EIS GROUP™

Design Document

Author	Chris Co
Status	Draft
Document Number	
Document Version	00.01.00
Last Revision Date	August 19, 2020

Revision History

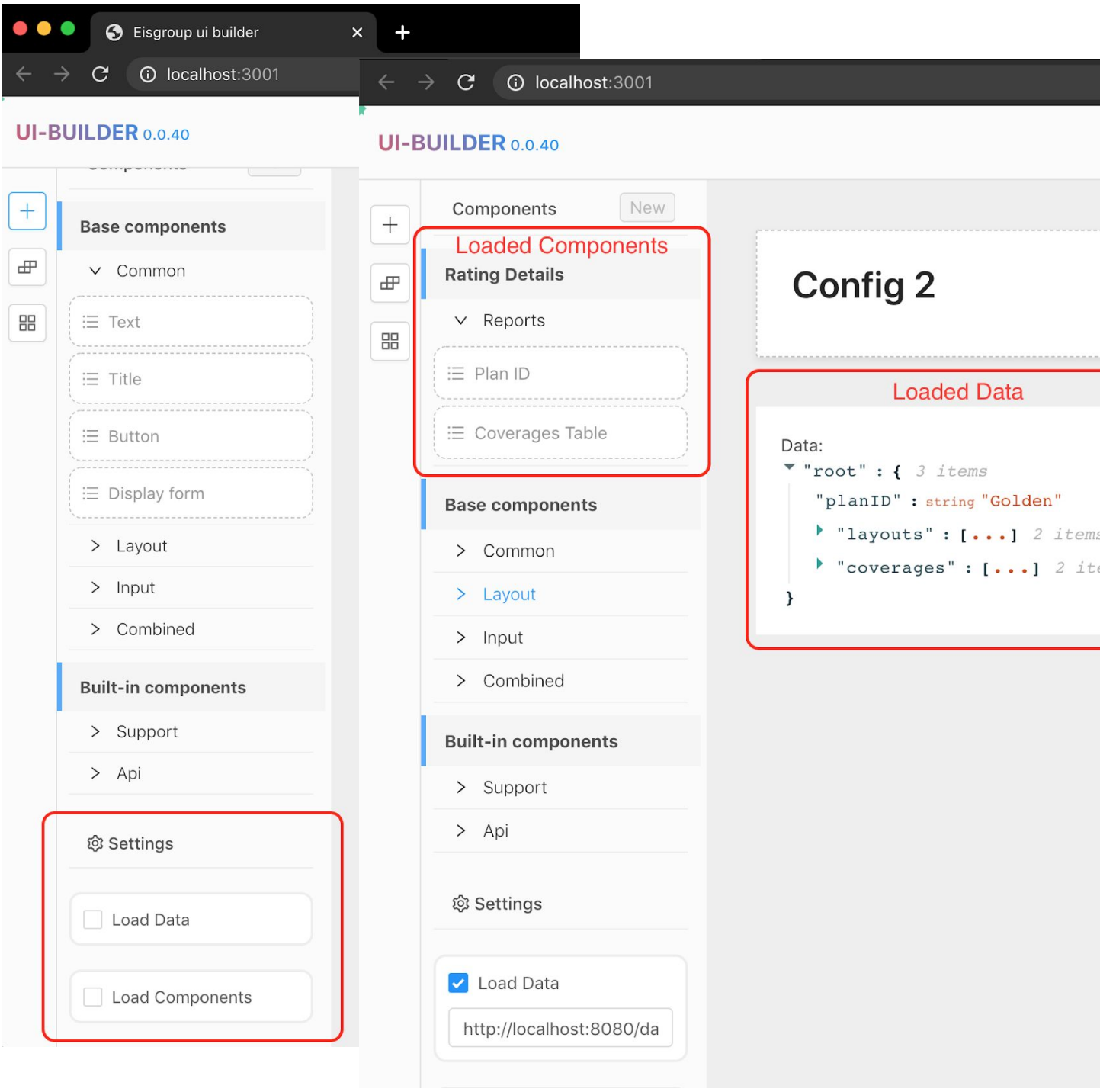
Name	Date	Reason For Changes	Version
Chris Co	July 1, 2020	Initial Version	00.01.00

Copyright

1 Introduction

1.1 Overview

The Dynamic Component loader (labeled as **Load Components** in the UI) is a feature within the UI Builder that allows the user to define an URL, which will be used to fetch JSON data from an external API. This JSON data contains `**builderConfig**`, that loads predefined components on the left menu of the UI Builder Editor. See illustration below (POC demo video is available):



1.2 How It Works

- Dynamic Component's URL is fetched on the fly (in runtime) to populate the UI Builder with a dynamic list of components generated externally.
- Dynamic Components will update when the URL changes in the UI Builder.
- Dynamic Component's behavior can be updated externally by its creator with a versioning system, without requiring update from existing integrations.
- Dynamic Components may use Loaded Data, which are also fetched from external API.
- If the Working Canvas uses a Dynamic Component, but the Component is missing (i.e. not loaded due to URL change), that Component is ignored, without throwing error.
- If the Working Canvas uses a Dynamic Component with Loaded Data, but the Data is missing (i.e. not loaded due to URL change), the Component must be able to render without Data, and not throw error.
- Users can customise a Dynamic Component like any other Build-in Components.

1.3 Glossary

Dynamic Component - a UI Component loaded from external API that may use Loaded Data.

Loaded Data - JSON data fetched from external API, made available for use within UI Builder.

Working Canvas - an area within the UI Builder for dragging and dropping components into.

Other **UI *** terms** - please see UI Builder Glossary.

2 Why?

2.1 What Problem It Solves?

Rating Details reports require **complex components**, which currently, the UI Builder is incapable of building, due to its architectural limitations.

Examples of use cases include, but not limited to:

- Nested Tables within Tables (many levels deep)
- PieChart or any Custom Component to be nested within any Table Cell, Row, Header…
- Different data formatting beyond simple strings (i.e. localised Currency/Percent/etc.)
- Dynamic Layout that depend on UI State set by the user in runtime (i.e. Dropdown)
- Custom callback logic that depends on runtime state and external data
- Encapsulation of complex data mapping from the end user (i.e. clients)
- Any combination of all above, which cannot be predefined by the developer because it's an **Infinite number of possibilities**.

The **UI Renderer** created by the OpenL team has solved all problems above. But it cannot be integrated to the UI Builder without the Dynamic Component loader.

2.2 Why It Cannot Be Static?

As explained above, the Dynamic Component cannot be created manually by the developer to cater to all possible configuration variations at *pre-compile time*. Because the moment a Component is predefined, it no longer allows the **freedom to customise** and **compose** in creative ways.

2.3 How It Benefits Business?

- Production cost (R&D) can be reduced by at least **x3 times**, because new UI Components no longer require involvement of a software programmer.
- It has been proven by the OpenL team that Business Analytics **people can create** Components **themselves** within hours of reading the UI Renderer docs.
- Business Analytics people understand business much better than developers, and they can create new UI Components customised to their needs much faster, and **more efficiently**, than software engineers in the long run.
- **Faster delivery** cycle - business is not blocked by developers for major UI changes.
- **Unlimited** configuration possibility, with "*Learn Once - Use Everywhere*" approach.

2.4 How It Benefits Developers?

- **Separation of concerns** - documentation, communication, and misunderstanding between different stakeholders is *reduced to almost zero*, because developers no longer need to understand business logic - they just create generic UI Components that are customised by the business.
- **Reduced** new hire **training cost** and **on-boarding time** - EIS can hire any React developer and get them up to speed without insurance policy knowledge.

2.5 How It Benefits Clients?

- **Reduced Support Cost**
- **Faster iteration**
- **Appealing offer** - Unlimited Configuration + DIY without a programmer.

3 Design

3.1 API

1. The Dynamic Component URL shown in the demo is a single string, but it can be a list of URLs, which will load Components from **multiple external APIs**.

2. Each page/scene in the app can have a different Dynamic Component URL/s - this allows users to **combine components** from different teams.

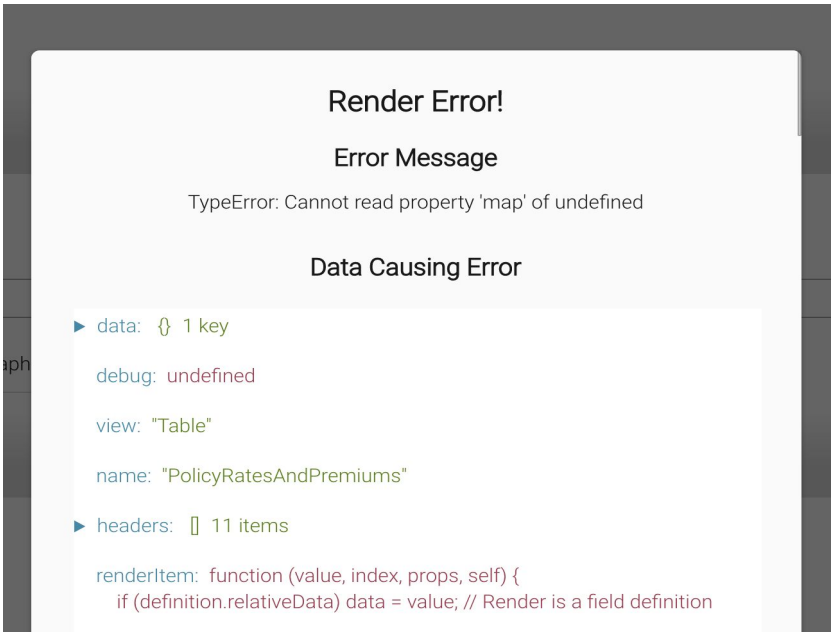
3. **Versioning** can be used to prevent breaking changes in production. Semantic versioning should be used, similar to how *npm* works.

4. Creators of Dynamic Components are responsible for regression testing.

5. The UI Builder can have different versions of the same Dynamic Component for different pages/scenes, but only one version can be active per UI Engine instance.

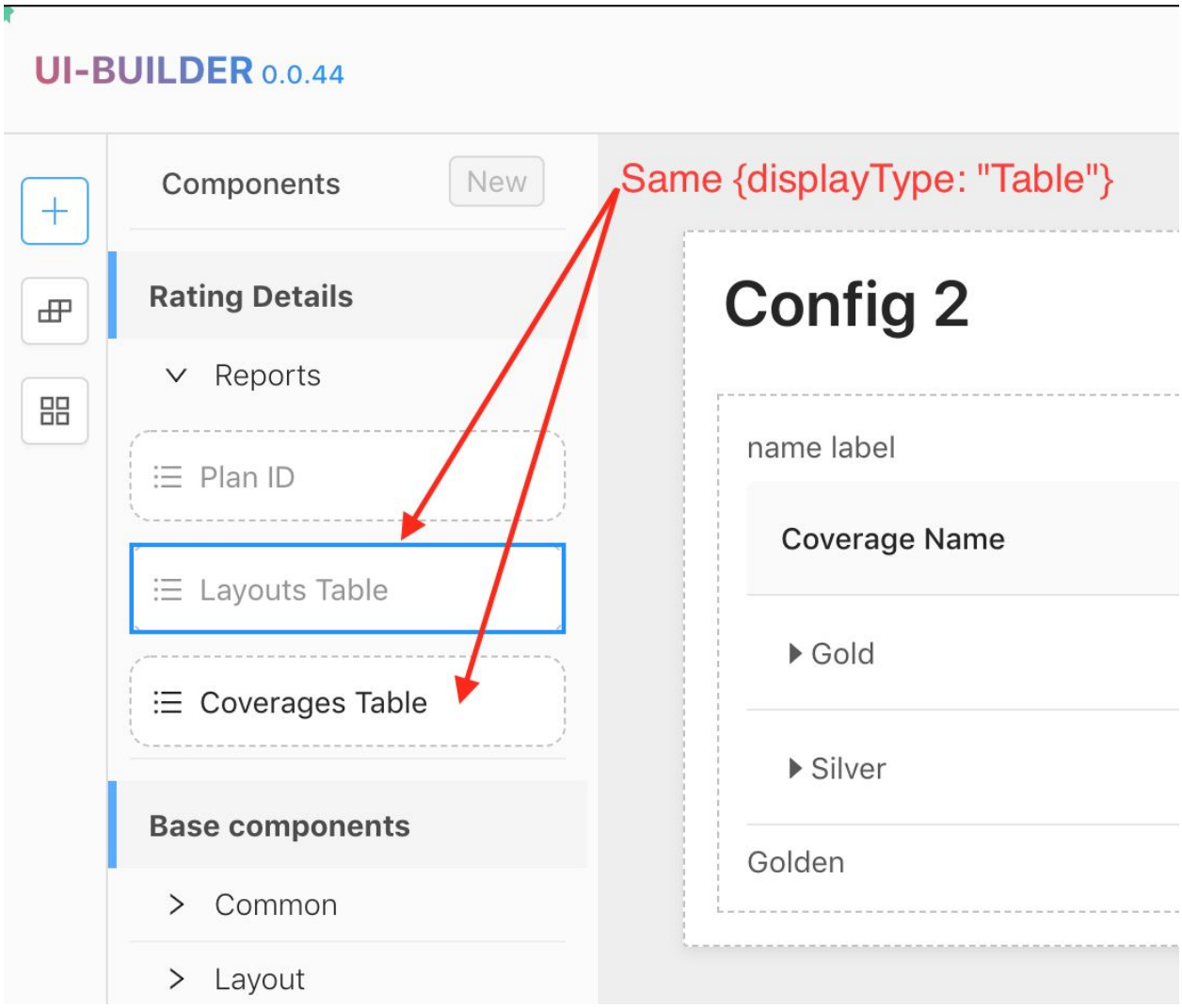
6. **Lazy load** Components via a JSON response callback option provided by the UI Builder.

7. **Errors** caught within Dynamic Components can be suppressed in production mode, or shown as warning messages in debug mode using React `**componentDidCatch**`:



3.2 Requirements

1. The UI Builder must be able to **update `builderConfig`** at runtime.
2. The UI Engine must be able to support all UI Renderer **transform patterns**.
3. The UI Builder needs to support **multiple instances** of the same Component Type.
   See illustration below:



3.3 Example

Dynamic Component JSON Response:

```
{
    name: 'Rating Details',
    elementConfigs: [
      {
        view: 'Title',
        displayName: 'Plan ID',
        name: 'planID',
      },
      {
        view: 'Table',
        displayName: 'Layouts Table',
        name: 'layouts'
      },
      {
        view: 'Table',
        displayName: 'Coverages Table',
        name: 'coverages',
        headers: [  // -> must be defined if data contains nested tables
          {
            id: 'coverageID',
            label: 'Coverage Name',
            // Custom render function for Table Cells (columns in default layout)
            renderCell: {
              view: 'Expand',
              name: '{value}',
              index: '{index}',
              // toggle extra table item expansion (row in default layout)
              onClick: 'handleItemExpand',
              // items: [{}], // can also make it expand any content inside clicked cell
            },
          },
          {
            id: 'fundingStructure.contributionType',
            label: 'Contribution Type'
          },
        ],
        // Each Expand Item
        renderItem: {
          view: 'Col',
          relativeData: true,
          items: [
            {
              view: 'Col',
              styles: 'margin-bottom-small radius',
              items: [
```