

BYOD (Bring your own Data)

Business Motivation

Our core challenge lies in enabling customers to integrate their own data with our system's foundational data to create a more seamless experience while unlocking meaningful insights.

While our **core data** is relatively stable and constrained—something we can learn, structure, and optimise over time—**client data** is inherently diverse, dynamic, and unpredictable. Attempting to create a universally “stable core” that accommodates all variations of client data is both impractical and counterproductive. Instead, our focus is on **data composition**, ensuring that customers can **easily bring their own data (BYOD)** while making it intuitive to define and link it to core system entities.

The reality is that many customers **have data, but it's fragmented across different sources**, often residing in general-purpose tools like Excel, with no single, well-defined structure. This results in a **lack of clarity, duplication, and inefficiencies** when attempting to leverage their own data effectively. To address this, we aim to simplify the process by:

- **Allowing customers to seed their data models from existing examples**, so they can start with what they have and refine it over time.
- **Providing a guided experience** to define attributes, types, and constraints—removing the need for deep technical knowledge.
- **Automating taxonomy, term, and acronym extraction** to surface structure from unstructured data, helping users organise their data from day one.

By making the **import, definition, and linking process frictionless**, we empower customers to take control of their data without requiring specialised knowledge. This ensures that **data composition**—rather than rigid standardisation—becomes the foundation for a scalable and flexible integration model.

Inspiration

- [Graph.Build](#)
- [YouTube Video](#)

Pseudo Story Map

- [BYOD - Miro Board](#)

Existing Work / Progress

- [Figma: Levon and Jessie - Media Magic](#)
- [Ryan's PoC Loom Video 1](#)
- [Ryan's PoC Loom Video 2](#)

User Experience Motivation (Process)

Guided Model Definition Wizard

Breaking the setup into a step-by-step **wizard** helps non-experts tackle one task at a time. A well-designed wizard UI presents each stage (model name, attributes, etc.) separately with clear “Next” and “Back” controls ([Wizard UI Pattern: When to Use It and How to Get It Right](#)). This **guided progression** prevents overwhelm by segmenting a complex process into manageable steps ([Wizard UI Pattern: When to Use It and How to Get It Right](#)). Include a progress indicator (e.g. step numbers or a timeline) so users know where they are in the flow ([Wizard UI Pattern: When to Use It and How to Get It Right](#)). Each step should have a concise heading (e.g. “*Define Your Data Model*”) and brief instructions. For instance, when asking for the model definition, prompt with friendly language: “*What kind of item are you modeling? (e.g. Product, Event, Customer)*”. Avoid technical RDF jargon – use familiar terms like “*Type*” or “*Category*” instead of “*Class*” or “*Ontology*”. Keeping the tone approachable ensures **power users who lack data modeling expertise** aren’t intimidated. Tooltips or info icons can offer additional explanation for concepts like “constraints” or “data type” without cluttering the main interface (a form of *progressive disclosure* to reveal details only if needed ([How to Simplify Complex Interfaces in UX](#))). Overall, a wizard with clear labels and helper text will gently onboard users through model setup, aligning with UX best practices for multi-step workflows ([Wizard UI Pattern: When to Use It and How to Get It Right](#)).

Easy Attribute Definition with Types & Constraints

After the model name is set, guide the user to **define attributes (fields)** in a simple, form-like step. An intuitive pattern is to list attributes in a table or list where each row is a field with an editable name, type dropdown, and options for constraints. **Show, don't tell:** if the user has provided a dataset (BYOD), display a few sample values for each field to give context. For example, in a CSV import interface, showing a snippet of data under each column name helps users understand what that column contains ([design - designing CSV column mapping interface - User Experience Stack Exchange](#)). One UX solution is to present sample data in dropdowns or beneath labels so users can infer a field's purpose even if the source headers are cryptic ([design - designing CSV column mapping interface - User Experience Stack Exchange](#)). This way, a field originally named "lname" in their file might show example values ("Smith, Garcia, Lee"), cueing the user that it represents a last name.

When selecting **data types** (text, number, date, etc.), use plain language and icons (📅 for date, 📊 for number) to make choices obvious. Auto-detection can simplify this step: if the system detects a column looks like dates or integers, pre-select that data type (users can override if wrong). For **constraints**, stick to familiar concepts like **Required** (must have a value) or **Unique** (no duplicates), instead of abstract terms. Keep advanced settings (like RDF URI templates or relationship cardinality) hidden under an "Advanced Options" toggle to avoid confusing the user ([How to Simplify Complex Interfaces in UX](#))⁹. The UI might, for instance, have a checkbox for "This field is required" rather than requiring the user to understand cardinality of 1..1. By progressively disclosing advanced constraints only when needed, the interface stays clean and focused on basic input ([How to Simplify Complex Interfaces in UX](#))⁹.



It's also helpful to incorporate validation cues early. If a user marks a field as numeric, you can immediately flag any sample value that isn't a number. Gentle validation messages (in-line, next to the field) guide users to catch mistakes in data type selection. All these patterns – showing examples, suggesting types, and simplifying constraint settings – contribute to an **intuitive attribute definition** step that demystifies data modeling for non-specialists.

Visual Data Mapping Interface

Once the schema (model and fields) is defined, the **"bring your own data" mapping** comes to life through a visual mapping interface. This typically shows the user's source data fields on one side and the new schema's fields on the other, allowing the user to

connect or confirm mappings. A proven UX pattern is to present a side-by-side or over/under list of fields with controls to map them, accompanied by sample data. The interface might automatically attempt to **auto-match columns to fields** by name similarity or data type (e.g. mapping “First Name” in the file to the **First Name** attribute). In a case study for a data mapping module, the team noted the system should be smart enough to auto-map by similar names and types, then clearly highlight any unmapped or ambiguous fields for the user’s attention ([Case Study: RxBenefits Data Mapping Module](#)⁴). This reduces manual work and uses visual cues to focus the user on what needs input. ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁵ *Example of a data mapping UI:* The importer has auto-matched three CSV columns (A, B, C) to fields (Task name, Description, Assignee) and shows that all values pass validation for those fields. The user can **Confirm mapping** for each or choose to ignore a column. (Image source: a Flatfile-powered importer in ClickU ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#)) ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁰)

As shown above, a good mapping UI will list each source field, indicate the target it’s mapped to, and provide actions: e.g. **Confirm mapping**, **Ignore this field**, or even **Create new field** if the data has an extra column not in the mod ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#)) ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁶). Using **color-coding** and visual emphasis speeds up user decisions. For instance, using a standout color (green) for the “Confirm” button signals it as the recommended action ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#)) ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁶). In the example, all correctly matched fields have a green *Confirm mapping* button, encouraging users to quickly approve them. Secondary options like “Ignore” can be greyed or less prominent, since they are fallback choice ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁴). This color consistency becomes a visual language: once the user has seen one green confirmation, they intuitively recognize all green buttons as the affirmative action moving forward ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁷).

Another key element is displaying **data validation feedback** during mapping. The interface should show if each field’s values meet the expected format or constraints. For example, below each mapped field, you might show a summary like “100% of rows have a value for this field – all good!” or “Some values are missing or invalid.” This immediate feedback (often with icons like  or ) helps users catch issues early. In one design, the mapping screen showed checkmarks for fields where “all values pass validation” and warning symbols for those that need attention ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#)) ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁸). Providing a preview of a few data

rows under each field (as in the image above) also reassures users that the mapping is correct. These visual cues and controls create a **visual data mapping** experience that is both powerful (allowing custom mapping) and user-friendly for non-experts.

Taxonomy Linking and Lookup Values

Linking schema fields to existing taxonomies (controlled vocabularies or lookup lists) can be one of the more abstract concepts, so the UI must make it feel natural. One approach is to treat taxonomy-linked fields similarly to dropdown selection or tagging interfaces that users are already familiar with. For instance, if a field “Category” should link to an existing taxonomy of categories, the field settings can offer an option like **“Choose allowed values from...”** and present a list or tree of the taxonomy terms. The user experience should be akin to selecting from a list of predefined options, rather than thinking about “linking an ontology”. In fact, many content management systems do this: WordPress’s Advanced Custom Fields (ACF) plugin, for example, replaces the default category selector with a more **user-friendly, visually appealing interface** (checkboxes, radio buttons, multi-select) for choosing taxonomy terms ([ACF Taxonomy Field: Complete Guide with Code Snippets](#))⁴ . This demonstrates how offering familiar selection controls makes linking to a taxonomy feel like a regular form input, simplifying a complex data relationship into a simple choice.

When the user maps their own data, taxonomy linking often involves **reconciling incoming values with the approved list**. A good UI will flag values that don’t match the existing taxonomy and guide the user to resolve them. During data import, this can be part of a **“Repair problems”** step in the wizard. For example, an app might import a “Priority” column but require the values to match a set of priority levels (High, Medium, Low) that exist in the system. If the user’s data has numeric or differently-named priorities, the interface can highlight those as issues and let the user map them to the correct taxonomy terms. ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁵ *Handling taxonomy mapping during import:* The “Priority” field from the CSV is linked to the system’s Priority taxonomy. The importer flags the incoming values 1, 2, 3 with warning icons (orange ⚠️) because they don’t match the expected Priority values. The user can resolve this by choosing corresponding allowed values from drop-downs (“Our values”). This way, all data is aligned to the existing taxonomy before import. (Image source: Flatfile importer ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))²)

In the example above, the UI clearly indicates the mismatch: *“100% of your rows fail validation (repair on next step)”* and shows an orange alert icon ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁸ . It then provides an interface for the user to **map each unrecognized value to an existing term** (the “Choose one” dropdowns under “Our values”). This visual mapping of values to

taxonomy terms is critical for an intuitive experience. Rather than dumping an error, the system engages the user in an interactive correction, with readable labels (“*Your values*” vs “*Our values*”). Once the user selects how, say, “1” maps to “High Priority,” the warning can turn into a confirmation.

For simpler cases, if a field simply needs to link to a reference list (like Country codes or Department names), using an auto-complete dropdown is an intuitive pattern. As the user types, suggest matching terms from the taxonomy, preventing mistakes and educating them on valid entries. If multiple selection is allowed (e.g. tagging multiple topics), use familiar token-style inputs or checkbox lists. Consistency and clarity are vital: if the field is taxonomy-linked, the UI should make it clear that the user must pick from existing values (e.g. by labeling it “Choose an existing [Term]” rather than a free-text input). By leveraging common form patterns (dropdowns, checkboxes, autocompletes) and providing interactive resolution for mismatched data, the design makes **taxonomy linking** straightforward for power users who don’t need to know the complexities under the hood.

Simplifying Complex Data Concepts in the UI

Designing for non-expert power users means abstracting away complexity and surfacing only what’s necessary for decision-making. Several **best practices** can ensure the UI remains approachable:

- **Use Familiar Language:** Rename technical terms to domain language or common phrasing. For example, instead of “OWL Class” use “Type of item”, instead of “xsd:string” just say “Text”. Clear, non-technical labels prevent confusion and reduce cognitive load ([How to Simplify Complex Interfaces in UX](#)) ([How to Simplify Complex Interfaces in UX](#))¹ . If RDF-specific notions are involved (like URI prefixes or ontology references), consider whether the user even needs to see them. Often these can be auto-managed or tucked into advanced settings with explanatory text.
- **Progressive Disclosure:** As mentioned earlier, show basic options first and hide advanced settings until the user opts ([How to Simplify Complex Interfaces in UX](#))⁹ . A simple toggle for “Advanced options” can conceal fields like “Data format constraints” or “URI patterns” which typical power users won’t initially understand. This keeps the main interface clean and focused on the core tasks, while still allowing expert tweaks if necessary.
- **Visual Hierarchy & Clarity:** Arrange the interface to mirror the user’s mental model of the task. Group related inputs (e.g. group all attribute fields together under an “Attributes” section) and use panels or step indicators for each major phase (Definition, Mapping, Review). A clear visual hierarchy with distinct sections and headings helps users navigate complex forms more easily ([How to Simplify Complex Interfaces in UX](#))

([How to Simplify Complex Interfaces in UX](#))⁹]. Also, maintain consistency in how interactive elements look and behave. For instance, all dropdowns should function similarly, all required fields might be marked consistently, etc., so users can predict the interactions without re-learning at each step ([How to Simplify Complex Interfaces in UX](#))⁹].

- **Contextual Help and Examples:** Strategically place helper text or examples near complex concepts. A short description under “Data Type” explaining “e.g. Text = letters and symbols, Number = only digits” can be extremely helpful. Likewise, when linking to a taxonomy, a sentence like “Choose from existing categories (e.g. Science, Arts, Sports...)” or a link to “View list of allowed values” can clarify the task. These bits of microcopy act as the user’s guide within the UI, ensuring they understand the implications of their choices.
- **Automate and Suggest Defaults:** Wherever possible, let the system shoulder the complexity by auto-suggesting sensible defaults. For example, **auto-detecting field properties** (type, constraints) or pre-filling fields based on the data can simplify the user’s job. The RxBenefits data-mapping project emphasized the need for the system to map what it can and call out only the mismatches for user attention ([Case Study: RxBenefits Data Mapping Module](#))⁴]. This principle can extend to model design too – if the user uploads data first, the app could propose a model outline (with field names/types) which the user simply verifies or tweaks, rather than creating everything from scratch.
- **Feedback and Error Handling:** Ensure that at each step, the user gets immediate feedback on their input. If they try to proceed without defining any attributes, for instance, an inline prompt could say “You need to add at least one attribute to continue.” Clearly highlight errors in a friendly manner, and whenever possible, suggest a fix. For example, if an attribute name is left blank, instead of a generic error, the message might be “Attribute name can’t be empty – e.g. ‘Title’ or ‘Description’.” By guiding the user to a solution, you prevent frustration and keep the journey smooth.

Lastly, learn from **real-world products** that tackle complex data tasks with elegant UX. Tools like **Flatfile Concierge** (integrated in platforms like ClickUp) show how a typically technical process (data import and field mapping) can be made visual, with a spreadsheet preview and step-by-step validation ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#)) ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁰]. No-code knowledge graph builders like **Graph.Build** or **KgBase** provide drag-and-drop modeling and hide the semantic technicalities behind intuitive actions (e.g. dragging a connection between two nodes creates a relationship without the user needing to write a triple) ([Creating a Knowledge Graph \(or any Graph\) Model without Coding](#)) ([Creating a Knowledge Graph \(or any Graph\) Model without Coding](#))⁸]. Also, content taxonomy tools demonstrate how to constrain inputs to authorized values without confusing the user ([ACF Taxonomy Field: Complete Guide with Code Snippets](#))⁴]. By studying these patterns and incorporating their best ideas – guided wizards, previews of data, confirm/ignore actions, auto-matching, and friendly term selection – you can **bring the BYOD schema design to life**

in a way that empowers power users. The end goal is a UI that **simplifies complex data concepts** into an intuitive journey: from defining a model to mapping their data and linking it to rich taxonomies, all without ever feeling “too technical” to the user.

Sources: Practical examples and patterns were drawn from real implementations like UX StackExchange discussions on CSV mapping interface ([design - designing CSV column mapping interface - User Experience Stack Exchange](#)) ([design - designing CSV column mapping interface - User Experience Stack Exchange](#))⁶ , case studies of data import too ([Case Study: RxBenefits Data Mapping Module](#)) ([Case Study: RxBenefits Data Mapping Module](#))⁸ , and design guides (Nielsen Norman Group, Smashing Magazine) on simplifying data-heavy workflow ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#)) ([Designing An Attractive And Usable Data Importer For Your App — Smashing Magazine](#))⁸ . These illustrate successful approaches to guided setup, visual mapping, and taxonomy integration in modern, user-friendly applications.

Architecture Motivation

Benefits of Using RDF as a Data Model: Focus on Compositionality and Integration

1. Compositionality: Building Flexible, Extensible Data Models

RDF’s graph-based nature allows data to be **built up incrementally**, making it a perfect fit for compositional data modelling. Instead of imposing a rigid schema upfront, RDF enables:

- **Incremental Growth** – New data elements, attributes, and relationships can be added **without disrupting** existing structures. This allows users to start with a minimal model and refine it over time.
- **Modular Data Composition** – Different datasets (internal, client-provided, third-party) can be merged seamlessly, forming a **unified but flexible** knowledge graph.
- **No Fixed Schema** – Unlike relational databases, RDF doesn’t require strict table definitions, making it adaptable to diverse and evolving data sources.
- **Reusability & Sharing** – Common data structures (e.g., taxonomies, vocabularies) can be **reused across multiple domains** rather than reinvented, reducing redundancy.

2. Seamless Integration Across Heterogeneous Data Sources

Many organisations struggle with integrating **disparate** datasets due to varying formats, structures, and evolving business needs. RDF is designed to handle this challenge by:

- **Standardised Data Interoperability** – RDF is based on open standards (W3C), ensuring compatibility across different data ecosystems, APIs, and knowledge graphs.
- **Cross-Domain Linking** – Unlike traditional databases that struggle with connecting datasets, RDF's subject-predicate-object triples **natively support relationships across multiple domains**.
- **Unified Data Access Layer** – RDF enables querying across structured (SQL, enterprise databases) and semi-structured/unstructured data (CSV, JSON, XML, Excel) without requiring rigid transformations.
- **Ontology & Taxonomy Integration** – RDF supports linking data to well-defined **industry ontologies and taxonomies**, allowing structured reasoning, classification, and automated lookups.

3. Flexible and Scalable Data Composition

- **Federated Querying** – SPARQL (RDF's query language) allows querying across multiple distributed datasets **without the need for complex data pipelines**.
- **Graph-Based Representation** – Unlike hierarchical or table-based storage, RDF's graph model allows **dynamic relationships** between entities, making it easier to **merge, enrich, and extend** data.
- **Future-Proofing** – RDF allows the **progressive addition of new entities and relationships**, ensuring that today's data models don't become tomorrow's technical debt.

4. Semantic Understanding & AI-Driven Insights

By structuring data as a knowledge graph, RDF enables:

- **Context-Aware Data Relationships** – Entities aren't just connected by IDs but **semantically described**, making data more meaningful and machine-readable.
- **Automated Reasoning** – Inference engines can deduce new relationships from existing ones, unlocking **hidden insights** without manual data engineering.
- **Data Enrichment** – RDF allows **enriching core data** with external linked data sources (e.g., Wikidata, industry standards), increasing the depth of analysis.

Why This Matters for BYOD & Data Composition

For users bringing their own data, RDF ensures that:

They don't need to **conform to a rigid schema** before starting.

Their data can **evolve over time**, linking to **existing models and taxonomies** effortlessly.

It's easy to **merge diverse data sources** into a single, navigable structure.

AI-driven automation can **suggest and enrich models** based on their data patterns.

By leveraging RDF, we shift from **data standardisation** to **data composition**, ensuring integration is a smooth, scalable, and future-proof process.