ERPort and Dust Framework – is it possible to excel Excel?

Options for managing our data

The reason behind all software is to build a model of a system, and manage its elements: edit items, calculate values, visualize results, prepare or execute actions. But how can you do this?

Find a tool...

You can use an **existing 3rd party solution**. It gives you some comfort, you are able to compare multiple solutions and chose the best alternative. You can also balance between the price and features, from free or freemium tools to the relatively expensive industry standard solutions.

However, it is generally hard to decide if it fits our real needs; there is a risk in quality, support and documentation; it brings platform and tools dependency, and in most cases, you will have integration problems.

We can do it better...

You can also **develop a system** to have all your needs solved, but that is very expensive and risky; adds the vendor to the platform, tools dependency; still problems with integration and later modifications. Theoretically, local solution should be the best in the long run, but that is not true, the real result is that above a certain limit in time and size, the failure rate increases.

"So my basic proposal is that as systems get bigger and more expensive they get more complex and complex things are harder to deal with and therefore more likely to fail. So if the system is under, say $750,000, it has a good chance of succeeding. Once it approaches $2 million it has less than a 50% chance of succeeding. And by the time it gets much larger than that, the chances of success drop to near zero."

John Dix: Complexity of IT systems will be our undoing (Network World, Nov 3, 2010)

Just give me an Excel sheet...

Excel is a natural solution, because it is "already there" in the office; you can *store*, search, organize information in it, add adequate *business logic* (value sets, expressions and calculations, …), and it also has sophisticated presentation layer (coloring, aggregation, charts). Adding new values, data tables, logic and report requires basic "office" knowledge, so it is relatively cheap.

What is the problem then? In short term: integration is still a problem (requires complex scripting); shared usage can cause problems; "easy changing" also means less controlled modifications and errors. In the long run, the tasks generally tend to "outgrow" Excel: data rows and columns grow with time; the data structure and script modifications become too complicated; there are many partial or outdated reports and charts. In a typical company, there are many Excel data containers that are hard to synchronize (data life cycle, standard report structure and design).

Summary

	Excel	3rd party	Local development
Price	“no”: Office is already there	Free to expensive	Expensive, generally underestimated
Introduction	“no”: already there	Standard “known” installation procedure	Iterated deployment / testing / adaptation rounds, generally underestimated
Creation / change knowledge	Cheap (advanced office user)	Free to expensive (trained experts)	Expensive (experts must learn local implementation)
Custom content (data, logic, presentation)	Excellent	Opposite to price	Generally good
Adaptation	“Too simple” (good and bad at the same time)	Opposite to price	Generally bad (the more complex system is harder to change)
Reliability (bugs)	“no”: it knows what it offers	Opposite to price	Risky, generally underestimated
“Outgrow issue”	Serious problem	Opposite to price	Can be excellent if properly planned
Structure	Hidden, generally a problem in the long run	Generally provided by the vendor (good and bad)	Can be excellent if properly planned
Integration	Hard	Hard	Hard
Shared usage	Generally problematic	Generally good	Depends on planning

Consequences

It seems that IT industry, despite of several decades of development, enormous growth in computing and networking capacity, millions of experts with specific education worldwide, tons of tools, methodologies, languages, platforms and acronyms, can't solve user requirement in a reliable way. This is even stranger, because in IT, we don't have to deal with the degradation of materials, wear under physical load: whatever we create works as programmed, as long as the running infrastructure is operational, in some cases for 10+ years without restart.

The summary table clearly shows that **Excel has its right to be the de facto standard tool for data management**, at the very heart of most companies. To make it short: we, developers are too expensive, nearly not as reliable as we think of ourselves, and tend to fail in the long run just as Excel does. It is a rational decision to stick to the simpler, and at start less expensive solution that later can also be fixed by using cheaper experts.

Our quest

Excel has its weakness in the table. An approach that 1: can target some of them yet provide adequate solution to the rest, and 2: works on a market segment that uses Excel but has problems with its limitations can have a chance to enter that arena. We split our efforts accordingly by creating a data analysis assistant tool: ERPort, and Dust Framework that provides the infrastructure responding to the problems of development listed above.

ERPort is a tool for consultants. They go to various companies under a time limited contract, they have to collect huge amount of data in weak structure, clean and organize it, then convert to a target structure, derive meaningful knowledge out of it, etc. They use Excel for its flexibility, but hit many of the limitations (data amount, complex structure, management, integration). ERPort gives them the ability to store, manage, search “any” data; analyze and transform the data to the target formats; support server-client operation and multiple users; later it will also be used to organize the project workflow (collecting data, making decisions, etc.)

Dust Framework provides data structure management, persistence and server-client data communication without coding (part of ERPort operation); adaptive automatic user interfaces (grid, property sheet for data); custom search on any field and expressions; data import and export (primary source and target is Excel, so whatever the user may miss from ERPort, can still use all features of Excel).

ERPort in itself has a serious market potential. In our team we have 20+ man-year experience in SAP consultancy, and valuable partners with similar requests. In this case, “we eat our own dog food”: ERPort is a tool that our consultant owner wants to use in his own projects. The system will incorporate his unique experience and tested methodology in its functionality and starting templates, but also the requirements from our partners who are our first customers.

Dust Framework is the engine behind ERPort, similar to a graphics or physics engine behind a 3D game, a product on its own that can be used in many other applications. Among others, it provides configurable data structures and managing storage, search, server-client communication and user interface to them. This is a significant, 20-50% of any system code, from database table scripts, language objects, data loading and saving, etc. Even though this is supported by many existing frameworks generating codes, table scripts, user interfaces, it is still a measurable part of any development, the product must be tested – and later, it has to be generated and tested any time a data structure changes. With Dust Framework, this part of the process is automatic and reliable, so reduces the development cost, system flexibility and stability.

Introducing ERPort

Scenario

Company A buy company B, the data must be migrated.

Typical numbers of IT systems, tables, data amount to be listed.

Core features

Uploading files, managing, reopening them – keep all files in their original format, know when received, can search and display them any time. The same tools for any files (images, reports the same way as data files), support mass uploading.

Import – plain Excel, but any custom text, or known data format can be integrated easily. During the import, meta descriptors are created automatically. Main task is to import all data, have no configuration or format error, you must see all data in ERPort with the same name and content as the client sent it. Can search by type, content and import time.

Analyze – see the imported data, identify value sets, times, etc. Create calculated fields with transformed data (Date type, transformed value sets, relationships). Later: analytic tools: aggregations, charts, exception finders, etc.)

Transform – populate target tables from the sources. No transformation is allowed here, only field matching. Search, analyze this table the same way as the source.

Export – create Excel (or any other format that you can create template for) files for further processing. This can be creating complex reports in Excel, migration scripts to target databases, etc.

Repeated operations – the components above (import, analysis, transformation, export) are “objects” in the ERPort project, consequently, they can be repeated: you can import another input data file with the previous configuration (or customize it for the modified content); re-populate the target table, re-export the content, etc.

Extended operation

Presentation – you can share your project with customer managers, so they can survey your findings. Very important when you have multiple projects, and hard to manage appointments with the responsible people.

Direct content access – you can configure ERPort to provide transparent access to some of the files stored in the project (images, reports, etc.). You can then use ERPort links to these files in exports, if you don't want to export to share those files.

Task management – you can assign tasks to external partners, like acknowledge a finding, make a decision, create a new transfer file with modified content, etc. They can do it in the same environment, and you can see their cooperation, plan timeline, etc. Essential for a short deadline project.)

Object export-import – you can save and load or drag and drop any object between ERPort projects (assuming that their references are also available). This means you can either reuse object (like a client contact person), a configuration (source or target type, optional values of a field), an action (import, calculation, transformation, export, report).

Templates – we create and provide standard templates for the most common actions, so you don't start an analysis project from an empty project, but a pre-configured object set created by our experts, holding years of experience in the same actions.

Clone projects – you can also create your own standards, and for example, when you go the the same client with a new contract, just clone and purge a previous project, that is already configured with all the types, contact information, etc. to the client. Even client companies can create their own ERPort objects for faster initiation of projects.

Benefits of ERPort

A single, central location to manage all operation and data – compared to keeping these data in emails, folders, different files, never knowing where is the latest version, what is missing and whom you are waiting for.

Transparent view of the complex data structure: tables, their sources, relationships, possible values.

Parallel operation: instead of switching between sheets, you open, drag, drop, customize any number of views on the same data, compare items, create aggregation reports on ad hoc subsets, etc.

Repetition of actions: you don't have to remember the settings, expressions, process chains, they are all saved in the project, you just activate and update when you need them.

Cooperation support: your clients can review the progress whenever it is convenient for them; you can create, organize, assign and monitor the analysis tasks as you go, all within ERPort.

Reusability: as listed above, your collected knowledge related to your tasks and clients, and also our own or community knowledge shared via templates, can be integrated and reused immediately in any new project. Nothing that has value is lost and should be re-created when you finish a project in ERPort.

ERPort, the data analysis tool allows a more sophisticated operation, both in the actual project, and on company level at the consultant and the client. You have all data at reach and for sharing at one location; you can plan your actions and monitor the progress; can give immediate and asynchronous access to your results to the client management; and all your findings, configurations, helpful tricks are collected and reusable for you or your organizations.

Introducing Dust Framework

Software development have not changed much in the past 40 years, any improvement in its stability and efficiency is a great opportunity open for anyone.

The statement comes from Bill Gates (Boao Forum, March 29, 2015.), and Excel seems to be the most versatile tool for ad hoc data storage, manipulation and presentation. This indicates a fundamental error in our industry, like it would be seriously off the track for decades, as suggested by Victor Bret (DBX Conference, July 30, 2013.), who also presented an inspiring view of computer interaction as a thinking medium (EG conference on May 2, 2014.)

The comparison between Excel and development (either local and 3rd party) shows that there are segments where all are weak: long term survival or integration; and others where Excel wins by far: quick and cheap creation and adaptation, "complete solution". For clarity: Excel is not a villain or competitor to beat in this context, but the contrary: it is a brilliant, coherent collection of services that is in many cases better than what we developers can offer to our clients. Its features and our weakness analysis can show us the tasks that we must solve, or at least prove its potential solubility to enter the arena.

These tasks must be hard and the solution must require a fundamentally different approach, otherwise it would have been found and implemented in the past decades. First of all: regardless of any platform, toolkit, great experience and smart tricks, **if we developers want to be competitive, we must work less!** Because 1: when we work, clients must pay our time, and 2: we as all human beings make mistakes – the more time, the more mistake to fix later. Excel is the ultimate answer having zero programming for solving a data management task. Of course a custom system can be better, but only if we do what is absolutely necessary, and reuse the rest, and share whatever we do to increase our common efficiency.

Dust Framework is created around a genuine, coherent concept, to increase development efficiency and reliability, collecting 30+ man-years development experience in various areas.

Data management

For our clients, data is a flexible element; its structure or available choices change eventually, following business or operations modification. An optimal IT infrastructure can easily follow data structure changes but also enforces global system coherence (from raw data to yearly reports).

Excel allows us to do so just by adding a new sheet, new column, row to our sheets, and that's all. Of course, this is not a proper solution, easy to make mistakes and in the long run, completely mess up everything – but our administrators are very smart people, they add their human intelligence to Excel and keep a complete infrastructure working, regardless of many glitches in the system.

In development, we “translate” the structures to the infrastructure components (database tables and queries, source codes, user interface elements, etc.) and this operation takes significant amount of time initially. Later, if these structures change, the modification time is hard to estimate because apart from the actual code changes, we need testing; side effects, and the business logic becomes too complicated. The structure of the data is “buried into” the system source code and configuration, and this requires a lot of work on creation and modification. This “housekeeping” is only needed to “introduce the data” to the different running environments (programming language, templates, user interface), and only we, developers need, not the clients, but they have to pay us to make it before we could do the job they wanted. A total waste from their perspective.

With Dust Framework, this part is eliminated completely. Dust has a meta descriptor language (similar to many existing solutions); its data management (accessing values, saving and loading, sending it over the network to a client and back, putting it on a user interface, etc.) is controlled by the meta description of the data. Consequently, Dust can handle any data without custom code, it can deal with data structure modification not only without recompiling, but even restarting the running system.

Of course this is natural for Excel, you don't “recompile or restart your workbook” when you change a column – but takes a long time and management to have such a change in your custom IT system. However, Dust Meta management forces you to understand, plan your data structures, and consider the effects of your changes; and as the meta structure is transparent to the system, it can automatically check any side effects and referring locations that become invalid if you commit that change. This is the same as compile errors in a strictly typed languages, but configurable templates or script languages have no such aids and side effects can lurk in your system for a long time before causing a problem.

With the Meta (“DSL descriptor”) layer, Dust is more robust and organized than an Excel workbook, while it is almost as flexible – contrary to the very complicated change management of a software that can still be fragile in this situation.

Objects

The essence of any information system is to represent external entities with IT data items, manage their content and interaction. Today this requires a lot of work and still has fundamental problems.

These entities are a person, a product, a room, a time period, etc. ; the data items are programming language construct containing information about them. “Entities” can have various data structures, which is represented by class hierarchies, but that is still not always enough (multiple inheritance). An existing entity may “gain” or “lose” some of its attributes, like a man becomes a husband, hospital patient block is assigned and then removed from him (and as long as he is in the hospital, he “is” a patient). It is also possible that a data structure definition changes while many objects exist in the system “having” that structure already.

In Dust, the existence of all entities is represented by a common “Entity” level. Data structures are added to the Entity as Models. An Entity must have exactly one “primary model” that shows what this Entity is, but other Models can be added and removed from the Entity, in a controlled way. The content of a Model is given by the Type, which above the field definitions, also contains other, required types, so if an Entity is a Car, it must also be a Vehicle, and as a Vehicle it always has a Location. This is a requirement hierarchy, but in the Entity, a flat list of Models (this is how multiple inheritance is solved). Changing the Model list of an Entity is natural operation, and you can also enforce consistency by automatically adding required Models, or deny removing a Model that is required by another.

In traditional programming, we “create” an object, and then “load” data into it from an external storage (a file, database records, etc.) This is again a questionable representation of the fact that the “thing” exists outside our program, we should never “create” it. Dust see this from the opposite perspective: you have a virtually infinite “cloud” through which you can reach anything if you can identify it by its type, a unique identifier or known attributes. For example, you are looking for an Employee by a unique identifier, so you request the cloud to give it to you. That Entity may already be loaded previously, then you get that Entity; if not, then the Type configuration contains where the Employees are stored and how to get them. The new Entity is created in the cloud at this moment, loaded, and is given to you. You have not seen how it was created or loaded because that is not your responsibility.

Dust Framework uses a genuine approach that separates Entity (“a thing”) and its Model (“set of data as a specific object”) set. This way it can follow the natural changes of both the represented “thing” and the information system. This also solves problem of the traditional approach like creating and maintaining “data source codes” or multiple inheritance. You never “create” or “load”, but only “evoke” the Entities from the Cloud, which is responsible for creating and loading them.

This saves development and testing time both when creating and changing IT systems.

Service components

The real reason to create a software is to write a business logic responding to user actions or other events. Despite of our best efforts, this is again neither effective (we have many "housekeeping" tasks), nor transparent (the code, language, tools and coding flavors hide our intentions).

Service components also have a certain life cycle: they are created and initialized with some configuration parameters, then react to requests, and eventually are released. For example, to send a mail, you need connection information, some code that can actually send a mail or read the content of a mailbox, and actual commands to read the content of a folder, or send a mail.

By analyzing this example, you can see that the "data" part is mostly independent from the implementation: the mail standards define data objects, structures, field types, possible and default values, even the actions. A mail service must be declared and work exactly the same way on all platforms, with all possible mailer libraries. This scenario is already known and handled in Dust Framework: you can use the Data Management features to define mail service types; a configured mail server connector, or a mail is an entity with the proper Models containing the data. Consequently, the configuration part is handled by standard Dust tools.

The interaction is similar: the Services and their actual Commands can be declared using Meta configuration, and the Framework is responsible to call your relevant function to do the operation, in the example, to send the mail or read a mailbox content. As a developer, you can use a simple API to access the data you need (your own configuration or the content of the mail to send), and write the logic that does the job. It is also the responsibility of the framework to actually connect your implementation to the Service Entity, and create your mail sender object when needed.

Dust Framework Kernel itself uses several services for its own operation (for example, to load Types, or the configuration of your application), so there are common services already declared. Surprisingly, they are quite similar, like Stateful for services that are better be initialized once, boot time, released on shutdown, and keep running all the time under the application (like a database connector or similar "open" services), Processor for anything that only has a generic "process" services, Evaluator that have an "evaluate" feature returning a result, etc.

Many applications can be built only using standard services, but you are free to create any custom behavior. Your custom service can "inherit" multiple services, for example a template engine that formats a mail is Stateful (to load the template) and Evaluator (to generate the mail text using the actual parameters) at the same time.

Dust Framework Service management lets you define the custom service data types and behaviors, or reuse the existing ones. Having these definitions you can declare your service components by setting up their parameters the same way as you do with data object. This both shortens the learning curve (use the same interfaces, review and learn from the standard and kernel modules) and increases the transparency of your application.

Message communication

Communication between business logic components is a very delicate issue: function calls. One of the most sensitive segment regarding modularity, refactoring, layered design; not to mention call tracing, parallel processing, or returning multiple values instead of one.

Dust Framework builds on strict separation of service components: they can't “see” each other in standard scenarios, so you can't call a function of another service entity, the only way of communication is sending a message, and a Message is practically an Entity having Message as its primary model, containing information about the message itself: the command, target, and processing flags. All data content goes into other Models in the same Entity. This approach has many benefits for the performance penalty.

The message parameters don't affect the API anymore, because they are not listed in the function signature. You only have to recompile your component when the behavior (the command list of an implemented service) changes; the parameter change appears in the Type of the parameter model and handled similarly to any other Type change.

All message calls go through the single entry point of message sending in the Kernel, so you have the option to monitor, or even inject special activities to a certain activity (filtered by Service, Command, source or target type or even instance), even runtime. You can configure parallel operations or proxy mechanism here, for example, the mail sender is represented by a proxy Entity on the client; your code sends a sendMail command to this Entity, but if you are on the client, the proxy Entity forwards the message to the server, where it is executed – the same code on the server is the same, but the command is executed locally.

A message can have any Models and fields, and the access of those fields can be configured, multiple fields can be “out” or “in/out”, so the default limitation of single return value does not limit your code in Dust.

The other issue is problems with multiple service layers. For example, you want to send a message to all clients having birthday on the current week. The content of your message is a “filler” for message templates, but the clients have different default notification channels: some requested Facebook, others Skype, yet others email. For email, you also want to provide background template. This is not a problem, you populate the “send” function with a list of clients, the “filler” and the mail background template. The dispatcher only cares about the list of users, selects the default channel “sender” service and relays the received message; and only the mail sender checks if there is a background template parameter in the message Entity; the dispatcher and the other senders are not disturbed by them.

Dust Framework limits communication among Services to sending Message Entities. This approach solves many issues “out of the box” for Dust based applications, like multiple return values, transparent proxy mechanisms, even runtime configurable communication monitoring, or sending “hints” to other worker components in the execution chain.

Application structure

Finally you build your application from the service components. You have to use various methods here depending on the current platform, language, annotations, configurations, tools, etc. The result is very far from being transparent, easy to "debug" or reuse.

Using that all service components are represented by entities, the application is their interconnected network. This structure can be stored in any persistence store (local or remote configuration files, database, etc.), and edited also with the same tools that you already use for Type and Service configuration. This means, your application structure, "the plan" is not only transparent, but also organic part of the running application. In Dust, there is no "planning" and "development" phase, the Kernel actually executes the plan of your application, from the initial prototype to the final product.

If you examine what this plan means, you can see two layers. The "blueprint" contains the definition of all components with their complete configuration; this is independent from the actual platforms, or the modules you have chosen to execute the tasks. This structure allows you to create and reuse complete blocks, even to create a fully configured, tested toolkit repository to make your development even more efficient. You can also replace any segment here with mock data sources and loggers for testing.

A separate "deployment" segment allows you to select the actual implementation (for example, in Java, the jars containing the binary codes, and the assignments that connect the Services to Java classes). You can change the selected binary to change platform, GUI library, component implementation, insert a special version of the same code with performance monitoring, etc.

A very strong benefit of Messaging appears here: the blueprint can contain partially configured messages as well. Therefore, you may add "layer hints" to your blueprint as well, the running code does not have to know about it at all (like the email background template mentioned in the previous example). You may even have all Message instances in the blueprint declaration, which means the application will not need dynamic memory management, only value setting and message sending – a very important feature in low memory / embedded programming scenario.

In Dust Framework, your application is only another network of Service, and even some pre-configured Message Entities. In other words: Dust Framework "executes the plan" of your application, makes it transparent, easy to manage and refactor. This also helps organizing complete building blocks and create fully transparent test scenarios.

Further reduction of business logic

The operation of an information system can be separated to a complex and mostly static part (database, server-client, ...) and a more dynamic, simpler segment (user data validations, business logic), where clients focus on the latter, and want to change eventually.

Dust Framework focuses on minimizing the amount of "user level code", when the important segment is not the source itself, but the parameters it works with. A typical example of such parts are the expressions: we use them all the time, type into source codes of filters, decision making, search operations, validation, etc. However, in this way, we hide our actual requirements in complex codes, can make errors, hard to maintain and very hard to allow our clients to play with the parameters (like they would do in Excel).

Technically, all expression is a tree of simple comparisons or calculations, the compiler parses it, builds the tree and then generates machine level code from it. Dust contains a special module for working with expressions: Types required to describe the elements of this tree, and services that can execute the nodes. When you think of an expression, you have this tree in your mind – now you don't translate it to a programming language, but edit it in the same environment that you use for everything, and Dust will execute it. Of course, this does not a guarantee against failures – but makes things faster and cheaper for your clients.

The benefits are obvious. Your calculations are transparent, not hidden in complex source codes. You can allow your clients to change parameters, or even the complete calculations, so they can improve their system without you. Not writing code means total flexibility. Just create a new type and some Entities, like by uploading a data table in ERPort. You can configure any calculation or validation on this new data, and use it immediately with complete validation support, just like all other data types in your system, without even restarting your application.

A bit more abstract advantage is platform independence. If you have a server-client application with lots of data validation, it is better to be executed both sides, so the user gets immediate reaction on all changes, but the server is protected against incoherent actions. In traditional systems it may mean either code duplication or being forced to use the same language on both sides (fat client, applet, or JavaScript on server). But if you create your validation with configured Dust Expression, and the Expression module is available on your platforms, you will never have to worry about how to run or change them.

We don't have to stop here. Just like Expressions, running code also consist of simple elements like variable declarations, function calls, blocks, cycles, etc; your compiler translates the text file to a tree of these simple elements and generates the executable code. Is it possible to handle "programming" the same way as "calculation" with the Expression module? Yes, it is already done by all compilers, many workflow engines, etc. We plan to have this Process module in Dust too.

That means you will write code for the really heavy-weight, but fundamentally static components, like a database or mail server connector, template engine, application server, etc. You can safely share and reuse because they wrap the services of standard IT components, they don't change often, and when they do, that is really carefully planned and managed. On the other hand, you can use configurable Expressions and Processes to implement the business logic requests from your clients, thus they remain transparent and easy to modify even without knowing too much about programming and your tools.

Dust Framework allows yo to separate technically complex "internal" segments from the business logic requirements of your clients, and provides Expression and Process modules to implement them by configurations. In this way, the custom logic remains transparent, and even the client may modify it later.

Single entry point – Access Control, listeners, interceptors

The real life "things" have strict access requirements: you should not see or modify anything outside your competence. However, when we translate these elements to programming language constructs, this information is lost, and requires constant efforts to enforce.

In Dust Framework, Entities that represent the "things" simply never appear as "objects" for the programmer: the values can only be accessed using the Dust API functions, a single entry point where the kernel "knows" who you are: what is the actual component, who is the user trying to access that value. Consequently, the Kernel is in the perfect location to intercept any invalid read or change access, blocking the call or returning random values, logging the unauthorized request for development or security reviews.

Of course, this is not an ultimate protection, because when your code legally accessed sensitive information, you can still write it to a less protected location and break security rules – there is no protection against malicious programmers as long as they can use a programming language that has no data protection feature. However, with Expression and Process module we remain in Dust kernel level and all data movement is transparent. That means, as long as your custom business logic is implemented without low level programming, data protection can be enforced.

The same way, you can't call functions only send messages to other components, there is only one gate: the kernel "send" function. It can check the current process, the actual component requesting the action, the state of the system, the user, etc. It is possible to deny, log the unauthorized action or even set a silent alarm, direct the intruder to a fake "honey pot" segment, etc.

From a broader view, access control is just a specific application of the fact that Dust Kernel allows you to listen to data access and function calls. This is what you completely lose when using a programming language: if you have an object, you can directly refer to its fields and member functions. You never "get the control" when external code access your data or functions, and this is a painful issue, resulting in many coding standards and toolkits: "beans" with practically empty getter/setter functions, and tricks to hide this in template engines or languages like Objective C; Reflection; Aspect oriented tools to "listen to" a function call; etc. In Dust Framework, you just instruct the Kernel that you want to intercept a message or listen to a data change, again with the same simple configuration.

You have a single entry point to access data or call functions in Dust. The Kernel can be aware of any access control requirement, block, log or redirect unauthorized attempt. This is directly connected to the standard user management terms (Role, Group, User, …) by standard configuration settings. In a broader view, you can intercept any service call and listen to data change, and modify these settings dynamically in a running system.

Platform neutrality

The more complex system requires more complicated source codes and becomes dependent on many tools and language features. In the long run, it is very hard to separate and make it work in another environment.

Dust Framework is a very complex system, but platform neutrality is one of the key aspects considered in the design and implementation. The solution concepts is there in every book about programming: create components with single responsibility, separate and wrap external dependencies into implementation neutral black boxes, etc. For example, loading a JSON file, which can be done with a single instruction by many common libraries, is a task for 5 completely independent components work in concert: get a “path” and provide a stream; get a stream and provide string lines; parse a series of lines by JSON syntax and provide “SAX-like” events; process SAX events and build Entities; using a pool to cut and resolve internal references.

Dust Framework on an abstract level represents a new understanding of software development. Its main information body is the structure, represented by types and services required to store and execute any application, including itself. Its design process resulted a modular network of strictly separated modules. Consequently, its code base is tiny compared to its complexity; this means a relatively small effort to port to other platforms.

Currently most of its code is written in Java language level 1.3, except for streaming and collections where we have separate modules for 1.3 and 1.7; and natural exceptions like the Jetty servlet container wrapper or user interface modules. This means the kernel “should” compile and run on older machines, BlackBerry and android phones, tablets. We have experience in hybrid platform development in Windows C# and Apple Objective C; the same concepts already worked there.

IT industry competition focuses on the always new, always different platforms, languages and features. Dust Framework focuses on the never changing fundamentals of information system development, and has the power and allows you to wrap any specialties of any platform by using custom modules. Including those that the Kernel depends on.

Internationalization

Interaction between a user and an information system (log lines, user interfaces, reports, mails,etc.) requires texts, and with human languages, that means different encoding and values for different people – for the same context and data.

We are quite used to a kind of “bottom up internationalization”: typing constants and manipulating string segments in source code (including the notation of source file and compiler encoding), then extracting the constants to some configuration files. The same applies to user interfaces, typically HTML and JavaScript files, server side template engines, language dependent images.

Text manipulation is a typical area where configuration prevails over code, which is only: select a template to the actual language, replace the data placeholders with text representation of the provided data (which is again language dependent and can be a filled template again). It is also possible that we don't have the language parameter directly, but have to select by the preferences of the target person, even handling multiple template sets when you send a notification about the same situation to several people.

Dust Framework provides specific modules for handling text templates, formatting values; layouts fragments, etc. You can separate locale aware segments of your application from generic code and core data, and indicate the actual local information to any text data. Selecting the proper data and templates, building the final texts is done by specific Dust modules (that you can access via simple “util” functions). Of course, you still can manipulate strings directly in your custom code – but for internationalization or any basic text manipulation, you have full support in Dust.

Internationalization is a kind of stepchild in any IT system. We know about its importance and there are many toolkits and refactoring aids to help solving the problem, but it is still there and requires constant effort to keep all resources up to date. Dust provides a complete and coherent resource management solution that you can rely on.

Configurable user interface

We all know that if there is no catastrophic error in the data management, the system's value depends on the user interface ergonomy: how "helpful" it is for the users doing their job. "Smart GUI" tend to be complex, therefore hard to change or migrate.

The most noted acronym here is MVC, the Model-View-Controller separation; the Model being the data, the View is the user interface and Control is the interaction between the two. The Control logic tend to appear linked directly to the user interface events (value changes button clicks), either in source code or the now popular, fully solution dependent, incompatible GUI descriptor languages.

This is yet another typical situation where the "static, complex" and "flexible" data and logic is merged, resulting too much and complicated code, expensive and risky maintenance. To go back to the base, there are two separate systems here: "generic UI" is responsible for displaying data and allowing interaction with it – and "custom" representing the clients requirements: actual data structures, coherence and ergonomy logic and presentation / interaction needs.

The generic UI consists of the Types that can describe the container and layout hierarchy, and data controls, their data and event binding. This is the same for 90% of the features on all platforms, but you can extend it with custom types to cover exceptions. So, you can use a hierarchy of Entities of these types to fully describe a user interface. To make it work on an actual platform, you need the implementation that connects each Type (Panel, EditBox, Button, etc.) to wrapper classes in that environment, that 1: configure the widget according to the settings, 2: receives user events and transform the to the value change of the bound data value, and 3: listen to any change in the bound data and update the screen element (either the value or style, location, visibility, etc.). This is a very complex module, but have to be created once for each GUI platform, and any Dust application can appear on it.

In the "custom" system MVC, "Model" is the data, "View" is the user interface declaration built from the generic Types, and "Control" is the business logic responsible for data consistency, validation, error reporting, etc. The Control listens to the underlying data, so is called whenever any changes on the user interface. Its reaction can be changing other values, and also background information, like the status of a field, list of error messages, etc. The user interface is configured to react to these changes like by disabling a field or set the color to red.

The custom logic can either be written in a programming language, that means the logic can be active only in an environment that supports it (for Java code, fat desktop clients or thin web clients where the validation logic runs on the server on every change); or using Expression and Process modules. In the latter case, if the client supports these modules, any logic can be executed locally. Naturally, the client can still rely on transparent proxies to work with server-side service components.

Dust Framework separates generic complex GUI building and interaction from the custom layout and logic required by your clients. You can focus on the latter in a transparent and platform independent way; but you can still extend the GUI layer with any custom feature if needed.

Learning curve and vendor lock

Of course your have obvious questions are: the performance overhead, the learning curve and the vendor lock. We tried to minimize all of them.

First of all, the kernel is just like any other elements of the system, it should be configurable. Dust Framework kernel components use the same Meta descriptions and the mechanisms described later as the applications built on them. The Kernel loads its own configuration and then starts providing services to your application – if it is fast enough then your app should do fine as well. But if you are not satisfied with the kernel performance and have better ideas, you can replace our implementation through module configuration. OK, for this, you have to use our tools to generate some sources, recompile and restart the system, but it is still way simpler and faster than a Linux kernel build. In this case, you can also extend or change the Kernel type and service definitions, but of course your applications will not run with our Kernel implementation anymore.

Second, yes, you have to learn using our standard property sheet editor and grid finder panels, but that is completely basic. Then you also have to learn the basic terminology that we use to describe our system components. You can use the same tools to learn it because you can display and browse all items in the same environment, like the Type or FieldDef type. The important part here is that you don't learn complex, language dependent structures, only concepts, that you will also use when define your own types, services, components, applications.

The programmatic API is very small, the core is 4 functions and a few interfaces, and some convenience functions to cover repeated call constructs. The key here is that the Dust kernel does not provide actual services, only a way to access data and other services. All “real” services are organized in modules.

You will be “locked to” our concept of an application, the organization of data elements, and an absolutely minimal API that your code sitting in ModuleA can use to communicate with other components. Apart from that, you write “normal” code, in this case, POJOs, no annotation, .properties, .xml, etc. magic: you edit all settings in the same environment. You can change any module, including even the kernel when you need to; you can replace compatible modules anytime.

Benefits of Dust Framework

Software development, regardless of its unique environment, high performance equipment, tools, platforms, methodologies and legions of experts, is neither efficient, nor reliable enough today.

Working with Dust Framework forces you to create a clear vision of the system to be created. With the transparent Type and module definitions, you can see how your components will interact, what is the optimal segmentation of information across your Entities. You can set up your initial ideas, from the basic data structures, through validation and interaction logic, to custom user interfaces and reports, and start using your system right away, without even restarting the framework. You can continuously refine the operation of your system, replacing simple mock responders and loggers with real service components; configure and run test scenarios with zero or minimal amount of custom coding.

With this transparent layout, it is much easier to communicate with your clients, verify your solutions in an active system, spot misaligned understanding in a quickly accessible functional prototype. This reduces the “late refactoring needs”, but that still may appear eventually. However, the fully transparent configuration of system structure, data objects, service components, expressions and processes, user interfaces, reports etc. makes it easier to estimate side effects and decrease the associated risks.

With Dust framework, developers can create separate and truly reusable components for the complex technology components. It worth more development effort, and these components can be shared and traded behind actual client implementations. There can be parallel competition, both for contracts with clients and creating the best technical components. The final result is creating truly reliable components, and using them, much more stable client systems with significantly lower cost, as the implementation effort is shared among companies.

Today creating a true multi-platform application has a very high cost, appears as a serious limitation on the client features and may cause a maintenance mayhem because any modification means parallel changes in different, complex codes. With Dust configurable user interface modules, client business logic is reduced to the simple validation and interaction logic, which is much easier to maintain – or removed entirely by using configured Expression and Process Entities.

Dust focuses on atomic modules and homogeneous data and message management. That means that the task of connecting to an external system is again reduced to the bare minimum: wrapping an external interaction library (like a mail connector, a text to speech engine, etc.) into a simple, standard service entity. The result is a much lower cost of integrating external systems (a custom data store, event source or control mechanism). On the other hand, the flexible identification scheme allows sharing and integrating external Dust modules into your systems, so it will be more and more likely that the “external system” already has a Dust connector module, or even is implemented in Dust.

If we review the initial cost comparison table, it is evident that the listed primary features of Dust Framework address the critical problems. They allow significant reduction in actual programming work, which reduces the costs of any development and increases the stability of the system at the same time (we make less errors to find and fix). It enforces strict planning which never depart from the development process because the kernel always executes the current application plan. The new application model creates an open marketplace for really reusable technology modules, and that promotes a healthy competition among module providers for the benefit of our clients.

Current status and future

If we had all these components production level ready, we would already be in the headlines as the "next big thing" in IT. Of course, we are not there yet.

From the past, we have several previous versions of Dust Framework, and many segments of these concepts were actually used in various production systems. For example, a fully configurable, data and behavior bound, platform independent user interface layer is part of the national agricultural data management system (IACS) in Hungary. A Cloud / Entity / Model based data management layer was created and used in a release management process monitoring system at Continental Ltd., Break Systems Division. A hybrid platform configurable application structure with pre-configured messages and its complete user interface appeared in a prof of concept Java server – Windows C# client project. We also created a Java server, iOS Objective C hybrid solution for project monitoring, with advanced statistics views.

The current target is a more common hybrid system: ERPort is a Java server – HTML client application; the current Dust Framework development is controlled by the actual needs of ERPort. Currently we have a running Kernel with data management and message processing, a partial persistence management, a Jetty servlet container wrapper (the server is a standard Dust application, but its configuration contains a Jetty service component; the client can connect to it and request data, send modifications and commands to the other service components through the Jetty). The current deadlines don't allow creating the complete configurable GUI modules, so the first ERPort version is a "normal" web application with a limited JavaScript Dust middleware.

Our first aim in the future is to build the modules listed in this paper, but Dust should not stop there. The most significant further features will be the planning and development area: full support for meta management, Expression and Process libraries, Text and Resource modules. That will significantly decrease the learning curve for using Dust Framework as a component or application design and development platform. Together with a properly managed component marketplace, this can be a great boost to involve other companies and developers as component providers, and also source of valuable feedback for the further development.

We should also put serious effort on testing and monitoring. The kernel is an excellent location for monitoring data access, message paths, performance and resource usage. We should create various configurable test components that can quickly populate a test cloud; message senders, response validators, stress test source components that emit batches of configured messages, measures response times and resource utilization.

A very important segment can be the embedded environments. Dust Framework is not limited to "big hardware" and Java language; it should also have plain C implementation, limited kernel versions for small devices. The concept of configurable component cloud; message dispatching and processing exactly matches to home automation, internet of things or smart city scenarios. We have some experience with Zigbee (Ember DevKit) and Arduino home automation components, the similarities are significant.

The current version and development roadmap of Dust Framework follow the needs of the ERPort requirements and deadlines, implementing the necessary components as they are needed. With more resources, the listed features can be created faster, and there are serious opportunities beyond becoming yet another application building framework as well, for example opening a global component marketplace, or entering the Internet of Things / Smart City area.

Business concepts

The aims and potential of Dust Framework is far beyond the reach of a small startup company. This distance must be managed properly to let this system grow.

Our current resources are barely enough to follow ERPort development plan and create the milestones. This project combines the technical and business experience of our team, and will become a useful aid to data analysts. The market is relatively small: professional business experts working for major companies – but on the other hand, we have very smart buyers who know what they want and increase their efficiency by a flexible technology and good domain knowledge is important for them. Based on Dust Framework features our company can become a small but reliable partner for consultants, adapting ERPort or creating custom solutions based on the existing technology can sustain us for long time.

Dust Framework is a greater challenge. We are convinced that this technology can bring a significant change to software development, if managed properly, but we identified serious bottlenecks.

We must bring this framework to a level for public introduction. This requires 1: the listed main kernel features to be stable and well designed; 2: have it running smoothly on more than one major platforms, and 3: have the critical modules (database, messaging, entity instance management, synchronization) in good shape and performance optimized. To achieve this level we need technical help, consultancy, reviews on our design, and help from some experts at the target areas.

If the framework is accepted by the public, there will be a sudden increase in contact requests, ideas; most likely performance and code issues. This can give us a great boost and raise Dust Framework to a much higher level, if we can handle the load – otherwise this chance may slip away. At this point we should have an adequate background: materials, contact people, who can organize and manage support requests, judge and answer business ideas, etc.

And finally, if Dust Framework really becomes a global phenomenon, it should be backed by a global organization, either by growing or in cooperation with existing global players in IT business.

We have no secrets to keep. Dust Framework is not built on a fundamental discovery, all concepts can be found in many books on programming or system design; its code base is very small, can be copied to a floppy disk if you like; anyone could have written it in the past few decades. But nobody actually did it, because it is based on a genuine and coherent understanding of programming, where each decision literally collects several years of experimenting versions of these ideas in real life, big and small projects. It is easy to understand how it works. But we also know why not in other way, what is the roadmap for the next months and years, and how to deal with those tasks.

We know that our team can create and prosper fine on the ERPort project. We also know that we are too small to launch Dust Framework to the path it should fly, but we are more than ready to take this challenge and looking for partners to do this together.

Doubts? Questions?

You probably have seen too many nice features in this document; there should be a catch, it must be a vaporware.

Come and meet us at Craft Conference, April 26-29, 2016. Budapest.
Check the codes, configurations, give us challenges,
see how we deal with them in Dust Framework.

Thank you!