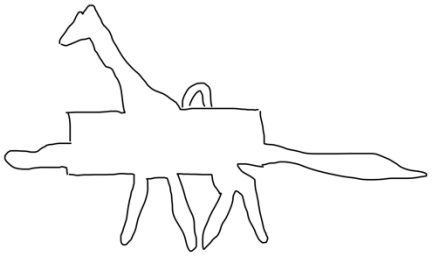
Configurable Multi-Tenant Applications in a Global Agile Enterprise

Morten Heine Sørensen
mhs@formalit.dk

You claim to have invented a general suitcase
that can transport any animal.
But what you have in fact done
is to look at giraffes and alligators
and then invent a giraffe/alligator transporter:



P. Urzyczyn, 2007
Drawing by the author based on memory

INTRODUCTION

As companies become increasingly global, so do their software applications. Imagine a company which starts out as a national company focused on a national market. Over time, the company increases its geographical coverage through acquisitions, mergers, and expansion. Eventually it may cover most of the world.

In the resulting enterprise, there may—for obvious historical reasons—be a plethora of software applications that solve the same needs for a given business domain. These overlapping applications originated in the constituent companies out of which the global enterprise was formed.

The idea naturally emerges to either buy or build an application that can be used through-out the countries of the enterprise for some business domain.

For the most basic problems that are common to many companies, such as billing clients or paying employees, standard solution are readily available.

For the more specialized areas of the enterprise, related to what the company actually does, it may be desirable to build a custom application that meets the requirements.

The reader can easily imagine other scenarios than geographic expansion that will result in the same fundamental situation in which an attempt is made to build a custom application for a business domain to be used by different legal entities with non-uniform requirements.

We refer to these legal entities as *tenants*, whether divisions of the same global enterprise in different countries operating in the same business domain and using the application internally, or different customers within the same business domain using the application provided by some company.

In this paper we present challenges that may be encountered when building such a shared application in a global enterprise. We also present principles that may be used to address the challenges We describe each principle in turn; some deal with what you should *not* do, others with what you *should* do.

1: NO CODE BRANCHING

Imagine the application is initially built for the first tenant, not knowing how the requirements will vary for other tenants.

The application requires users to

- Register
- Provide some data
- Upload some documents
- Sign some documents

Employees of the company approve or reject the uploaded documents (which in the latter case must be uploaded again) and eventually approve or reject the users.

For approved users, their data and documents are sent to the backend system of the country. These are typically different from country to country.

Sooner or later, variations in requirements will surface, resulting in the need for changes that are inconsistent with the existing application.

For instance, the following may be relevant for a new tenant:

1. The data to provide are different.
2. The documents to upload are different.
3. The documents to sign are different.

Some more subtle cases may include:

4. If the user enters a 9-digit phone number, automatically add a “0” in front.
5. When the user enters social security number, it should be validated by an external service.
6. The user should not sign any documents, this is handled in another system.
7. At the end of the process, when sending data to the tenant’s backend system, first send it to one system, get the result and send that together with the original data to another system to keep the two backend systems in sync.

One possibility for handling such variations is to fork the code into separate branches and make individual changes for each country in the branch of the code that they belong too.

The advantage of this is that there are no dependencies between the branches. For instance, a change in one branch does not require retesting for countries in other branches.

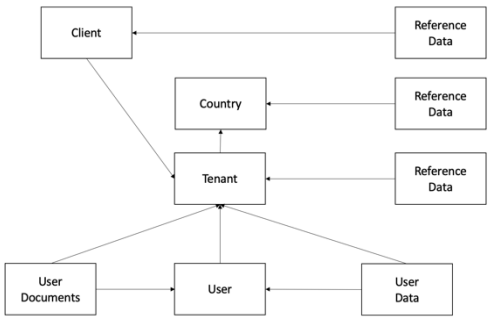
However, there are many disadvantages too:

- Adding shared enhancements to the application requires extending and testing each code branch.
- If, as in our case, the requirements may vary for each tenant and even each client of the tenant, the number of branches would explode.
- Essentially, we would be developing many different applications, thus requiring many developers.
- Deployment, general management of environments, etc. become more cumbersome as the number of branches increase, even with automation.
- The basic idea of a global, shared application and its benefits are lost.

Thus, we recommend, despite conflicting requirements, to stay with all tenants in the same branch, i.e., *single code line.*

It may be argued that retesting all tenants when changes are made can be handled by automated regression tests, either unit tests or integration tests. In practice such unit tests or integration tests may not capture the errors we get, may be excessively time-consuming to maintain and may be difficult to keep stable. But we will later see how the need for regression tests can be minimized.

To deal with multiple tenants, we build the concept of tenant into the data model of the application, by letting each entity have a reference (foreign key) to the tenant.



So, the application has a representation of the user, the user’s data and the user’s uploaded documents, as well as the documents the user has signed. All these objects live within a single tenant.

Every tenant belongs to some country and may have a number of clients. The users belong under the tenant but are also related to one or more clients of the tenant.

There is also reference data:

- For each tenant, e.g., a set of physical branches
- For each country, e.g., a set of regions
- For each client, e.g., a list of sites

The list of countries in the world might be seen as a global list, since it is really a property of the real world. However, it turns out that for political reasons, the list of countries may vary from country to country (not indicated in the model).

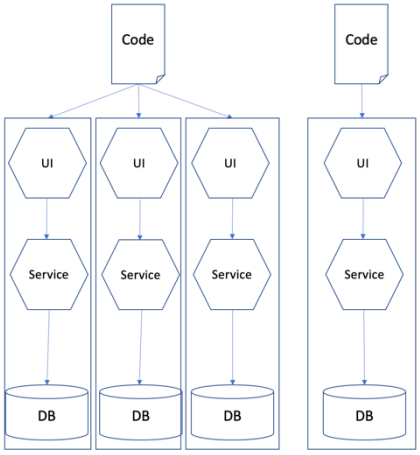
In conclusion, to enable single code line and to ensure isolation between data for the different tenants, we build the tenant into every table of the database. Moreover, we allow the code to have variations depending on which tenant is executing the code.

In the next section we make a brief excursion into a side topic and then return to a discussion of how the code variations for a given tenant may look.

2: NO ENVIRONMENT PROLIFERATION

While the code is single line with no code branches, it could be conceived to have deployment to separate environments for each tenant, or for some groups of tenants. Alternatively, a single environment can handle all

tenants. The two situations are depicted in the below figure.



In the separate deployment, there is a separate deployment instance for each country, or group of countries.

In the single instance approach, the single instance recognizes for each URL, which tenant the user is executing under.

A variation (not shown in the diagram) is to have separate instances for UI/Service, but shared database.

We prefer the simplicity of the pure single instance approach, although it does not ensure performance-wise isolation between different tenants unless additional structure is imposed.

3: NO CHECKING ON TENANT

We now return to the topic of what variations should be permitted for the tenants in the code.

Consider, for example, the requirement that for a new tenant, the user should not enter middle name.

The most direct approach is to insert a check in the code of the page where the input field occurs, which will omit the field for the given tenant.

However, as the tenant specific requirements multiply, it becomes increasingly difficult to manage them.
For instance, the middle name may initially have been an optional field, so hiding the field has no bearing on validations, but if the field is later made mandatory, we have to remember to make it optional for the tenant where it is hidden.

Also, as new tenants emerge, they too may want to skip the field, which requires coding again.

Thus, the code becomes more and more cluttered and increasingly difficult to maintain without unintended side effects between one tenant and another.

The problem is that this approach is *ad hoc* as opposed to *generic*. It is a giraffe/alligator suitcase rather than an animal transportation device. It leads to an arbitrary collection of cases whose logic is cramped together in the code without any attempt to identify some kind of generality that uniformly covers the individual cases.

We should not rely on this case-by-case approach which checks the tenant and does something very specific, but instead recognize the pattern and generalize the approach to capture the pattern.

This is not just a matter of striving for generality due to aesthetic preferences. As mentioned, the accumulated customizations make the code increasingly difficult to maintain without unintended side effects. But more importantly, the release of each new tenant requires significant development, testing, regression testing, etc.

What we should do is to introduce the concept of a *configuration* for each tenant. That configuration should explain in some general way how the application behaves for this tenant.

However, we are not safe yet.

Imagine we introduce the feature “omit middle name”. Then this feature can be disabled for all existing tenants and enabled for the new one.

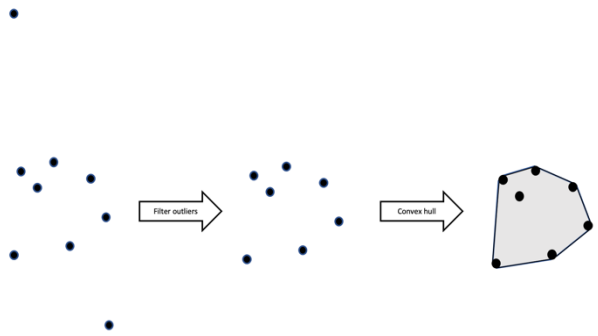
But this way we have not accomplished anything substantial compared to having the direct check for the tenant in the page with the middle name. Those configurations we define should be more general than simply saying yes or no to an array of tenant specific customizations. We elaborate this in the next section.

4: THE CONVEX HULL

So, our configurations for tenants should capture some kind of generality, rather than dealing with a range of special cases individually.

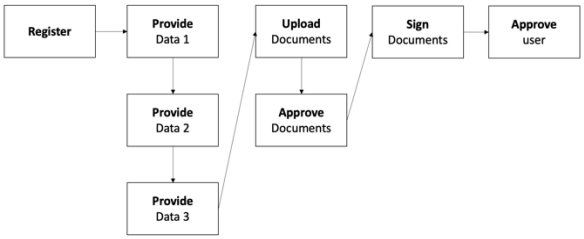
Moreover, some special cases should be disregarded entirely, when they are too specific to make sense at any level of generality or where the generality would become too elaborate.

In other words, we disregard the exotic special cases, and generalize the remaining ones by considering not just those cases, but also the cases that lie in their span, i.e., what in geometry is known as the *convex hull*.



For instance, we do not just make the middle name optional, we make *all* the *fields* optional, so that they can be included or not in a given configuration. Also, we make each of the *forms* with fields optional. Similarly, the documents *to upload* and *sign* are all optional.

We may go on and make some *steps* optional, e.g., omit document uploads and signature of documents as a whole. Also, the employee steps of approving either individual documents or the user as a whole may be omitted. So, in the below overview of the application, most steps are optional, with the possible exception of registration.



The question arises how many special cases we know up front. When we start building the application, we may know the requirements for several tenants. Or we begin with a single one. In both cases, we will usually need to make the application increasingly general and configurable as we learn about new special cases.

Notice that the convex hull is not just taking the union of all elements. For instance, if tenant X has a radio button for a field and tenant Y has a drop down where two of the values are those that tenant X chose between, the two fields have to be harmonized into one, which for tenant X would have two values and for tenant Y would have more.

In the next section we again make a brief excursion into a side topic and then return to the discussion of how the variations for a given tenant may look.

5: 2ND ORDER PRODUCT MANAGEMENT

How come the tenants of our application do not have the same requirements? Indeed, in a perfect world with full alignment of business processes, the approach might be the same for the different tenants, if they are all under the control of the same global enterprise. In practice, however, this is often not the case.

Speaking of requirements, the typical steps in a sprint, from requirements to running code, look like this:

1. User stories (product management)
2. Development (dev team)
3. Testing (QA)
4. Release (devOps)

In many projects, refinement of Step 1 boils down to describing web pages to be implemented accompanied by an explanation of how each element of the pages works. The design of backend services and database tables from this can be done by the devs.

The problem with this approach in the current setting is that it takes the perspective of a user *for a single tenant*. But when building the application to be configurable, it is about specifying not what the application should do for a given tenant, but rather *how* it should do it. It's not about the externally visible behavior for a given tenant, it's about the internal organization of the features in such a way that they can be composed to arrive at different results.

It's not that some tenant wants to omit middle name. It's that all the relevant fields are selected explicitly for each tenant's configuration.

In other words, we need extra time between Step 1 and 2 to analyze and design the desired configurability. And we need this done by folks that understand the internal workings of the application. And we need data to form an opinion about the initial version of the application and its generality, and if not available, we must simply start out with tenant 1 and prepare for configurability as we see fit.

Notice that, since we disallow checks on tenants in *code*, we should do the same thing in *requirements*. In a sense, this the goal of the extra step between steps 1 and 2.

Finally, an important thing to understand is whether it is acceptable to fix a scope and simply not be able to cover all possible tenants and clients. For inconsistent requirements, this will provide us with the option to either

ignore them, align them, or implement them via configurability.

6: ZOOMING OUT

Having seen several tenants allows us to define the set of user data fields that are relevant for one or more of the tenants. Maybe we can even envision additional specific fields not used by any of the current tenants.

However, as additional tenants emerge, they may require new fields not yet defined. If configuration only allows selection of pre-defined fields, coding may never end. In other words, it may not be possible to conduct the exhaustive analysis that the final application requires.

Instead, a possibility is to generalize the application to cover *whatever* requirements would come up, i.e., to allow the configuration to specify *arbitrary*

- Custom steps
- Custom forms in standard or custom steps
- Custom fields in standard or custom forms

This can be seen as an increase in the level of abstraction. The abstract approach has the obvious benefit that it can accommodate any field, form and step that the tenant comes up with.

The downsides are that we lose the ability to have specific validations or other behavior for the known standard fields. Also, navigation and associated concepts (like filtering the search for users based on their current step) become much more complicated. Finally, the amount of configuration needed in a fully abstract approach in which all field are generic, would be greater than in an approach based on common standard controls.

So, the best option is a combined approach with a set of common standard fields, forms and steps as well as the ability to add custom fields, forms and steps.

Incidentally, the implementation of the standard fields may have few differences from custom field of the same type, so we might from the start aim for the fully abstract approach. However, it is in any case useful to have a set of predefined fields, so the configuration does not have to be done from scratch. Also, the focus on specific standard fields crystalize the different kinds of validations that are relevant to consider.

Besides, there actually may be some differences between standard and custom concepts of the same type.

For instance, the employees may have functionality to filter users based on the step they have arrived at and specifically want to distinguish users in the step where they have to upload documents depending on whether or not they had a document rejected. Taking into account that the upload documents step is standard step, we can create the desired filter directly.

In conclusion, we can view the possible levels of abstraction as a scale ranging from one concrete application for one tenant and client at the bottom to a general tool allowing the formation of arbitrary flows. The latter would be a kind of web framework or programming language. Between the two extremes is an extensible set of steps, forms and fields based on common scenarios combined with the ability to create custom variants of all concepts.

7: CONDITIONALS

Even with the more abstract approach, we may encounter central requirements that cannot be covered.

For instance, we may have conditionals in various cases:

- One field should only be shown on a form, depending on the value of another field on the same or previous form.
- Documents to upload may be relevant or not depending on the value of a field from a form.
- Documents to sign may be relevant or not depending on the value of a field from a form.

These can be implemented as a general concept of conditions referring to a field and a possible value. That is, in the configuration for a tenant, it may be specified that a field, say, middle name, is only to be shown depending on the value of some other field. A more plausible example is that the documents to upload depend on the nationality of the user.

A slight variation is that the possible values of one drop down may be filtered depending on the value of a field from another field on the same drop down.

Through-out the application, wherever things may depend on the user's entered data, the same concept of conditions is used in the configuration.

8: VALIDATIONS AND PLUGINS

Another set of requirements is about validations. For instance, it should be possible to specify certain validations for any field in the configuration, e.g.

- min/max values (for numbers)
- negative/positive distance from today (dates)
- min/max length (for strings)
- regular expression (also for strings).

But for some fields, like social security number, the validations may be so specific that it is hard to see how they could be the result of a general kind of configuration.

Therefore, it may be relevant to introduce the possibility for a custom validation that simply refer to a devoted class with code that computes the validation.

Similarly, in some cases, conditions involve something that is not another field, but rather a kind of internal field. For instance, some documents may only be relevant to upload for EU citizens. Theoretically, this could be obtained by a long sequence of checks on country of citizenship, but it would be better to introduce the concept of calculated field, based again on reference to a devoted class that does the computation. So, as a special case we may have the internal field isEuCitizen with a custom class doing the calculation based on country of citizenship.

Another variation is that we may allow the values of a drop down to be populated by a custom class, intended for cases where the possible values would change frequently, so maintenance would be tedious, unless we have integrations that periodically or based on events update the list of possible values.

9: NO REGRESSION TESTING NEEDED

The vision of the application is that it should be fast, say a couple of days, for each new tenant and client to become onboarded to the solution.

This means that it should be a matter of configuration, not coding additional features, for the new tenant/client.

But as mentioned, we should allow for extreme cases, the concept of plugin, which requires coding and even new release of the code.

Ideally these classes should be released in a separate smaller release of those classes in particular.

However, the most important point is that the development of such plugins would not require retesting of other tenants or clients, since they are totally isolated and can only impact a change for those tenants where they are actually used.

In other words, there can be no unintended side effects.

10: REFERENCE DATA PLUGINS

Most business applications have a data model that can be split in two parts:

- Transactional data
- Reference data.

The second part can, in turn, be divided into two parts again:

- Values for the drop downs, e.g., salutations ("Mr", "Miss", etc.), post codes, etc.
- Basic concepts about the tenant and the client, e.g., the list of branches.

A special problem concerning how to deal with different tenants and clients without coding is that their reference data may be structured differently.

For instance, for one tenant and client, the branches of the tenant may be coupled with the sites of the client. For other tenants or clients not.

It may not be possible to decide how in general the reference data would look for arbitrary new client. The solution is to introduce the concept of a reference data mapper. During creation of the user, the mapper, which can be different for each client or shared, the code for the configured mapper will be run to set foreign keys from the user, to whatever reference data is relevant.

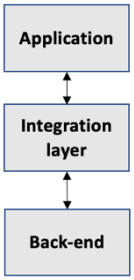
Everywhere in the application, it must be ensured that the application makes sense regardless how much of the reference data is available.

11: INTEGRATIONS

A main part of the application may be to send data for the users to backend system for each tenant.

Generally, we may distinguish three integrations and they all consist of messages sent back and forth between the application and backend system:

- Data about the user
- Data about the flow
- Data about the documents to upload or sign



For different tenants these issues can be expected:

- Data is sent at different stages for different tenants
- Many customizations are needed towards existing backend systems, including mapping between the data model of the application and data model of the backend system.

The first point may be dealt with by having integrations configurable in terms of which step it happens at.

For the second point, it should not be the purpose of the application to know about details of the internal model of backend systems, hence these tasks should be relegated to an integration layer. In the worst case, where this cannot be done, we could envisage another plugin to handle the mapping for a given tenant.

12: AGILE PRODUCT DEVELOPMENT

We conclude by describing a process by which the application can be designed, implemented, and tested.

We first provide an overview of the approach to the design of the application, i.e., what we have called 2nd order product management. The starting point is the variations for the initial set of tenants.

Step 1: External analysis.
Analyze every field of every form of every step for every tenant and nominate one tenant as the reference point. For the others identify every visible difference.

Step 2: Categorization.
Categorize the differences as accidental or essential to set standards for level of configurability. Some differences may be due to different taste or similar concerns. They are accidental.

Step 3: Identify different ways of handling variations.
For every variation consider these possibilities:

Alignment
This boils down to either promoting or rejecting the variation:

- *Promote*: accepting a variation and making it common for all tenants.
- *Deny*: rejecting the variation and forcing the tenant to work like other tenants.

This option should be used for all accidental variations.

Configurability
This amounts to viewing the reference point and the variant as two different instances of a more general scheme, probably with many more instances than these two, and allowing each tenant to configure the choice.

This option should be used for most essential variations.

Integration layer
This delegates the task of bridging the gap in expected integrations behavior between the global application and the backend systems for each tenant to the integration layer.

This option should be used for essential variations, where separations of concern dictate that the solution should not reside in the application.

Plugins
This is the last resort when trying to capture all variations, which is to allow the possibility to write a devoted class for a certain task and configuring the application to use it in certain places in the application.

Step 4: Identify commonalities
Identify common (i.e., *standard)* steps, pages, fields, etc.

Step 5: Identify custom variations
Identify *custom* features, steps, pages, fields, etc. and how they are defined.
Step 6: Identify need for additional structure
This means identifying fields depending on other fields and similar structures that may be missed on the first coarse-grained overview of differences

Next, we consider the development of the application. If the application is already developed with checks on tenant in the code, there are a few fundamental decisions:

- New repo, or same repo as existing tenants
- New environments or same environments
- Start from scratch or surgically refactor the existing solution piece by piece
- How to proceed with intricate refactoring

New repo
It is probably easier to get rid of accumulated debt by changing to a new repo. Users of existing tenants on the old application may eventually be migrated or would seize to be users due to the nature of the application (users are expected to be short-lived).

New environments
Having decided on new repo, it is most natural to decide on new environments as well.

Refactor vs start from scratch
We do not want to start from scratch. The application is probably developed over some time, many defects have been identified and fixed, and much knowledge is built into the code base. On the other hand, the handling of variations is ad hoc and should be eliminated.

We suggest an in-between approach where we build a new application, but each field, form and step is assembled from existing components and revised.
Approach to refactoring
Trial and error with the refactorings:

- Find out where to change
- Find out what to change
- Do the changes
- Then test
- Discover errors
- Then do more

Of course, the process is iterative, and design and implementation are heavily intertwined, and the level of configurability should be increasing over time. Things start out hard-wired and end up fully configurable; along the way, they are *half-wired*.

Finally, concerning test, if the configurable application is based on previous hardwired tenants, create configuration for the most advanced tenants and then test that we can reproduce the behavior of the original application using appropriate configurations for the new application without any tenant checking.

This is a powerful idea that can drive the development of many small parts of the new application, such as calculated fields, custom validations, etc.

CONCLUSION

We have explained how to build a configurable application in a global enterprise. Many details have been left out, but can probably be figured out by the reader.

Acknowledgements. I'm indebted to my co-workers, especially Andrija Pajic, for discussions on design and implementation of the application and its configurability, and to Nikola Schou for exchange of ideas about multi-tenant applications and deployments.