EBM-DC (eBanking Digital Cart) – Design patterns

Version: 0.1
Date: 2021/08/20

Created by **Russ Profant**

Revision history

Version	Revision Date	Summary of Changes	Updated By
0.1	Aug 23, 2021	Deleted legacy summary and created new summary	Russ Profant

Executive Summary

- Digital Cart (DC) module implements a simple core business functionality “the application for a banking product (currently deposit or credit card)”. This is also called a “flow” in the business domain.
- There are many different variations on this core functionality. The variations are always represented through slightly different UI screens and screen sequences.
- The current practice in DC is to add new variations of the “flow” to DC module by creating (mostly) new applications or “flows”.
- The main reason for this approach is that in the current design/ framework the screen flow is controlled in DC module instead of the UI.
- To accommodate the new screen flow there are two options in DC:
  1. *Customize existing flow (high risk, rarely used)*
  2. *Create new flow (little risk but extensive code duplication, mostly used)*
- This is becoming increasingly difficult to sustain and manage because the code duplication it rests upon (option #2 above - copying the whole application to a new code base) forces ever more code duplication (linear code growth) in the form of:
  1. *Maintenance - bug fixes must be done in numerous applications(see HTTP status codes)*
  2. *Enhancements – a single new functionality must be added to numerous applications (see VCI, e-Statements)*
- From the business perspective this trend is manifested through work duplication resulting in increased delivery times, increased costs, lack of resources/developers and corresponding scheduling pressures etc.

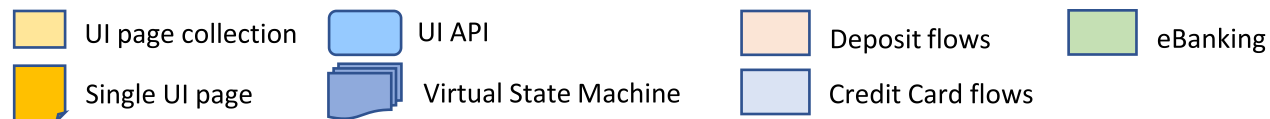
Diagram Summary of Current DC Design Practice & State

Product Page | Rates & Fees | Applicant | AML | eSignature | Cops | Offer | Cross-sell | Hard Stop | System Error

DFA	Smart Start	CCFA/Titan5	MTO TOO	Apply & Buy	Titan6	Another Flow
API	API	CCFA API / Titan API	API	API	API	API
VSM	VSM	VSM	VSM	VSM	VSM	VSM
Backend Helper	Backend Helper	Backend Helper	Backend Helper	Backend Helper	Backend Helper	Backend Helper

Compass

The diagram is not meant to be complete; it doesn't contain all the UI pages; it also doesn't contain AC flow and legacy flows such as OAO and SPCA. The purpose is to show the main trend in the current practice.



The lineage of the flows:
CCFA – mother of all flows; CCFA -> DFA; DFA -> Smart Start; CCFA -> Apply & Buy; CCFA -> MTO TOO; CCFA -> Titan6

Current DC Design Practice & State Explained

Notes on the previous diagram:

- ✓ “Flows” are full-fledged applications except in CCFA/Titan5 case where 2 flows are defined in the application
- ✓ A “flow” is made up of the collection of UI screens which are shared, the core application consists of:
  1. UI API – to enable communication between UI and DC
  2. Virtual State Machine – to manage the UI page flow and data exchange
  3. Backend Helper – to perform all API requests to back-end systems
- ✓ Currently the following “flows” are in production or development: CCFA, DFA, Titan5-6, Smart Start, Apply & Buy, MTO TOO
- ✓ ‘Virtual State Machine’ (VSM) is a flow-based collection of front-end-backing back-end virtual pages with one-to-one relationship to the UI pages that controls the UI page flow
- ✓ ‘Backend Helper’ is the component that performs all API calls, mostly to Compass
- ✓ Each API is a collection of API end points, one per UI screen
- ✓ ’Another Flow’ illustrates the unhealthy growth of this practice where the whole ”flow” is copied to a new project/application and then customized there

The Advantages of the Current Design

CHARACTERISTIC	DETAIL	BENEFICIARY
Precedent	It's already in production and it works. Therefore, the easiest way forward is to reuse it.	DC - encourages quick 'Copy-Paste-Edit' design and development
Simplicity	The pattern is simple and easy to understand. (API->Controller->Façade->ViewState). The same pattern is applied to all workflows. Anyone new who comes to the team can grasp it quickly.	DC - encourages quick 'Copy-Paste-Edit' design and development
Simplified UI	This is an "inversion of control pattern" that tightly couples back end (eBanking) to the front end (UI). The eBanking becomes the orchestrator of the UI flow. Therefore, all the logic can reside in eBanking where UI is little more than a dumb screen with some data validation and turn-on-off-field logic. This makes UI development a lot easier and makes it easier to test.	DC UI – no need for expert developers Business – UI development and testing is faster and cheaper
Form of versioning	The current pattern and practice can be viewed as a kind-of code versioning. The different flows/applications have different data elements which may not be compatible with other flows.	DC – no need to worry about versioning

The Disadvantages of The Current Design

CHARACTERISTIC	DETAIL	NEGATIVE IMPACT ON
Adding new functionality through code customization is risky	Customizing existing flow is very risky. These are production flows and adding new 'if else' statements to accommodate different conditions will easily introduce regression bugs into the code.	DC – Introducing high risk into development QA – extensive regression testing
Adding new functionality through code reuse creates code duplication or more accurately code multiplication	Reuse as 'copy and paste' of the existing code creating a new code base. With this approach we are duplicating the existing code base again and again increasing the maintenance costs and drastically increasing future enhancement costs.	DC - Maintenance of multiple code bases
With many flows, new features need to be added to multiple apps/flows	When a new feature needs to be added, such as VCI, it needs to be added to multiple existing flows. Sometimes this can be done in a single project, but often multiple projects are required essentially duplicating all the effort and the cost.	Business – cost are higher and delivery times are slower as "the wheel is reinvented" over and over
With many flows, bug fixes and code changes need to be done in multiple flows	When code needs to be changed or fixed, it must be done in multiple applications	Business – cost are higher and delivery times are slower

The Disadvantages of The Current Design cont.

CHARACTERISTIC	DETAIL	NEGATIVE IMPACT ON
Tight coupling of separate layers of the system	In software design and architecture terms, the current design breaks one of the most fundamental rules. **It tightly couples** two layers of software, the front end and the back end. It more than “tightly couples”, **it actually “fuses”** the front end with the back end.	DC - Legacy approach to software design mixing layers and responsibilities
Multiple flows with similar logic and code break DRY principle	DRY (Don’t Repeat Yourself) is one of the fundamental software design concepts. It’s being broken repeatedly.	DC - Breaking Modern Best Design Practices
When code is copied to a new application the existing functionality must be customized	When existing code is copied to a new base, the code is not ready and needs to be customized to fit the new project setup.	Business – cost are higher and delivery times are slower
Negative impact on dependent system primarily Compass	Compass has only a single client API. It’s very difficult to manage versioning when a single API serves numerous clients with slightly different requirements.	Compass - versioning becomes difficult to next to impossible with rising client count

Pros & Cons of the Current Design Practice in DC

The benefits mainly accrue in the beginning. When requests are mostly for new flows the benefits are obvious and significant using quick and easy 'copy and paste' approach.

The benefits are mostly in DC. Not much new design or development is required because 80+% of new work is just "copy, paste and edit".

UI benefits as well. Less work and the work is easier because it doesn't need to deal with page flow and application state since this is handled in DC.

The costs are mostly borne by business. The negative impact is felt by business in terms of actual costs and timelines and by DC in developer shortages.

As the flows multiply the costs rise too. When new requests become mainly enhancements, the disadvantages come to the fore in the form of effort multiplication.

The negative professional impact on DC. By breaking some of the fundamental rules of software design and engineering eBanking is not following industry best practices.

Conceptual Misunderstanding of the Domain

The Root Cause

The basic flaw in the current design practice comes from applying inappropriate design paradigm to the problem. When you have multiple variations of the same problem by using DDD you end up with multiple systems which is exactly what happened in DC.

Misunderstanding of the "flow"

The current DC design (Controller-Façade-ViewStates or CFV) is not based on good conceptual analysis of the problem. The main issue is the understanding or rather misunderstanding of the workflow or "flow" and where it belongs.

Non-Ubiquity of the "Flow"

Flow is not a good ubiquitous (DDD) concept because it means different things to different people in different circumstances in the same business domain. It's a proper concept to describe the visual screen flow but nothing more.

"Flow" misuse is historical in DC

The misunderstanding of the "flow" is long-standing in DC, and it predates the current CFV design/framework. Initially the "flow" was controlled by JBPM engine. The real problem was not JBPM but that this business workflow engine was used as the visual flow orchestrator.

High Level Analysis of the Problem Domain

DC flows as pseudo workflows

- The “application process for a banking product” is a 1-step business process where a customer supplies some information and the bank either grants the product or not, the information required changes from product to product but that’s not a process.

- An application for a banking product is akin to a purchase. And in fact, the component is called the ‘Digital Cart’ in eBanking. Typical eCommerce apps have one app for all their product purchases, they don’t have a separate apps for different product types like eBanking does for deposit (DFA) and credit card (CCFA) etc.

- The application process as carried out in eBanking in all its various formats such as CCFA/SPCA/DFA etc. simply breaks the customer data requirements into various screens that create a flow. The flow is real in the UI but in business terms it’s illusory.

- The reason the data request is broken up into a flow is to make the application process “seem” easier to the customer. In the past the prevailing UI design was to have many pages each with a small number of data fields (OAO). Now it’s the opposite, to have all data fields in 1 page(CCFA/DFA). In other words, the flow has changed completely but the business functionality stayed the same because **the visual flow is not a real workflow**.

Problem Analysis cont.

DC as API gateway & Compass as the true back-end

- Some say that DC is a **’backends to frontends**’ pattern. This idea is tempting but it has one crucial problem. The main data flow is the opposite, it’s not from back to front but from front to back (client sends in data and gets back an answer).

- In the full system tier analysis, **eBanking is the middleware**. A conduit for communication between the UI and the real back end which is Compass.

- It can also be viewed as a kind of **API gateway** (the opposite of ‘backends to frontends’) because its only purpose is to route the UI API data requests albeit repackaged (for API) to the correct Compass API.

Problem Analysis cont.

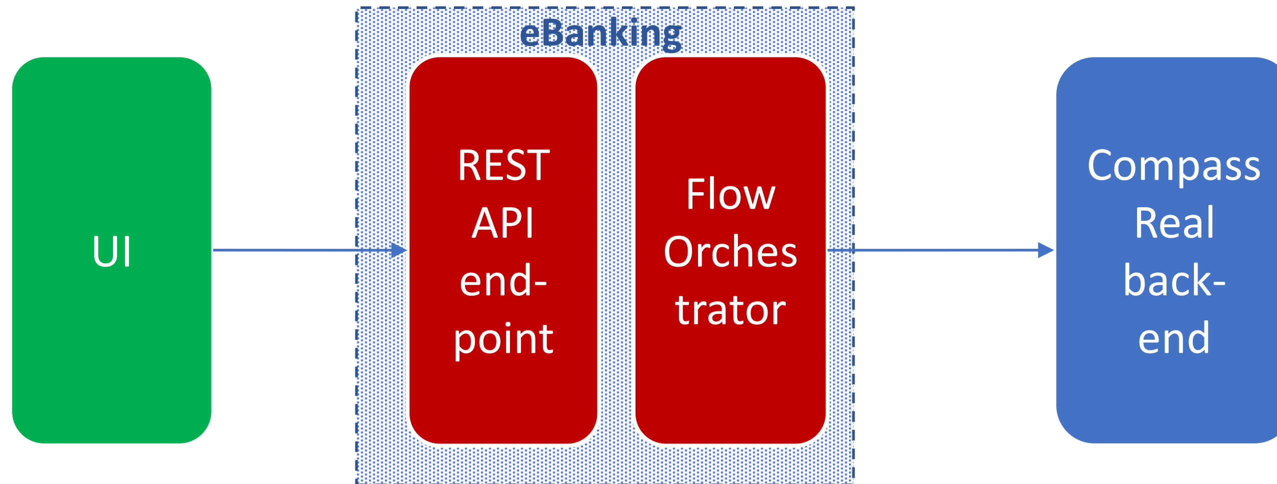
Compass as the business workflow engine & DC as workflow manager for Compass

- ❖ The **real business workflow resides in Compass** as the engine that adjudicates and fulfills product applications.
- ❖ To eBanking Compass’s business functionality is essentially a black box but its API does require a specific “flow” of eBanking calls.
- ❖ Here is the real eBanking workflow vis-à-vis Compass hiding in the DC visual flows:

1. Create Case in Compass.
2. Create/update/complete Enrollment in Compass.
3. Get Recommendation from Compass.
4. Update/Accept Recommendation in Compass.
5. Perform Product Setup in Compass
6. Get Fulfillment Summary from Compass.

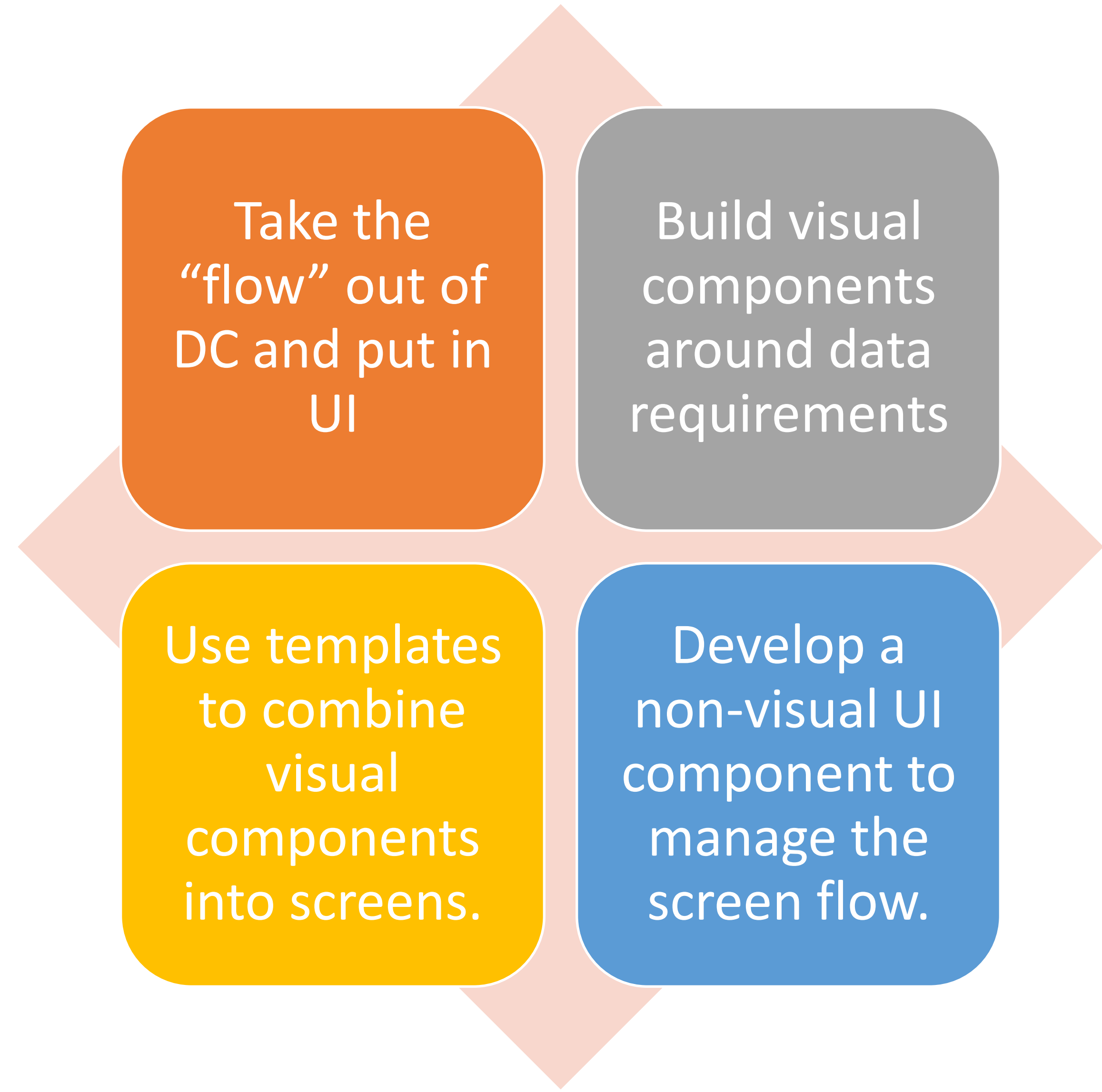
New Approach To eBanking Design

- This diagram illustrates the "split personality" of eBanking that a good design should respect:
  - REST API to UI flow
  - the orchestrator to Compass processing.

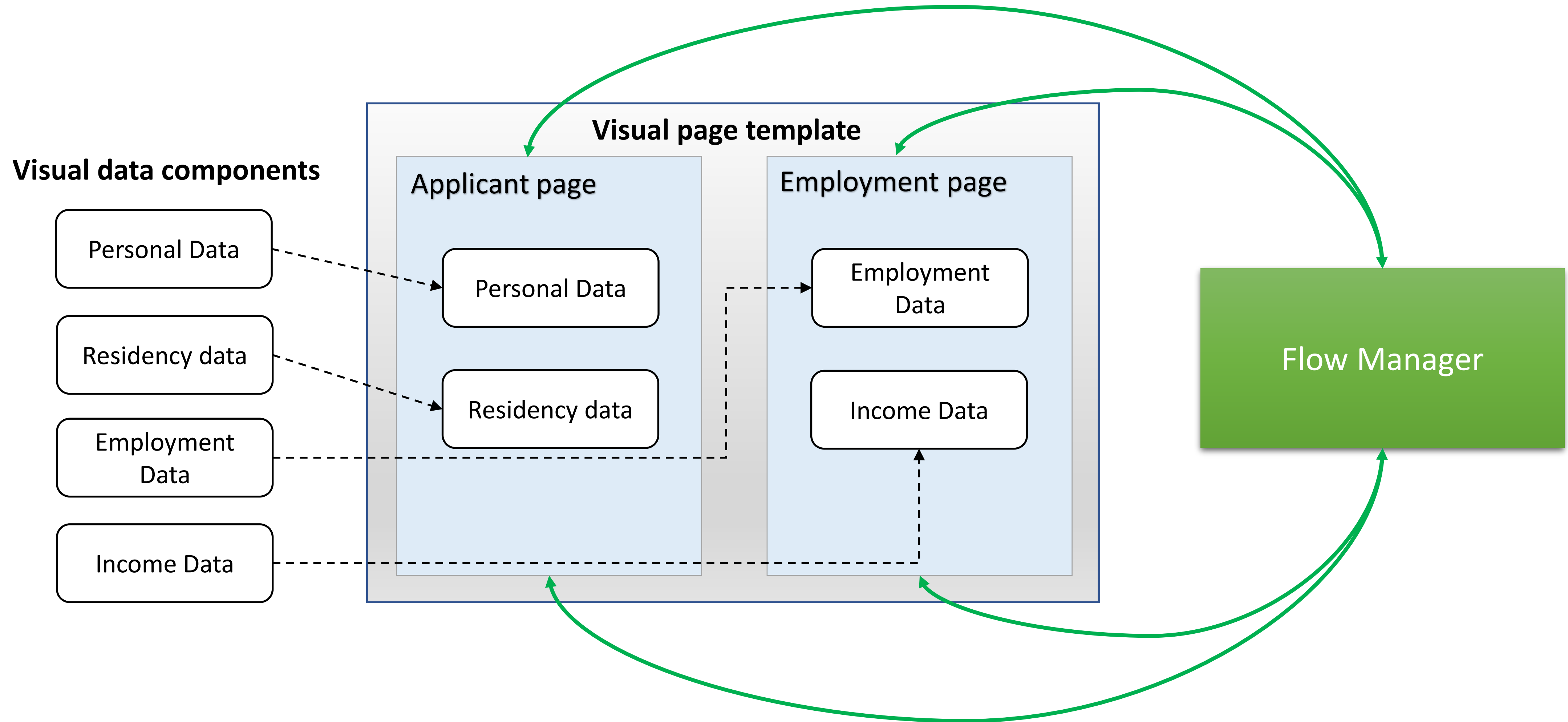


- Other goals and principles that we need to accommodate in this new architecture
  - All layers of software should be independent of each other and only loosely tied together through APIs and/or messaging
  - This will allow the UI to become a full-fledged application or "flow" with as many defined "flows" as needed
  - The remaining layers will become services to the applications and will cease to be self-contained applications

UX Design Incorporating Workflow



An Example of UX Design Approach



eBanking DC Design Approach

Design Patterns

- Use 2 design patters

REST API

Current API
- Carries state => NON-REST
- Single generic resource (/api/applications)
- End-point collections mapped per flow and UI screen

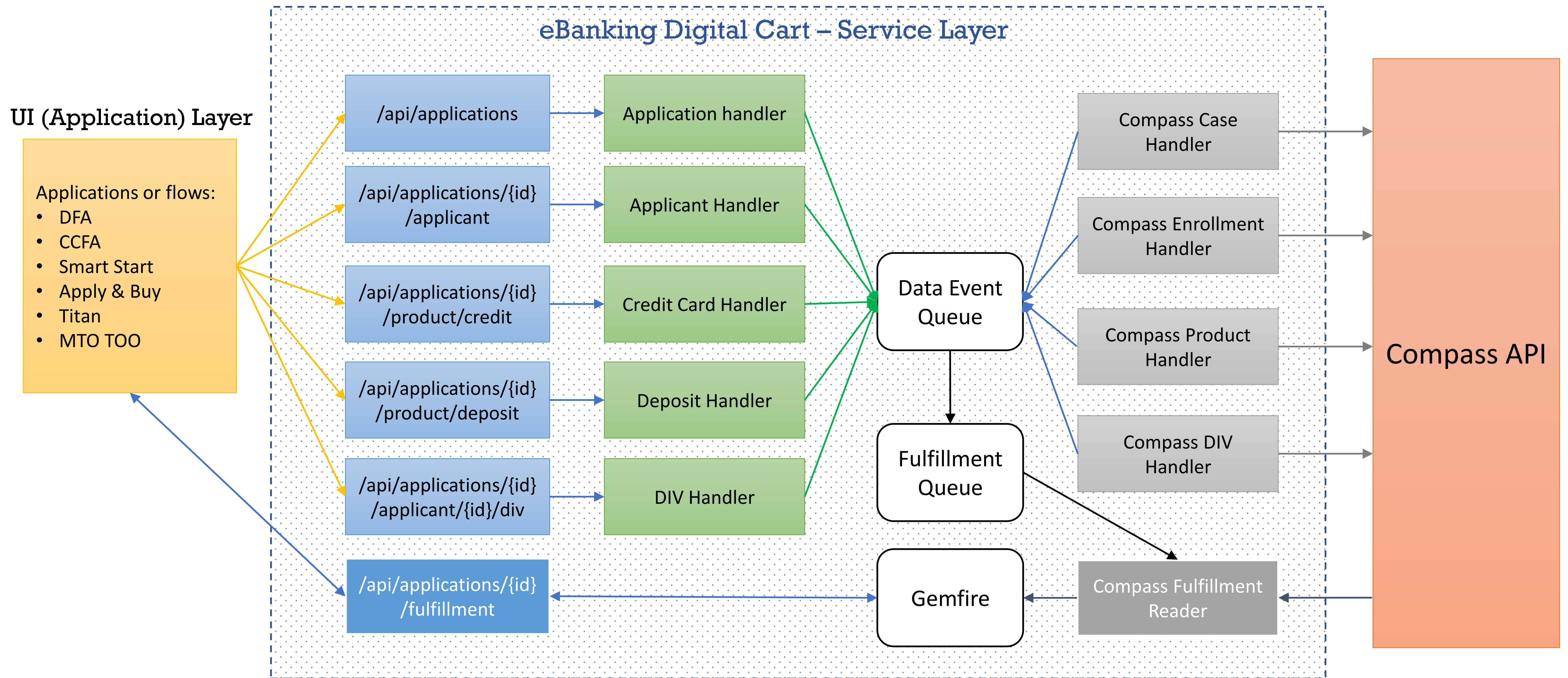
New REST API
- State is held in UI => REST
- True resource end-points such as ‘product’, applicant’, ‘application’ etc.
- Resource granularity can be adjusted based on data requirements not UI flow

Compass Orchestrator

- Avoid the dilemma of 1 generic or many specific orchestrators (1 per flow)
- Use choreography instead aka “silent orchestration”

eBanking DC Design

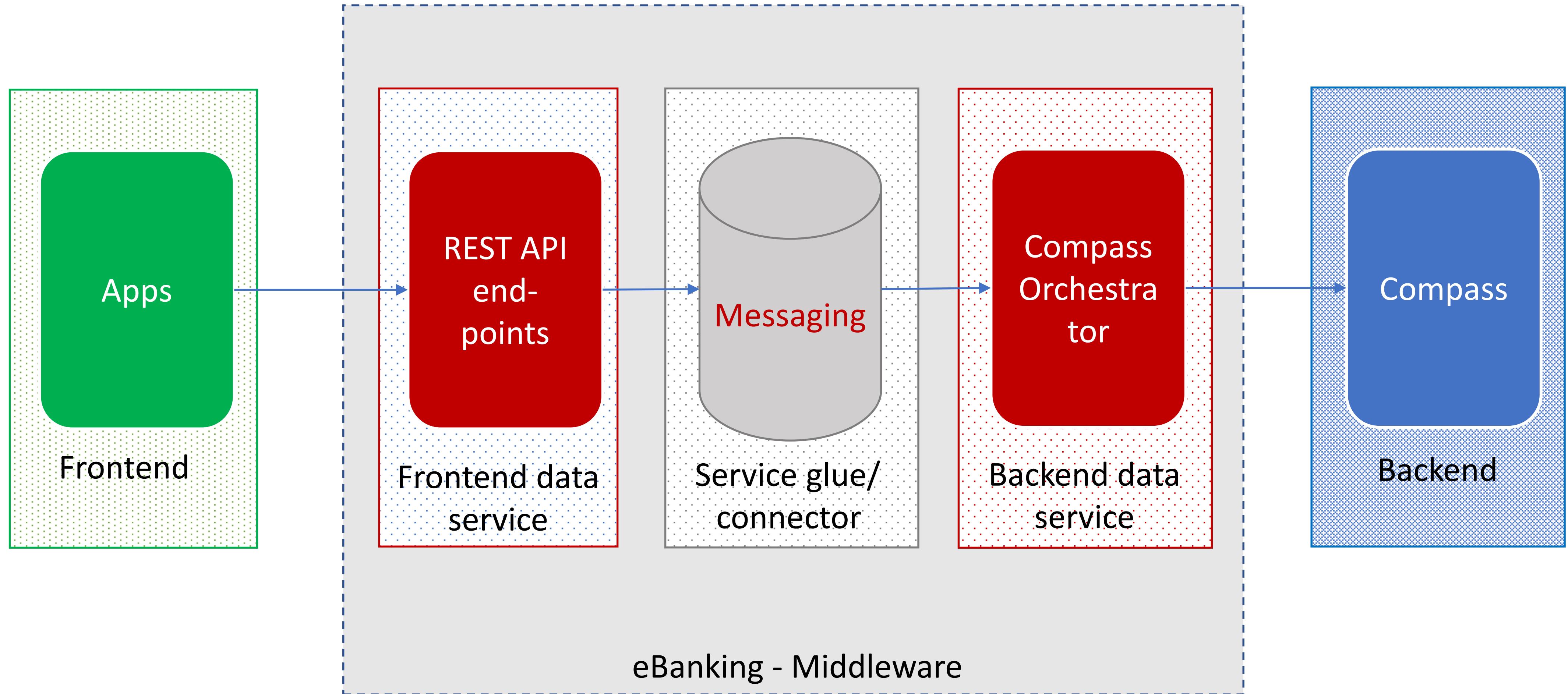
This is just a quick outline and is not meant to be complete



Summary of New Design Characteristics

- UI is no longer a collection of web pages but a collection of full-blown applications
- eBanking DC is no longer an application layer but becomes a service layer
- The full design is very generic and can be applied to almost any type of application because the application resides in UI and is not spread across layers
- The design is consistent with the best modern design practices of reactive architecture:
    - Almost full independence of all layers from each other
    - Service dependencies are dynamic and handled by messaging
    - Each layer deals with only its own concerns and responsibilities
    - Each layer consists of independent components whose dependencies are minimal
    - Each component can be independently designed, developed, maintained and tuned for security and performance
    - Minimal future design constraints
- This design can be extended by exploiting the natural fault line in eBanking by splitting DC into 2 separate services:
    - Frontend data handler
    - Backend data handler
- Possible further extension of the architecture can be done by moving the ‘Frontend data handler’ to the API gateway when it’s available since this DC component is essentially a type of API gateway

Extended New DC Architecture



How to Get There

Proof of Concept

- CCRI (Consolidated Collection Recovery Initiative) intake would be a good candidate for PC
- It's an application for a consolidation loan, a new type of product
- There is no existing flow that can just be copied to a new code base, this would need to be builtnot from scratch but based on DFA or CCFA flow

User Interface

- The main challenge and by far the most complex change would be in the UI
- Current UI will need to be transformed from a collection of dumb screens to a collection of applications
- The screens are all there, but the flow manager needs to be built for each application
- Also, the data needs to be passed from screen to screen when DC is no longer doing that

Digital Card

- Collections of UI APIs need to be converted into one collection of true REST APIs
- The current DC code would need significant rewiring to turn it into UI and Compass API handlers from the current VSM framework
- The most challenging thing will be the messaging as there is nothing in eBanking currently. With Azure migration, likely the best option would be to use one of the Azure messaging services.