Документация: Динамический UI-Генератор

Назначение

Этот набор классов предназначен для **динамической генерации пользовательского интерфейса** на основе **JSON-схемы**, получаемой, например, с сервера. Он позволяет создавать интерфейсы с **вкладками**, **параметрами** (readonly, number, flags, slider, select, radio), **командами** (кнопками) и **графиками**, обеспечивая **гибкость**, **масштабируемость**, **безопасность** (предотвращение утечек памяти) и **модульность**.

Архитектура

Система построена на принципах **компоновки**, **наследования** и **публикации/подписки** (Event Bus).

Основные компоненты:

- **`UIGenerator`**: Главный класс. Принимает `Идентификатор в виде строки`, `stateSchema` и `elementHandlers`. Отвечает за инициализацию всех подсистем, управление видимостью контроллера и запуск процесса генерации UI.
- **`ElementFactory`**: Создает экземпляры конкретных классов элементов UI (`TabsContainerElement`, `GraphicsContainerElement`, `TabElement`, `ParameterElement`, `CommandElement`, `ChartRenderer`) на основе данных из `stateSchema`. Обеспечивает слабую связанность `UIGenerator` с конкретными реализациями элементов.
- **`UIElement` (и потомки)**: Базовый класс для всех элементов UI *верхнего уровня* (`TabsContainerElement`, `GraphicsContainerElement`). Определяет общие свойства (id, visibility) и методы (render, update, destroy). Потомки реализуют специфичную логику.
- **`TabElement`, `GroupElement`, `ParameterElement`, `CommandElement`**: Классы для элементов вкладок.
- **`ChartRenderer`**: Класс для отрисовки *одного* графика с несколькими линиями (`lines`).
- **`UIStateManager`**: Управляет состоянием UI. Регистрирует все сгенерированные элементы и отвечает за их обновление (например, изменение значения параметра или видимости).
- **`EventManager`**: Централизованная "шина событий". Элементы UI публикуют события (например, `PARAMETER_VALUE_CHANGED`, `COMMAND_CLICKED`, `ELEMENT_VISIBILITY_CHANGED`). `UIStateManager`, `UIGenerator`, `DataConnector` и другие компоненты подписываются на эти события и реагируют на них.
- **`DataConnector`**: Отвечает за взаимодействие с сервером (например, через WebSocket). Подписывается на события `EventManager`, отправляет их на сервер и публикует полученные от сервера обновления обратно в `EventManager`.
- **`Logger`**: Управляет логированием с различными уровнями (`error`, `warn`, `info`, `debug`). Позволяет отслеживать работу системы и находить ошибки.
- **`SchemaValidator`**: Проверяет корректность `stateSchema` перед её обработкой. Повышает надежность системы.

Структура схемы

Схема — это объект JSON, описывающий структуру UI.

```
{
  "schema_version": "string", // (Опционально) Версия схемы
  "controller": {
    "name": "string",              // Уникальный идентификатор контроллера
    "display_name": "string",      // Отображаемое имя контроллера
    "layout": "row" или "column", // Ориентация размещения items (по
умолчанию "row")
    "visibility_head": boolean,    // Видимость заголовка (h1) (по умолчанию
true)
    "items": [                     // Массив элементов UI (tabs, graphics)
      // ... элементы ...
    ]
  }
}
```

Элементы `items`

Массив `items` в `controller` может содержать элементы типа `tabs` или `graphics`.

1. `tabs` (Вкладки)

Определяет блок вкладок.

```
{
  "type": "tabs",
  "id": "string",          // Уникальный идентификатор блока вкладок
  "visibility": boolean,  // Видимость блока вкладок (по умолчанию true)
  "width": "percentage", // Ширина блока в процентах (например, "50%")
(опционально)
  "tabs": [              // Массив вкладок
    {
      "id": "string",          // Уникальный идентификатор вкладки
      "title": "string",       // Заголовок вкладки
      "visibility": boolean,  // Видима ли вкладка (по умолчанию true)
      "groups": [              // Массив групп внутри вкладки
        {
          "id": "string",      // Уникальный идентификатор группы
          "title": "string",  // Заголовок группы
          "items": [           // Массив элементов внутри группы
            // ... элементы (parameter, command) ...
          ]
        }
      ]
    }
  ]
}
```

2. `graphics` (Графики)

Определяет блок для отображения графиков.

```
{
  "type": "graphics",
  "id": "string",          // Уникальный идентификатор блока графики
  "visibility": boolean,  // Видимость блока графики (по умолчанию true)
  "width": "percentage", // Ширина блока в процентах (например, "40%")
(опционально)
  "graphics": [          // Массив элементов графики (обычно один элемент -
сам график)
    {
      "id": "string",         // Уникальный идентификатор конкретного
графика
      "title": "string",      // Заголовок графика
      "visibility_title": boolean, // Видимость заголовка графика (по
умолчанию true)
      "frontend_props": {
        "type": "line",       // Тип диаграммы (пока поддерживается только
"line")
        "x_axis_label": "string", // Подпись оси X
        "y_axis_label": "string", // Подпись оси Y
        "x_range": { "min": number, "max": number }, // Диапазон по оси X
        "y_range": { "min": number, "max": number } или "auto", // Диапазон
по оси Y или авто-масштабирование
        "lines": [            // Массив линий (параметров/команд),
отображаемых на графике
          // ... элементы линий (x_constant, user_formula, real_time_data)
...
        ]
      }
    }
  ]
}
```

Типы элементов в `items` (вкладки)

Элементы внутри `items` вкладок определяются полем `"type"`.

`parameter` (Параметр)

Представляет собой элемент, который можно *отображать*, *изменять* (в некоторых случаях) и *читать*.

Общие поля:

- `"type"`: `"parameter"`
- `"param_id"`: `string` - Уникальный идентификатор параметра.
- `"display_name"`: `string` - Отображаемое имя параметра.
- `"frontend_type"`: `string` - Тип отображения параметра (см. ниже).
- `"units"`: `string` (опционально) - Единицы измерения, отображаются рядом с элементом.

- `"frontend_props"`: `object` (опционально) - Объект с дополнительными свойствами, специфичными для типа.

Поддерживаемые `frontend_type`:

A. `number` (Числовое поле ввода)

- **Описание:** Текстовое поле для ввода числа.
- `frontend_props`:
  - `"min"`: `number` (опционально) - Минимальное значение.
  - `"max"`: `number` (опционально) - Максимальное значение.
  - `"step"`: `number` (опционально) - Шаг изменения (для клавиатуры/колеса мыши).
  - `"default"`: `number` (опционально) - Значение по умолчанию.
  - `"style"`: `string` (опционально) - CSS-класс, применяемый к контейнеру элемента.

Пример:

```
{
  "type": "parameter",
  "param_id": "speed",
  "display_name": "Speed",
  "frontend_type": "number",
  "units": "RPM",
  "frontend_props": { "min": 0, "max": 3000, "step": 100, "default": 1500 }
}
```

B. `readonly` (Только для чтения)

- **Описание:** Поле, отображающее значение, которое нельзя изменить через UI.
- `frontend_props`:
  - `"default"`: `string` или `number` (опционально) - Значение по умолчанию.
  - `"style"`: `string` (опционально) - CSS-класс, применяемый к контейнеру элемента.

Пример:

```
{
  "type": "parameter",
  "param_id": "position",
  "display_name": "Position",
  "frontend_type": "readonly",
  "units": "steps",
  "frontend_props": { "default": 500 }
}
```

C. `flags` (Флаги)

- **Описание:** Набор чекбоксов, представляющих биты числа. Пользователь может отмечать/снимать флаги, значение передается как целое число.

- **`frontend_props`:**
  - `"bits"`: `object` - Объект, где ключи имеют формат `"N-описание"` (например, `"0-ready"`), а значения — отображаемые метки.
  - `"default"`: `number` (опционально) - Значение по умолчанию (битовая маска).
  - `"style"`: `string` (опционально) - CSS-класс, применяемый к контейнеру элемента.

Пример:

```
{
  "type": "parameter",
  "param_id": "status_flags",
  "display_name": "Status Flags",
  "frontend_type": "flags",
  "frontend_props": {
    "bits": {
      "0-ready": "Ready",
      "1-running": "Running",
      "2-error": "Error"
    },
    "default": 2
  }
}
```

D. `slider` (Слайдер)

- **Описание:** Горизонтальный ползунок для выбора числового значения.
- **`frontend_props`:**
  - `"min"`: `number` (опционально) - Минимальное значение.
  - `"max"`: `number` (опционально) - Максимальное значение.
  - `"step"`: `number` (опционально) - Шаг изменения.
  - `"default"`: `number` (опционально) - Значение по умолчанию.
  - `"show_value_display"`: `boolean` (опционально, по умолчанию `true`) - Показывать ли текущее значение рядом со слайдером.
  - `"style"`: `string` (опционально) - CSS-класс, применяемый к контейнеру элемента.

Пример:

```
{
  "type": "parameter",
  "param_id": "volume",
  "display_name": "Volume",
  "frontend_type": "slider",
  "units": "%",
  "frontend_props": {
    "min": 0,
    "max": 100,
    "step": 1,
    "default": 50,
    "show_value_display": true
```

```
    }
}
```

E. `select` (Выпадающий список)

- **Описание:** Элемент, позволяющий выбрать одно значение из предопределенного списка.
- **`frontend_props`:**
  - `"options"`: `array` - Массив объектов `{ "value": "...", "label": "..." }`, определяющих доступные опции.
  - `"default"`: `string` (опционально) - Значение по умолчанию (должно совпадать с одним из `value`).
  - `"style"`: `string` (опционально) - CSS-класс, применяемый к контейнеру элемента.

Пример:

```
{
  "type": "parameter",
  "param_id": "mode",
  "display_name": "Mode",
  "frontend_type": "select",
  "frontend_props": {
    "default": "auto",
    "options": [
      { "value": "auto", "label": "Automatic" },
      { "value": "manual", "label": "Manual" },
      { "value": "off", "label": "Off" }
    ]
  }
}
```

F. `radio` (Радиокнопки)

- **Описание:** Набор переключателей, позволяющий выбрать одно значение из предопределенного списка.
- **`frontend_props`:**
  - `"options"`: `array` - Массив объектов `{ "value": "...", "label": "..." }`, определяющих доступные опции.
  - `"default"`: `string` (опционально) - Значение по умолчанию (должно совпадать с одним из `value`).
  - `"style"`: `string` (опционально) - CSS-класс, применяемый к контейнеру элемента.

Пример:

```
{
  "type": "parameter",
  "param_id": "color",
  "display_name": "Color",
  "frontend_type": "radio",
```

```
    "frontend_props": {
      "default": "blue",
      "options": [
        { "value": "red", "label": "Red" },
        { "value": "green", "label": "Green" },
        { "value": "blue", "label": "Blue" }
      ]
    }
  }
```

`command` (Команда)

Представляет собой кнопку, при нажатии на которую генерируется событие.

Поля:

- `"type"`: `"command"`
- `"command_id"`: `string` - Уникальный идентификатор команды.
- `"display_name"`: `string` - Отображаемый текст кнопки.
- `"frontend_props"`: `object` (опционально) - Объект с дополнительными свойствами для настройки кнопки.
    - `"style"`: `string` (опционально) - CSS-класс для **неактивного** (или обычного) состояния кнопки (для `toggle`).
    - `"style_active"`: `string` (опционально) - CSS-класс для **активного** состояния кнопки (только для `toggle`).
    - `"style_clicked"`: `string` (опционально) - CSS-класс, применяемый **во время нажатия**.
    - `"display_name_active"`: `string` (опционально) - Текст кнопки в **активном** состоянии (только для `toggle`).
    - `"display_name_clicked"`: `string` (опционально) - Текст кнопки **во время нажатия**.
    - `"behavior"`: `string` (опционально, по умолчанию `"button"`) - Поведение кнопки: `"button"` (обычная) или `"toggle"` (переключатель).
    - `"auto_reset"`: `boolean` (опционально, по умолчанию `false`) - Автоматически сбрасывать `style_clicked` и `display_name_clicked` через `click_duration` мс.
    - `"click_duration"`: `number` (опционально, по умолчанию `200`) - Время в мс для `auto_reset`.
    - `"tooltip"`: `string` (опционально) - Текст всплывающей подсказки при наведении.
    - `"style"`: `string` (опционально) - CSS-класс, применяемый к **контейнеру** элемента команды (div.item).

Примеры:

- **Обычная кнопка:**

```
{
  "type": "command",
  "command_id": "START",
  "display_name": "Start Motor",
```

```
  "frontend_props": {
    "style": "base",
    "style_clicked": "clicked",
    "display_name_clicked": "Starting...",
    "auto_reset": true,
    "click_duration": 500,
    "tooltip": "Starts the motor.",
    "behavior": "button"
  }
}
```

- **Toggle-кнопка:**

```
{
  "type": "command",
  "command_id": "TOGGLE_LIGHT",
  "display_name": "Turn On Light",
  "frontend_props": {
    "style": "primary",
    "style_active": "success",
    "display_name_active": "Turn Off Light",
    "style_clicked": "clicked",
    "display_name_clicked": "Toggling...",
    "auto_reset": true,
    "click_duration": 300,
    "behavior": "toggle"
  }
}
```

Типы элементов в `lines` (графики)

Элементы внутри `lines` определяют **линии**, отображаемые **на графике**.

Общие поля:

- `"id"`: `string` - Уникальный идентификатор линии.
- `"visible"`: `boolean` - Видима ли линия (по умолчанию `true`).
- `"type"`: `string` - Тип линии (см. ниже).
- `"params"`: `object` - Параметры, специфичные для типа линии.
- `"frontend_props"`: `object` (опционально) - Объект с дополнительными свойствами.
  - `"round_precision"`: `number` (опционально) - Для линий `real_time_data` и `user_formula`. Определяет, до какого значения округляется `x` при обновлении. Ограничивает количество *уникальных* точек в буфере. По умолчанию `1` (Math.round).
- `"style"`: `object` - Стили для `Chart.js` dataset.
  - `"label"`: `string` - Метка линии в легенде.
  - `"borderColor"`: `string` - Цвет линии.
  - `"backgroundColor"`: `string` - Цвет фона (для заливки).
  - `"borderWidth"`: `number` - Толщина линии.

- `"borderDash"`: `array` - Паттерн пунктира `[длина_линии, длина_промежутка]`.
- `"pointRadius"`: `number` - Размер точек данных.
- `"fill"`: `boolean` - Заливать ли область под линией.
- `"order"`: `number` - Порядок отрисовки (меньше - ниже).

Поддерживаемые `type`:

A. `x_constant` (Вертикальная линия)

- **Описание:** Отображает вертикальную линию на заданной позиции `x`.
- **`params`:**
  - `"a"`: `number` - Координата `x`, где рисуется линия.
- **`frontend_props`:** Не используется.
- **`style`:** См. общие стили.

Пример:

```
{
  "id": "line_PLW",
  "visible": true,
  "type": "x_constant",
  "params": { "a": 20 },
  "style": {
    "borderColor": "rgba(0, 0, 255, 0.7)",
    "borderWidth": 3,
    "borderDash": [5, 5],
    "pointRadius": 0,
    "fill": false
  }
}
```

B. `user_formula` (Теоретическая кривая)

- **Описание:** Отображает кривую, вычисленную по формуле `y = f(x)`.
- **`formula`**: `string` - Математическое выражение, например, `"a * x * x + b * x + c"`. Используется `math.js` для вычисления.
- **`params`**: `object` - Параметры, используемые в формуле (например, `{ "a": 0.1, "b": 0, "c": 10 }`).
- **`frontend_props`:**
  - `"round_precision"`: `number` (опционально) - Точность округления `x` при вычислении точек формулы. Влияет на количество точек кривой. По умолчанию `1`.
- **`style`:** См. общие стили.

Пример:

```
{
  "id": "teor",
  "visible": true,
  "type": "user_formula",
```

```
  "formula": "a * x * x + b * x + c",
  "params": { "a": 0.1, "b": 0, "c": 10 },
  "frontend_props": {
    "round_precision": 0.1 // Более плавная кривая
  },
  "style": {
    "label": "Theoretical Curve",
    "borderColor": "rgba(255, 99, 132, 0.5)",
    "backgroundColor": "rgba(255, 99, 132, 0.1)",
    "borderWidth": 2,
    "borderDash": [5, 5],
    "pointRadius": 0,
    "fill": false,
    "order": 0 // Порядок отрисовки (ниже)
  }
}
```

C. `real_time_data` (Практические данные)

- **Описание:** Отображает точки данных, обновляемые в реальном времени.
- `params`: Не используется.
- `frontend_props`:
  - `"round_precision"`: `number` (опционально) - Точность округления `x` при добавлении новых точек через `updateLineData`. Ограничивает размер буфера. По умолчанию `1`.
- `style`: См. общие стили.

Пример:

```
{
  "id": "prakt",
  "visible": true,
  "type": "real_time_data",
  "params": {},
  "frontend_props": {
    "round_precision": 0.1 // Ограничивает количество точек
  },
  "style": {
    "label": "Practical Data",
    "borderColor": "rgb(75, 192, 192)",
    "backgroundColor": "rgba(75, 192, 192, 0.2)",
    "borderWidth": 2,
    "pointRadius": 3,
    "fill": false,
    "order": 1 // Порядок отрисовки (выше)
  }
}
```

Использование

1. Подключение скриптов

Убедитесь, что все файлы `.js` подключены в правильном порядке (согласно зависимостям) в вашем HTML или собраны в бандл.

2. Настройка `elementHandlers`

Создайте объект с обработчиками, которые будут вызываться при взаимодействии с UI (например, изменение параметра, клик команды).

```
const myElementHandlers = {
    onParameterChange: (paramId, value) => {
        console.log(`Handler: Parameter ${paramId} changed to ${value}`);
        // Отправить на сервер, обновить локальное состояние и т.д.
    },
    onCommand: (commandId) => {
        console.log(`Handler: Command ${commandId} executed`);
        // Отправить на сервер, выполнить действие и т.д.
    }
};
```

3. Создание `DataConnector` (опционально, но рекомендуется)

```
const dataConnector = new DataConnector(eventManagerFromGenerator); // Или
общий eventManager
dataConnector.connect('ws://localhost:8080/ws'); // URL вашего WebSocket
сервера
```

4. Создание и запуск `UIGenerator`

```
// Пример использования

const exampleSchema = {
    // ... (ваша существующая схема) ...
};

const handlers = {
    onParameterChange: (paramId, value) => {
        console.log(`Handler: Parameter ${paramId} changed to ${value}`);
    },
    onCommand: (commandId) => {
        console.log(`Handler: Command ${commandId} executed`);
    }
};

// Создаем генератор
const generator = new UIGenerator("controller_id", exampleSchema,
```

```
handlers);

// Если бы был WebSocket сервер, можно было бы подключить DataConnector
// const dataConnector = new DataConnector(generator.getEventManager());
// dataConnector.connect('ws://localhost:8080/ws');
// generator.setDataConnector(dataConnector);

// Генерируем UI
generator.generateUI('app');
```

5. Обновление UI извне

- **Обновить значение одного параметра:**

```
generator.updateParameter('position', 12345);
```

- **Обновить значения нескольких параметров:**

```
const data = {
    "parameters":
        {
            "speed": 300,
            "status_flags": 1,
            "position": 789,
        }

};
generator.updateMultipleParameters(data.parameters);
```

- **Прочитать текущее значение параметра:**

```
const currentValue = generator.getParameterValue('speed'); // Например,
получит значение из NumberParameterElement
console.log('Текущее значение speed:', currentValue);
```

- **Обновить данные линии графика (например, prakt):**

```
// Обновить *весь* буфер prakt новыми точками
generator.updateLineData('graph', 'prakt', [{x: 5, y: 12}, {x: 15, y:
15}]);

// Добавить *одну* точку к буферу prakt (требует обновления ChartRenderer
для append)
// generator.updateLineData('graph', 'prakt', {x: 10, y: 25});
```

- **Обновить параметры формулы:**

```
generator.updateFormulaParams('graph', 'teor', { a: 0.6, b: 12, c: 4 }); //
id графика, id формулы, новые параметры
```

- **Изменить видимость элемента (вкладки, группы, параметра, команды, линии графика):**

```
// Для элементов вкладок
generator.updateItemVisibility('config', false); // Скрыть группу 'config'
generator.updateItemVisibility('tabs_main', false); // Скрыть вкладки
generator.updateItemVisibility('graphics_main', false); // Скрыть график

// Для линии на графике
generator.updateLineVisibility('graph', 'teor', false); // Скрыть линию
'teor'
```

- **Изменить видимость заголовка контроллера:**

```
generator.setControllerHeaderVisibility(false); // Скрыть заголовок
контроллера
```

6. Уничтожение UI

Когда UI больше не нужен (например, при переключении между контроллерами), вызовите `destroyUI`, чтобы избежать утечек памяти.

```
generator.destroyUI();
```

События

Компоненты публикуют и подписываются на следующие события через `EventManager`:

- `PARAMETER_VALUE_CHANGED`: Публикуется при изменении значения параметра в UI.
  - Данные: `{ paramId, value, controllerName }`
- `COMMAND_CLICKED`: Публикуется при клике кнопки команды.
  - Данные: `{ commandId, controllerName }`
- `ELEMENT_VISIBILITY_CHANGED`: Публикуется при изменении видимости элемента (вкладки, группы, параметра, команды).
  - Данные: `{ elementId, isVisible }`
- `LINE_VISIBILITY_CHANGED`: Публикуется при изменении видимости линии на графике.
  - Данные: `{ graphId, lineId, isVisible }`
- `FORMULA_PARAMS_CHANGED`: Публикуется при изменении параметров формулы.
  - Данные: `{ graphId, formulaId, newParams, controllerName }`

- `SCHEMA_UPDATE_RECEIVED`: Публикуется `DataConnector` при получении новой схемы от сервера.
    - Данные: `{ schema }`

Валидация схемы

`UIGenerator` автоматически проверяет `stateSchema` с помощью `SchemaValidator`. При ошибках в схеме выводится сообщение в лог.

Логирование

Все компоненты используют `Logger`. Уровень логирования можно изменить в `GlobalLogger` или передавая индивидуальный `Logger` в конструкторы.

Безопасность и производительность

- **Утечки памяти:** Предотвращаются с помощью метода `destroy()` у всех `UIElement` и его потомков, который отписывает слушателей и очищает ссылки.
- **Производительность:** Для предотвращения подвисания можно разбивать большие операции на части с помощью `requestAnimationFrame` или `setTimeout`. Используется `requestAnimationFrame` для обновления `Chart.js`.
- **Обработка ошибок:** Критические участки кода обернуты в `try...catch`, особенно в `EventManager`.