐗.
- jLite is a XXD business oriented tool. That is great reading 🐥

Concern: *I'm afraid that* ***jHpister****, as a whole,* ***can become irrelevant*** *if jLite stays on that ambition*

My understanding is that ***jLite*** will be the main developed tool. And for a good reason. ***jLite*** has used the experiences from ***classic*** and is easier to work with and easier to maintain. And therefore jLite will to gain from answering all the use cases.

Suggestion: jLite development process

jLite team has started a great development process. This chapter contains improvement to this process. The main suggestion is to improve Landscape.

Steps:

1. Init – as today
2. Technology – tell what technologies to use (***new suggesion***)
   Select UiFramework: (Angular, react or vue). Database: mariadb, posgresql, etc. Dbmigration: flyway, liquibase. Reactive: webflux. Authentication: jwt, oauth2. WebServer: Tomcat, Undertow, Netty. Updatess file ***.hipster/technology.json***
3. Feature item - like Landscape today
4. Functinality – features from jHipster classic. view CRUD, Repository generator for an domain/entity, adapter generator (restController) for an domain, Kernel process for a domain, UiCrud for a domain, UiProcess for input, output domains. UI menu, UI microfrontend for a domain, DbMigration for a domain (liquibase scripts) ***(new suggestion)***

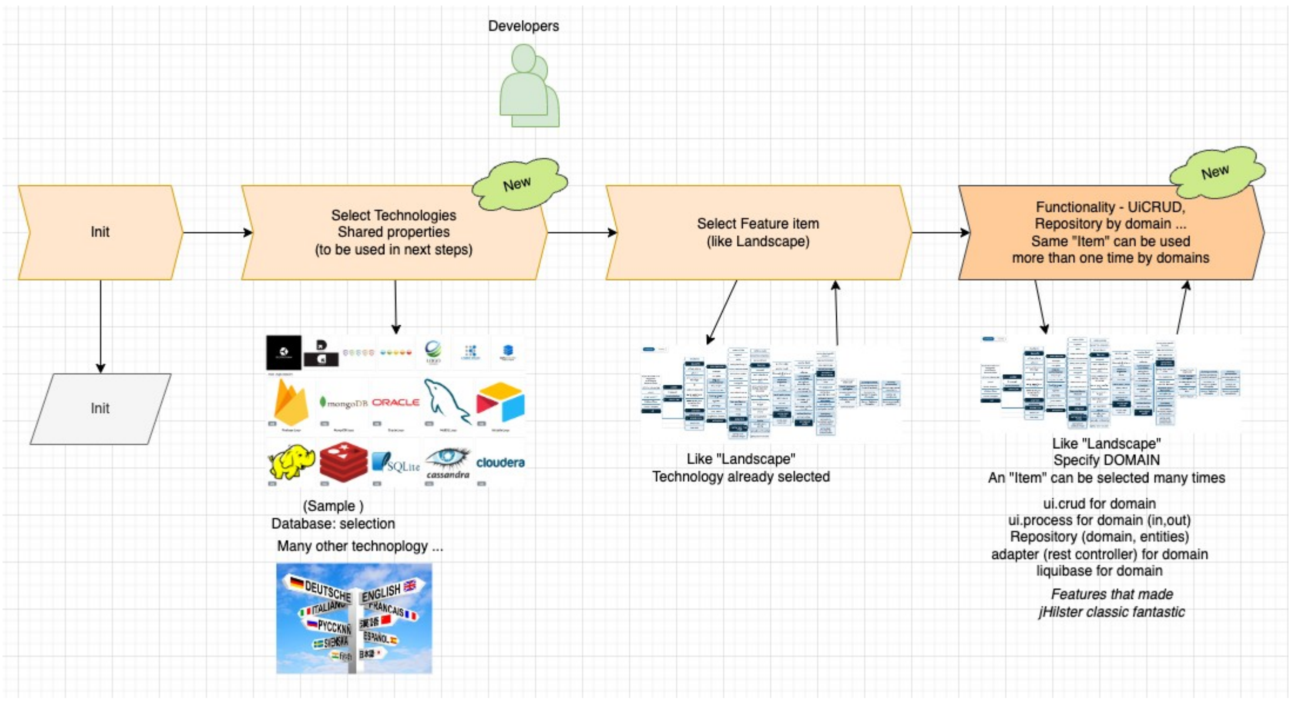


Figure: jLite developement process consists of different category of selections
(Disclamer: the figure is naive)

PS: This suggestion is still a big YES to Landscape. Landscape is an awsome overview.

Strong: jLite modules

jLite devides generation of an application based on several modules.
This is super strong and it makes it possible to maintain an application and upgrade it.

I have not really understood the difference between an ***organization, module, feature*** and a ***slug***?
So I hope you will bear with me.
And my use of "_", spaces and tile case is only samples.

Suggestion: jLite module devided

In this suggestion a ***module*** is based on an ***Item***.
And I see the ***module*** as the resulting set of generators that generates files for an ***Item***.

It is a question of building a "bom" ***Bill of Material***

Technology / shared property level

Item can be ***jpa persistence***. Or ***client core***.
This means that ***jpa persistence*** requires a database "***technology***" mariadb, mssql, mysql or postgresql. (In the code this is referenced as feature)

So the ***module*** will get an **id** of ***Item***+***Technology***. I.e. **jpaPersistence.postgresql**

Inside in jLite's codebase we will probably only have one ***Item*** java package named something like ***jpaPersistence***. Because all generated code is very simular for all database technologies.
And it is important to be able to add databases without rewrite too much.

I suggest that the user selects ***technologies*** in a section that is seperate from ***feature*** level (Landscape today).

Technology is Mutually exclusive So jpa persistence can not select more than one database.
Shared properties are free. User can select more than one language.

Feature level

Items can be more on a feature level. These are gradle-wrapper, maven-wrapper, gitpod, infinitest-filters, github-actions, docker-file, checkstyle etc.

Features can also require an option. I'm a little in doubt about that it is only properties?

Instance level

This is where jHipster Classic's super cool is split into operational ***Items*** that performs a single feature for a domain element, db entity, menu item or similary.

We look at a sample of generating DDD repository

Below sample is based on Colin Damin’s presentation of an invoice application. See: Litterature (2) ***Simple WebServices with JHipster Lite***

Here ***domain level*** will use an ***Item*** called “***ddd repository***” and that will generate two DDD Repositories one for “invoice-basis” and another repository for “invoice”.

This means we will get two modules: “***dddRpository.invoiceBasis***” and “***dddRpository.invoice***”

InvoiceBasis is a domain-fragment with an entity like definition like an entity in JDL.

Invoice is is a domain-fragment with an invoiceHeader entity and an invoiceDetail entity as we know from JDL two entities with a relation.

I suggest that domain definitions are seperate JSON documents or classses with a annotations that the Item “repository” gets as input and based on this input generates DDD Repository classes.

That means that JDL is NOT adviced to be migrated into jLite. Let JDL stay outside jLite.

I have a suggestion on making a generator item that takes a JDL and makes these domain definitions. See: ***jLite JDL domain generator***

Version level

We must be able to develop jLite in the future. We can make new editions of ***generator items***. We don’t have to be cheap in adding great features for the future.

It is important that jLite stays relevant by improving the generators.

The flip-side of changing existing generators can make the eco-system unstable.

Therefore it is a good idea to start our items with version “v10”

Module name samples

- jpaPersistence.postgresql.v10
- jpaPersistence.postgresql.v2x -- A version 2 we have in progress ...
- clientCore.angular.v10
- mavenWrapper.v10
- dddRpository.invoiceBasis.v10
- dddRpository.invoice.v10

Module is currently used in ***.jhipster/modules/history.json***

Definition: Instance vs singleton of module

Singleton/class

Current jLite modules can be used once.
A module like "***java-base***" generates some java classes. And this module can only be once in the ***Landscape***. And it does not make any sense to see ***java-base*** multiple times in *Landscape*.
Module **java-base** acts like a Spring singleton or like a class in Java.

Instance/entity/domain

When we want to generate for example a CRUD user interface then there will be a generation of code for each entity.
In the invoice example we have at least two domain fragments. First the input to the invoice called ***invoice-basis*** that contains the delivered orderlines for a customer. Second we have the ***invoce*** it self. The ***invoice*** got a header with the invoce number and possible amount total to pay and it got details for each orderline and possible taxation details.

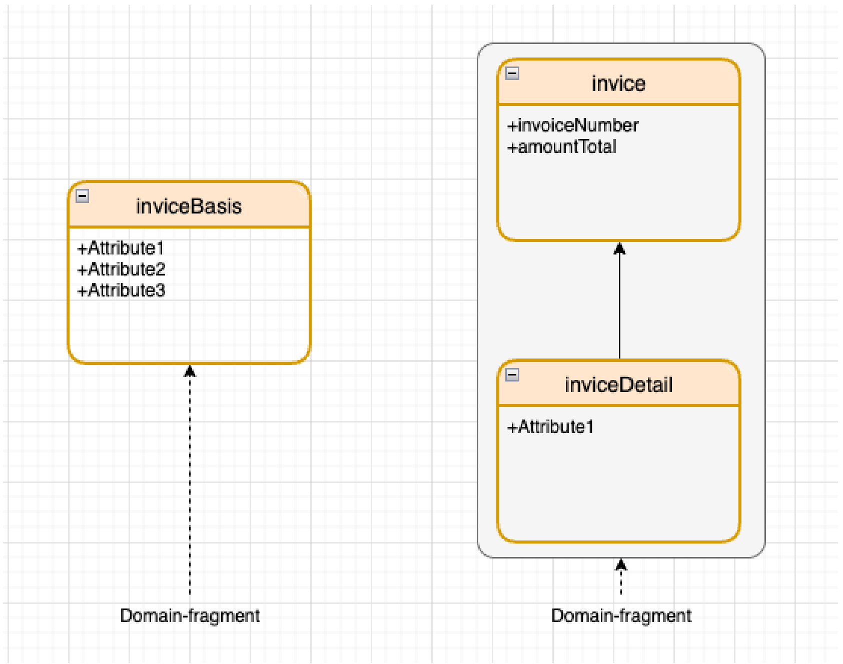


Figure: Sample "domain" of invoice and basis data to generate invoice. It is here to demonstrate the point of ***Instance vs singleton of module***.

Generate DDD repository / database-migration

jHipster classic would generate ***dddRpository*** a single module
or actually in "***the-big-single-module***".

jLite could easily generate a single ***dddRepository*** that contains repository for all domain-fragments in the system.

There are several options to discuss in jLite.

- Would we generate a dddRepository for invoiceBasis? Maybe invoiceBasis exists from a query in the ***order-delivery*** tables (not in model above) ? So we do ***not*** want a ***dddRepository.invoiceBasis***? And certanly we do NOT want a ***database-migration.liquibase*** entry for ***invoiceBasis,*** <u>because it is not a table</u>.

So we have actually 3 possible dddRepository entities.

- Module ***dddRepository.invoiceBasis***
- Module ***dddRepository.invoice***
- Module ***dddRepository.invoiceDetails***
  Maybe this entity is inside module ***dddRepository.invoice***?
- Maybe all three database entities should be in same repository?

The clue is that a generator Item like ***dddRepository*** can generate several modules with identity that can be shown in Landscape in one way or the other.

This is what ***Instance*** means here.

Generate CRUD for sample domain

jHipster classic would generate **ui.*CRUD*** as single module or actually in "***the-big-single-module***".

When we open up for instance level modules then jLite has a better and more relevant way to generate value for the customer.

Example – free from the liver:

- I would like a ui.CRUD for invoiceBasis (modules could be called ui.CRUD.invoiceBasis + openApi.invoiceBasis) - This would be a great start for an application
- I don't need a ui.CRUD for table ***invoice***.
- I would like a ***ui.CRUD_header_detail*** for ***invoiceDetails***.
  (module name: *ui.CRUD_header_detail.invoiceDetails)*

Strong: Using many small classes

Above instance approach will cause customer applications to consist of significant more classes. Each ***dddRepository.xyz*** will get an interface class for each and an implementation class + some more.

This is also a more fundamental consequence of jLite’s paradigm shift from jHipster classic. Because things are broken into modules.

My personal exiperience in 20 years of Java programming is that customer applications often suffers from classes that takes way to big responsibility 🐗

It is strong to generate all this. Generating code is done by the computer much faster than the developer can write the code. This means the ***cost*** of declaring “boilerplate” code i***s low***.

Written simple: ***Low cost***.💰

Concern: jLite modules costy to make

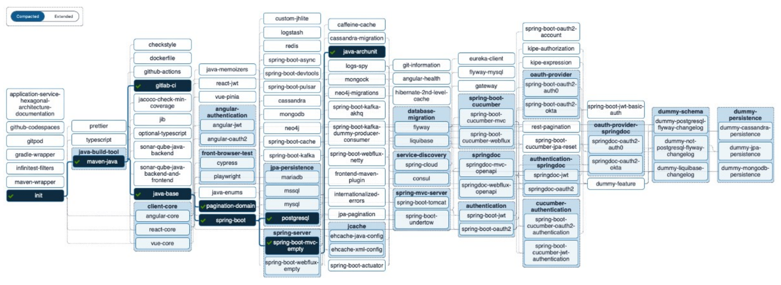


Figure: jLite got a little over 100 generator Items (modules)
Domain-fragment is a term that describes an instance that can be a pure database entity, a DDD-domain entity or similary without being specific.

Right now jLite is not covering what jHipster classic can do. In order to make enough generator Items to match standard usage of jHipster I assume we need 30 generators more and these generators are not of the simple ones.

These generators can be:

- ***database-migration*** per entity (SQL table)
- ***ui.Crud*** generation for a domain-fragment
- ***ui.Crud.filter.rows***
- ***ui.Crud.edit.row***
- ***ui.Crud.add.row***
- ***ui.menu*** generation for group of ui windows
- ***language*** - stuff
- ***openApiCrud*** for domain fragment
- ***dddRepository*** for domain-fragment
- ***demo-data*** for a domain-fragment
- ***kernelCrud*** for a domain-fragment (that minic a layered application as a hexagonal application), ***ui.process*** for input/output domain-fragment,
- ***openApiProcess*** for domain fragment
- ***kernelProcess*** for a domain-fragment.

I can emagine we might get appetite on ***Micro frontends***. This would probably make another 10 modules.

My concern is that jLite becomes a buttle neck. That will not be able to evolve fast enough.

How can we make jLite internal development easier and fast?
See: *Suggestion: Add jLite item generator*

Suggestion: Add jLite item generator

Internally in jLite it makes sense to make a item generator that adds generators inside jLite it self.

Note the terminology in this document:

- An ***item generator*** is the code that makes a module. An item exists in jLite and is identified with a "slug" (enum JHLiteModuleSlug) or
  a "feature" (enum JHLiteFeatureSlug).
  This means that an ***item*** is a common way to say ***slug*** or ***feature***.

- An ***instance*** is a domain name, domain fragment name, entity name, menu item name or window name or similary. Several modules can be based on same item generator just with different instance.
  Instance is optional. A module does not need an instance. Current jLite v0.45.0 does not use instances.

- A **module** is something that exists in the customer application. It is found in file ***.jhipster/modules/history.json***
  So modules does not really exist in jLite. A module is the final generated product produced by jLite based on user input of Item (slug, feature) and instance.

Item generator types

- Generate a new item ***NEW-module*** *(actually a **generator Item** in my eyes)*
- Generate a version item for an existing item. Maybe an IDE can do this?

Outcome of module generator

- Swagger: GET /api/modules/NEW-module
- Swagger: POST /api/modules/NEW-module/apply-patch
- Enum JHLiteModuleSlug added NEW-module
- Added JhipsterModuleResource for NEW-module
- Added JhipsterModule build<<NEW-module>> to service
- ... + more

Strong: jhipster.JDL

jHipster classic is a super cool tool that generates a CRUD application.
And it is very useful.

I will risk to state that ***JDL is the reason for jHipsters success***.

Concern: jhipster.JDL feature not in jLite

When jhipster.JDL is so strong then I get concerned when the ***ambition*** *of jLite is* ***not to provide similar feature***.

This must be seen together with the situation where jHipster classic's product age and position in product lifecycle is **not strong**. See litterature (1) Jhipster Lite Presentation.

It is not calm to see this conflicting situation.

I don't think jLite shoud make JDL direct part of internals in jLite. I think jLite should introduce an ***anticorruption layer*** between JDL and jLite.

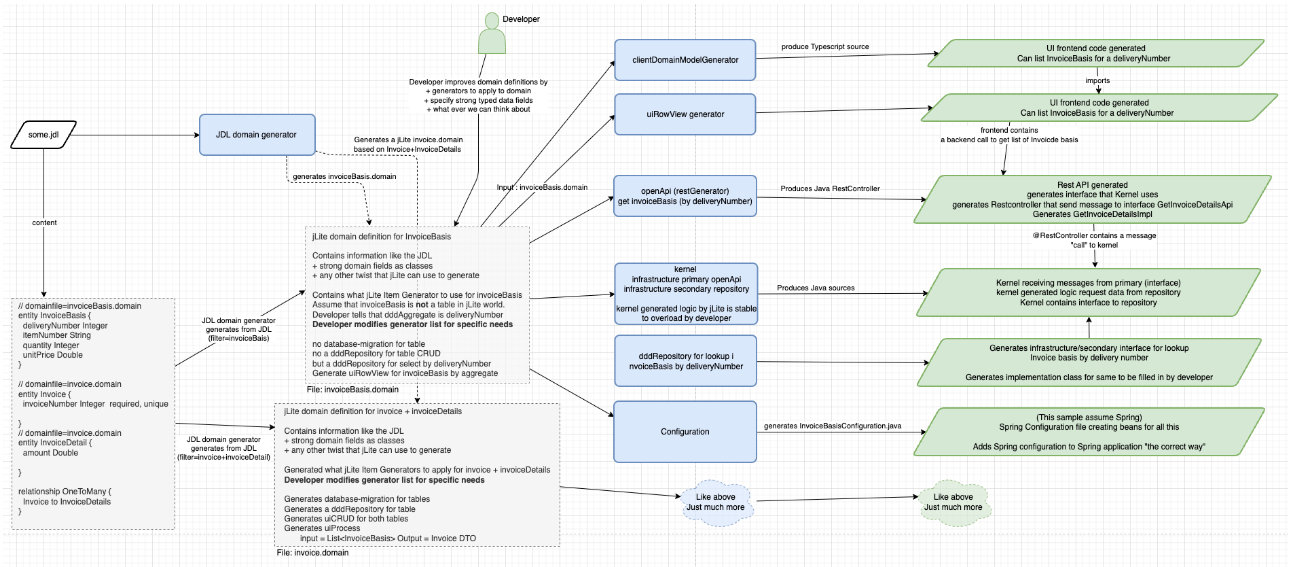
See suggestion in next paragraph.

Suggestion: jLite JDL domain generator

The suggestion is that jLite makes a domain generator that takes a file ***some.jdl*** as input and generates jLite domain definitions.

JDL domain generator does not directly generate **modules**. JDL domain generator generates domain definitions that ***other*** jLite generator Items will use to generate **modules**.

jLite domains definitions is also an **anticorruption layer** when we see it in relation to JDL.


Developer
Developer improves domain definitions by
+ generators to apply to domain
+ specify strong typed data fields
+ what ever we can think about
some.jdl
JDL domain generator
content
Generates a jLite invoice.domain based on Invoice+InvoiceDetails
generates invoiceBasis.domain
Input : invoiceBasis.domain
JDL domain generator generates from JDL (filter=invoiceBais)
JDL domain generator generates from JDL (filter=invoice+invoiceDetail)
jLite domain definition for InvoiceBasis
Contains information like the JDL
+ strong domain fields as classes
+ any other twist that jLite can use to generate
Contains what jLite Item Generator to use for invoiceBasis
Assume that invoiceBasis is not a table in jLite world.
Developer tells that dddAggregate is deliveryNumber
Developer modifies generator list for specific needs
no database-migration for table
no a dddRepository for table CRUD
but a dddRepository for select by deliveryNumber
Generate uiRowView for invoiceBasis by aggregate
File: invoiceBasis.domain
jLite domain definition for invoice + invoiceDetails
Contains information like the JDL
+ strong domain fields as classes
+ any other twist that jLite can use to generate
Generated what jLite Item Generators to apply for invoice + invoiceDetails
Developer modifies generator list for specific needs
Generates database-migration for tables
Generates a dddRepository for table
Generates uiCRUD for both tables
Generates uiProcess
input = List<InvoiceBasis> Output = Invoice DTO
File: invoice.domain
clientDomainModelGenerator
produce Typescript source
UI frontend code generated
Can list InvoiceBasis for a deliveryNumber
imports
uiRowView generator
UI frontend code generated
Can list InvoiceBasis for a deliveryNumber
frontend contains
a backend call to get list of Invoicde basis
openApi (restGenerator)
get invoiceBasis (by deliveryNumber)
Produces Java RestController
Rest API generated
generates interface that Kernel uses
generates Restcontroller that send message to interface GetInvoiceDetailsApi
Generates GetInvoiceDetailsImpl
@RestController contains a message "call" to kernel
kernel
infrastructure primary openApi
infrastructure secondary repository
kernel generated logic by jLite is stable to overload by developer
Produces Java sources
Kernel receiving messages from primary (interface)
kernel generated logic request data from repository
Kernel contains interface to repository
dddRepository for lookup i nvoiceBasis by deliveryNumber
Generates infrastructure/secondary interface for lookup
Invoice basis by delivery number
Generates implementation class for same to be filled in by developer
Configuration
generates InvoiceBasisConfiguration.java
(This sample assume Spring)
Spring Configuration file creating beans for all this
Adds Spring configuration to Spring application "the correct way"
Like above
Just much more
Like above
Just much more

```
// domainfile=invoiceBasis.domain
entity InvoiceBasis {
  deliveryNumber Integer
  itemNumber String
  quantity Integer
  unitPrice Double
}

// domainfile=invoice.domain
entity Invoice {
  invoiceNumber Integer  required, unique

}
// domainfile=invoice.domain
entity InvoiceDetail {
  amount Double

}

relationship OneToMany {
  Invoice to InvoiceDetails
}
```

Figure: ***JDL-domain-generator*** takes a JDL file as input. It generates a ***.domain*** file for each entity. Developer can specify grouping by adding comments in JDL file.

Steps:

1. ***JDL-domain-generator*** will also specify all the next level generators to run next step. The selection of next generators will end up generating an application like jHipster classic makes today. (As much as we want them to be identical).
2. Developer can change generators to match the requirement. In the above figure the sample ***InvoiceBasis*** is not a table. Therefore the developer can change generators for that domain and remove ***database-migration*** for that table.
3. JDL-domain-generator can/must be re-executed and due to ***three way merge*** the developers changes will remain in resulting domain files.
4. The developer will need to implement / overwrite the generated kernels to make the wanted logic.
5. The developer can introduce doman file that is not sourced from JDL.

Domain files are located in ***src/main/domain*** directory. Needs to take full advantage of ***genlog***.

Litterature

(1) JHipster Lite Presentation

- https://docs.google.com/presentation/d/1i0LOJ0GSWNG2-x0zY220IbQc0PVQ2pndQWEuQKGu8n0/edit#slide=id.g5b8c73bad6_2_90

  Presentation of problems in jHipster classic and the reasoning for making jLite.

  Simple WebServices with JHipster Lite by Colin Damon

(2) Simple WebServices with JHipster Lite

by Colin Damon

- https://www.youtube.com/watch?v=mEECPRZjajI

  Presentation of an invoice generating application based on invoice basis.