CKEditor 5 Plugin Development

Case study

# Nikolay Volodin

INTRODUCTION

Nikolay Volodin

Senior Drupal Developer

We help our clients bring their digital experience to life

Georgia Tech

W UNIVERSITY of WASHINGTON

PRINCETON UNIVERSITY

Berkeley UNIVERSITY OF CALIFORNIA

EMORY UNIVERSITY

uOttawa

McGill

Queen's UNIVERSITY

NORTHERN MICHIGAN UNIVERSITY

UNIVERSITY OF WATERLOO

YORK U UNIVERSITÉ UNIVERSITY

U.S. HOUSE OF REPRESENTATIVES

Québec

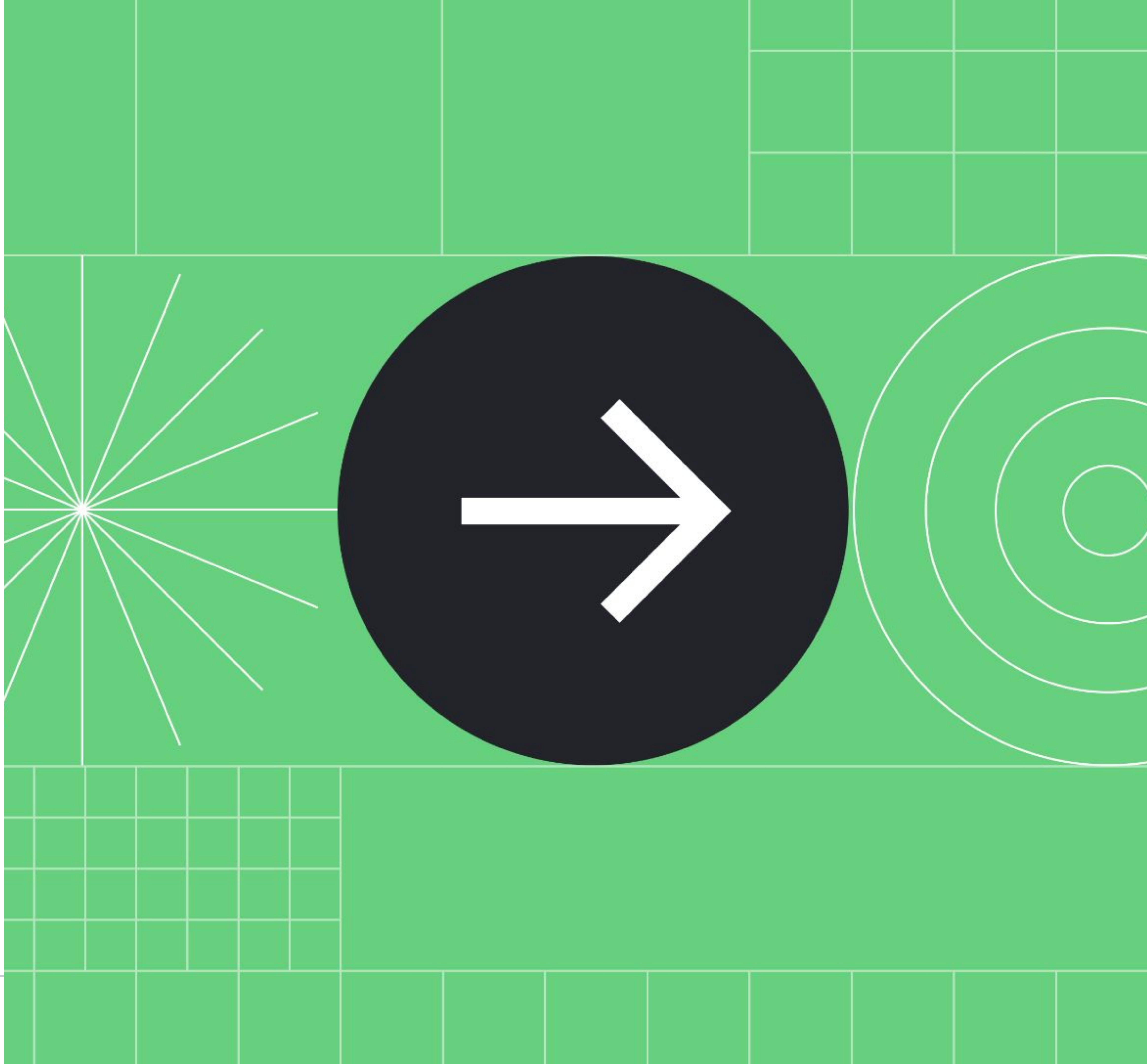
Canada

UNIVERSITY OF TORONTO

Agenda

1. Overview
2. Defining a new CKEditor 5 plugin in Drupal
3. Tools
4. Plugin structure
    a. Editing plugin
    b. UI plugin
    c. Command

Overview



Plugin repository

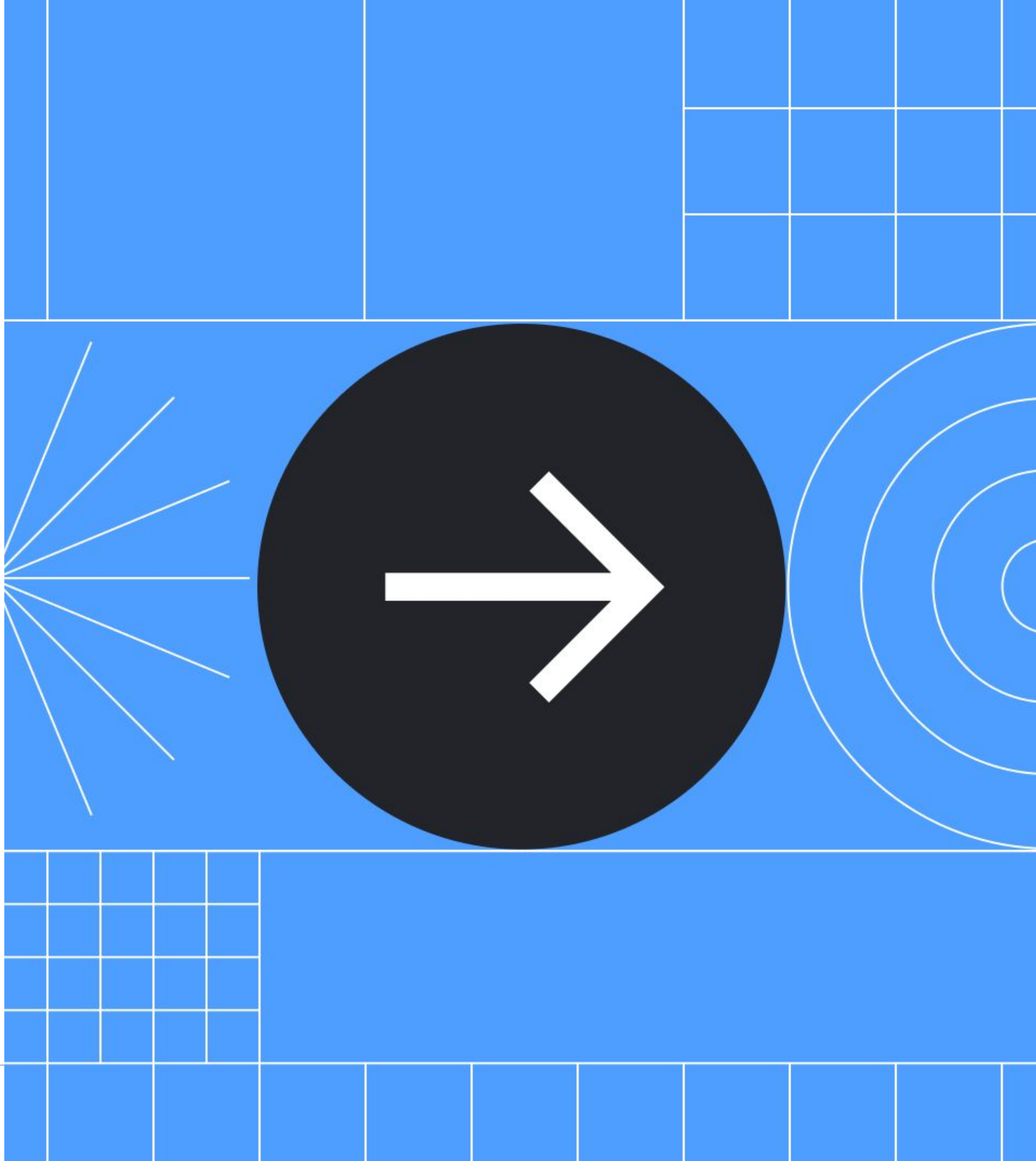
The code - https://github.com/klimp-drupal/ckeditor5_demo_link

```
"repositories": {
    "klimp-drupal/ckeditor5_demo_link":
{
        "type": "vcs",
        "url":
"git@github.com:klimp-drupal/ckeditor5_demo_link.git"
    }
},
"require": {
    "klimp-drupal/ckeditor5_demo_link":
"dev-master"
},
```

Migration from CKEditor 4

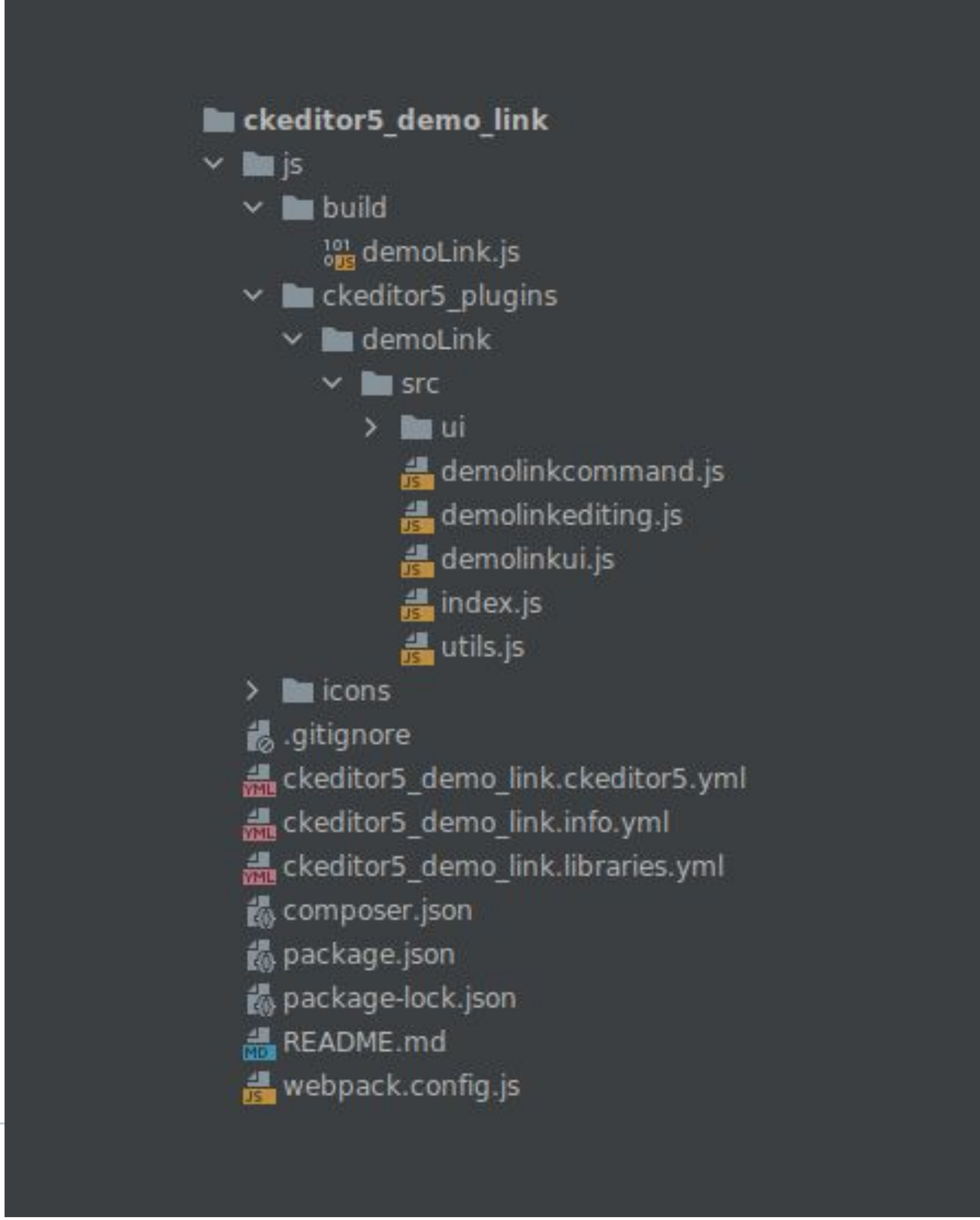
- CKEditor 5 is a rich-text editor with **MVC architecture**, **custom data model**, and **virtual DOM**. Compared to its predecessor, CKEditor 5 should be considered **a totally new editor.**
- Every single aspect of **it was redesigned**: integration, features, data model, API.
- There is **no automatic solution for migrating.**
- Any custom plugins for **CKEditor 4 will not be compatible with CKEditor 5.** Their implementation will be different and will **require rewriting** them **from scratch.**

CKEditor 5 plugin in Drupal



Plugin structure

- **<module_name>.ckeditor5.yml.** Defines:
  - Plugin
  - Library
  - Toolbar button
  - Parent HTML element
- **Library.** References the *js/build/demoLink.js* plugin file
- **Plugin source.** *js/ckeditor5_plugins/demoLink/src*



```
ckeditor5_demo_link
  js
    build
      demoLink.js
    ckeditor5_plugins
      demoLink
        src
          ui
          demolinkcommand.js
          demolinkediting.js
          demolinkui.js
          index.js
          utils.js
    icons
  .gitignore
  ckeditor5_demo_link.ckeditor5.yml
  ckeditor5_demo_link.info.yml
  ckeditor5_demo_link.libraries.yml
  composer.json
  package.json
  package-lock.json
  README.md
  webpack.config.js
```

CKEditor5.yml file

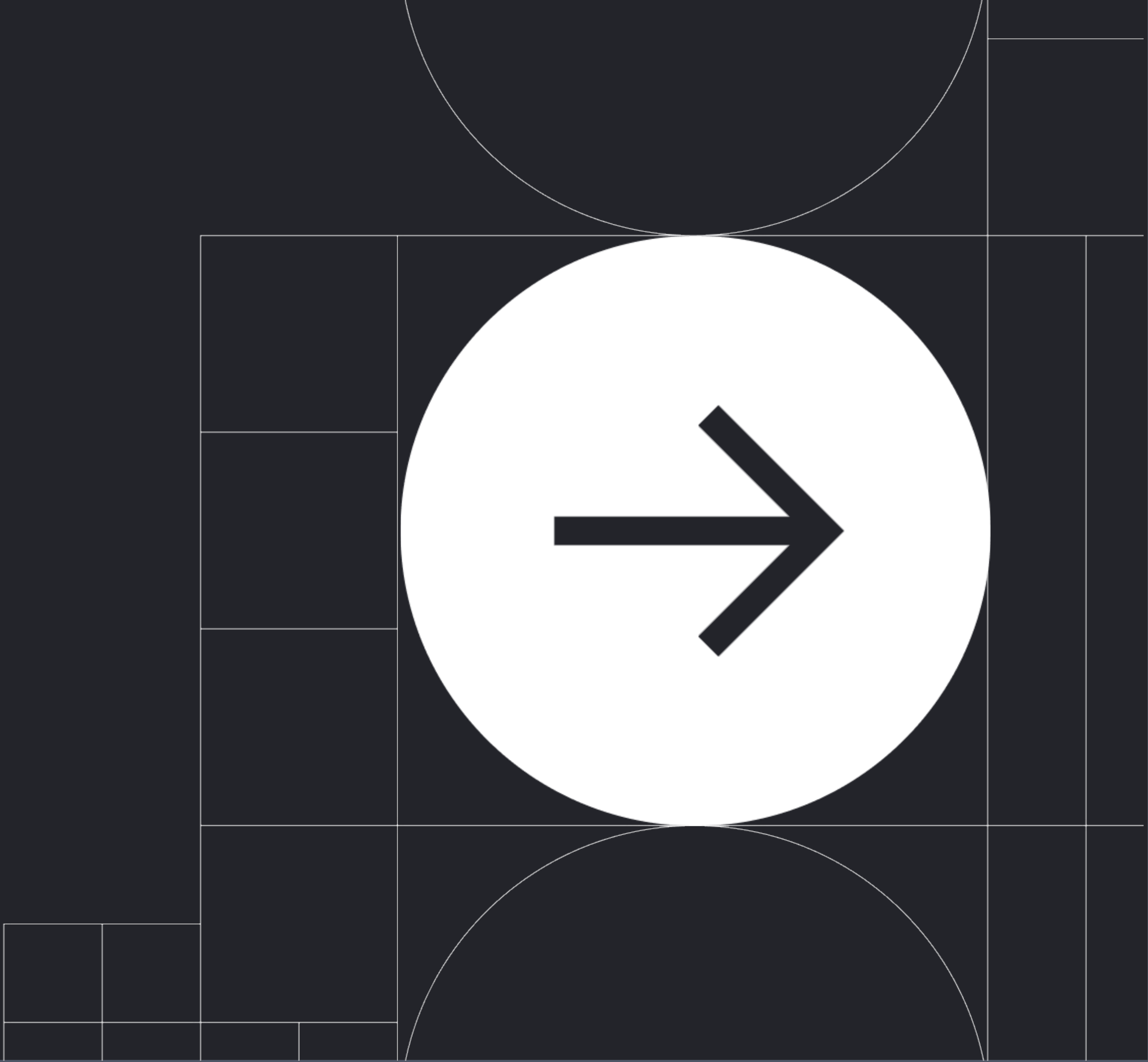
- CKEditor 5 part
- Drupal part
  - Label
  - Library
  - Toolbar button
  - Parent element

More info:

- [CKEditor 5 API overview]
- [CKEditor 5 architecture]

```yaml
ckeditor5_demo_link_demoLink:

 ckeditor5:
   plugins:
     - demoLink.DemoLink

 drupal:
   label: Demo Link

   # Drupal library with the plugin JS.
   library: ckeditor5_demo_link/demoLink

   # Toolbar button.
   toolbar_items:
     DemoLink:
       label: DemoLink

   # HTML elements to attach the plugin to.
   elements:
     - <p>
```

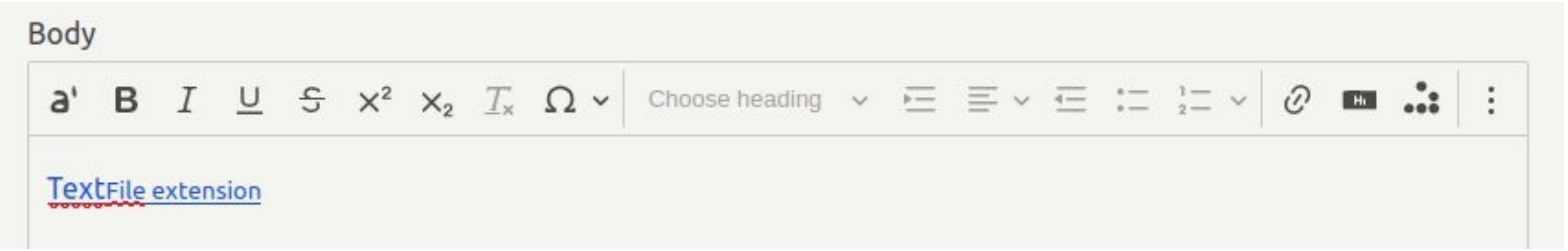
Tools



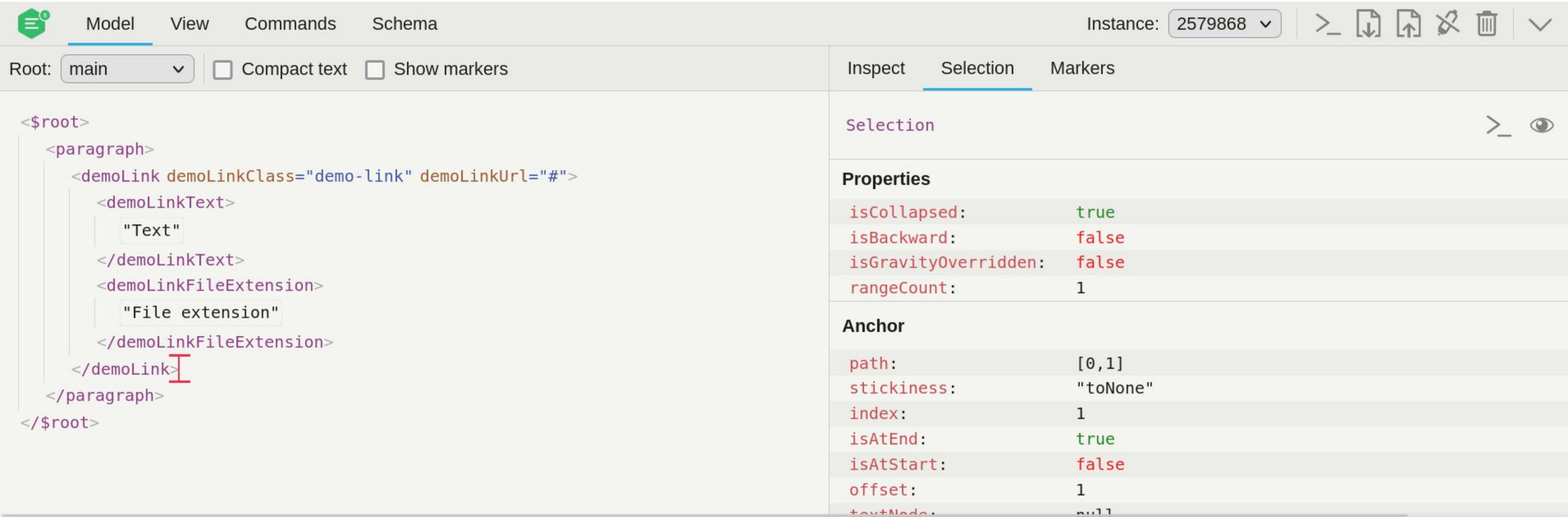
Tools to use

- **Webpack**. *webpack.config.js* - standardized across various modules, e.g. ckeditor_div_manager/webpack.config.js
- **CKEditor 5 Dev Tools** module
  - Demo CKEditor5 plugin example module - a demo module implementing the Block Widget demo plugin
  - CKEditor 5 inspector. Visualize and debug the model

CKEditor 5 inspector



TextFile extension



```
<$root>
  <paragraph>
    <demoLink demoLinkClass="demo-link" demoLinkUrl="#">
      <demoLinkText>
        "Text"
      </demoLinkText>
      <demoLinkFileExtension>
        "File extension"
      </demoLinkFileExtension>
    </demoLink>
  </paragraph>
</$root>
```

Selection

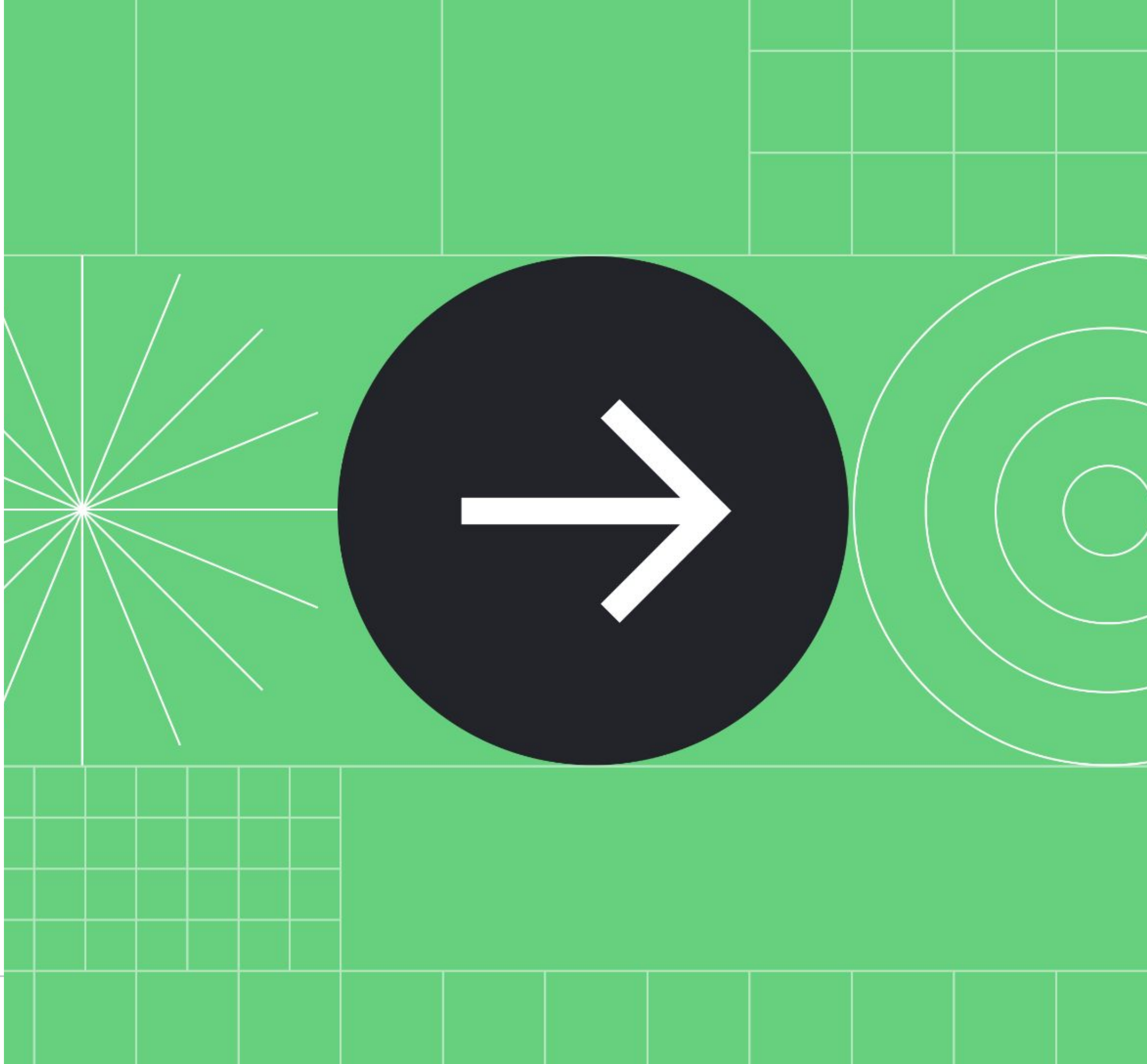
Properties

```
isCollapsed:            true
isBackward:             false
isGravityOverridden:    false
rangeCount:             1
```

Anchor

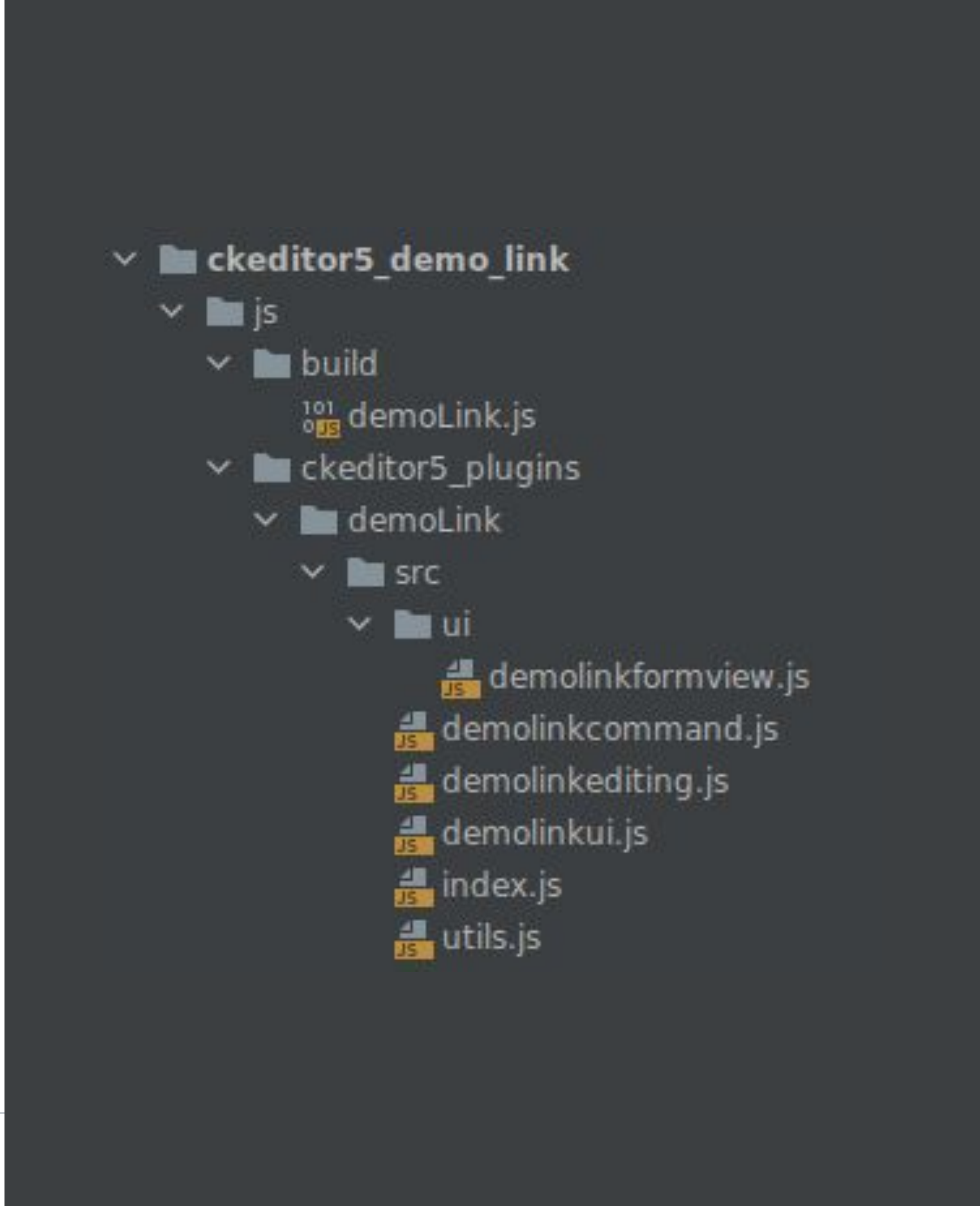
```
path:                   [0,1]
stickiness:             "toNone"
index:                  1
isAtEnd:                true
isAtStart:              false
offset:                 1
```

Plugin structure



Plugin's code

- Build
  - Webpack-minified js file
- Source
  - Index.js
  - **Editing plugin**
  - **UI plugin**
  - **Command**
  - Helper classes and files



```
ckeditor5_demo_link
  js
    build
      demoLink.js
    ckeditor5_plugins
      demoLink
        src
          ui
            demolinkformview.js
          demolinkcommand.js
          demolinkediting.js
          demolinkui.js
          index.js
          utils.js
```

Editing, UI & Command

Editing plugin:

- Defines elements' hierarchy
- Defines how data get converted from the abstract level to HTML and back

UI plugin

- Provides toolbar button
- Provides the form
- Handles selection

Command

- Modifies the element

PLUGIN STRUCTURE

index.js

js/ckeditor5_plugins/demoLink/src/index.js

- Is the **starting point**
- Technically could be the only file
- **Glues** together the **Editing** and **UI** plugins

```
import { Plugin } from 'ckeditor5/src/core';
import DemoLinkEditing from './demolinkediting';
import DemoLinkUI from './demolinkui';

/**
* The DemoLink plugin.
*
* This is a "glue" plugin that loads
* the {@link module:demoLink/DemoLinkEditing~DemoLinkEditing
DemoLink editing feature}
* and {@link module:demoLink/DemoLinkUI~DemoLinkUI DemoLink UI
feature}.
*
* @extends module:core/plugin~Plugin
*/
class DemoLink extends Plugin {

  /**
   * @inheritdoc
   */
  static get requires() {
    return [DemoLinkEditing, DemoLinkUI];
  }

  /**
   * @inheritdoc
   */
  static get pluginName() {
    return 'demoLink';
  }

}

export default {
  DemoLink,
};
```

Editing plugin

1. Elements’ hierarchy
2. How data get converted from the abstract level to HTML and back.

Model & View layers

[Model]()

- An abstract level of data representation
- May not correspond to HTML 1:1

[View]()

- HTML displayed
- Might be different for the End User and a Content Editor.

Schema, Conversion & Command

- Defines the model's Schema. How model elements can be **nested** + their allowed **attributes**
- Conversion
  - Upcast (View → Model)
  - Downcast (Model → View)
    - Editing pipeline. How the editor sees the plugin HTML
    - Data pipeline. How the end user sees the plugin HTML
- Binds the **command** to the editor

Schema

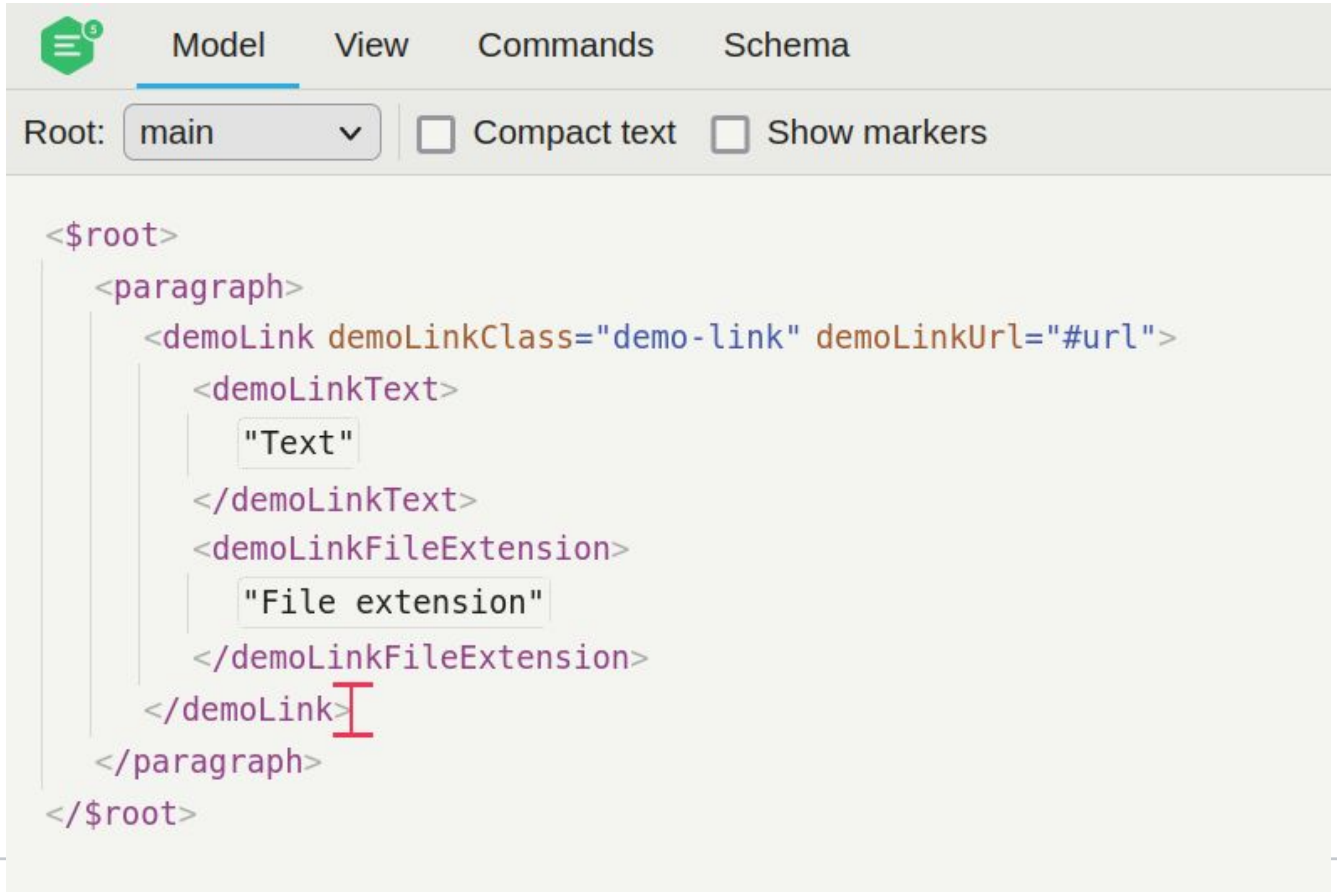
The model's schema defines the allowed and disallowed **structures** of nodes as well as nodes' **attributes**.

- **Where** an element is **allowed** or disallowed (e.g. *paragraph* is allowed in *$root*, but not in *heading1*).
- What **attributes** are allowed for a certain element (e.g. *image* can have the *src* and *alt* attributes).
- **Additional semantics** of model nodes (e.g. *image* is of the “object” type and paragraph of the “block” type).

```
// demoLink (parent element).
schema.register('demoLink', {
 inheritAllFrom: '$inlineObject',
 allowAttributes: [
   'demoLinkUrl',
   'demoLinkClass'


 ],
 allowChildren: [
   'demoLinkText',
   'demoLinkFileExtension',
 ],
});

// Link text (child element).
schema.register('demoLinkText', {
 allowIn: 'demoLink',
 isLimit: true,
 allowContentOf: '$block',
});
```

Model

The model is implemented by a **DOM-like tree structure** of elements and text nodes. Unlike in the actual DOM, in the model, **both elements and text nodes can have attributes.**

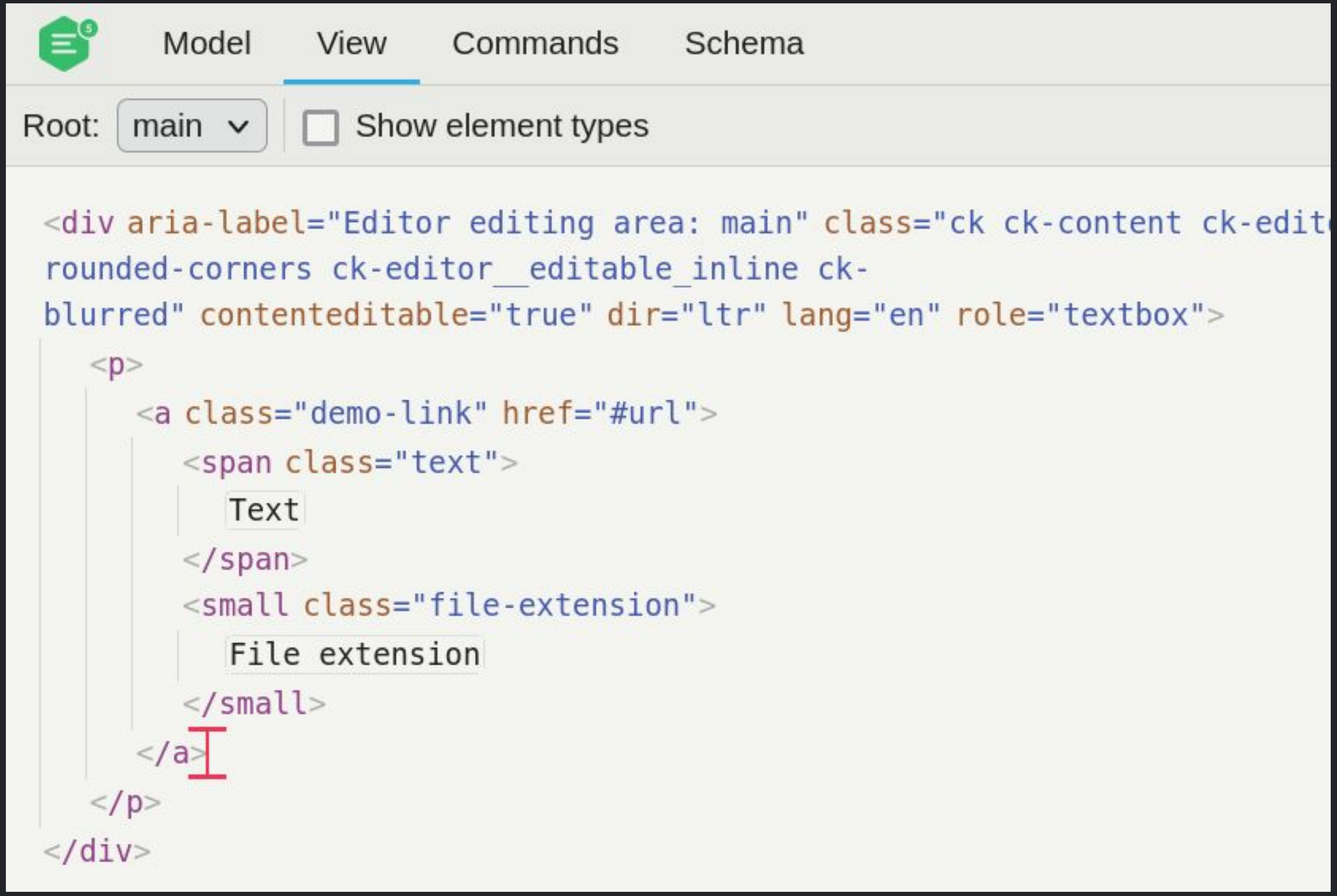


Root: main ☐ Compact text ☐ Show markers

```
<$root>
  <paragraph>
    <demoLink demoLinkClass="demo-link" demoLinkUrl="#url">
      <demoLinkText>
        "Text"
      </demoLinkText>
      <demoLinkFileExtension>
        "File extension"
      </demoLinkFileExtension>
    </demoLink>
  </paragraph>
</$root>
```

View

The View, on the other hand, is an abstract **representation of the DOM structure.**



Root: main ☐ Show element types

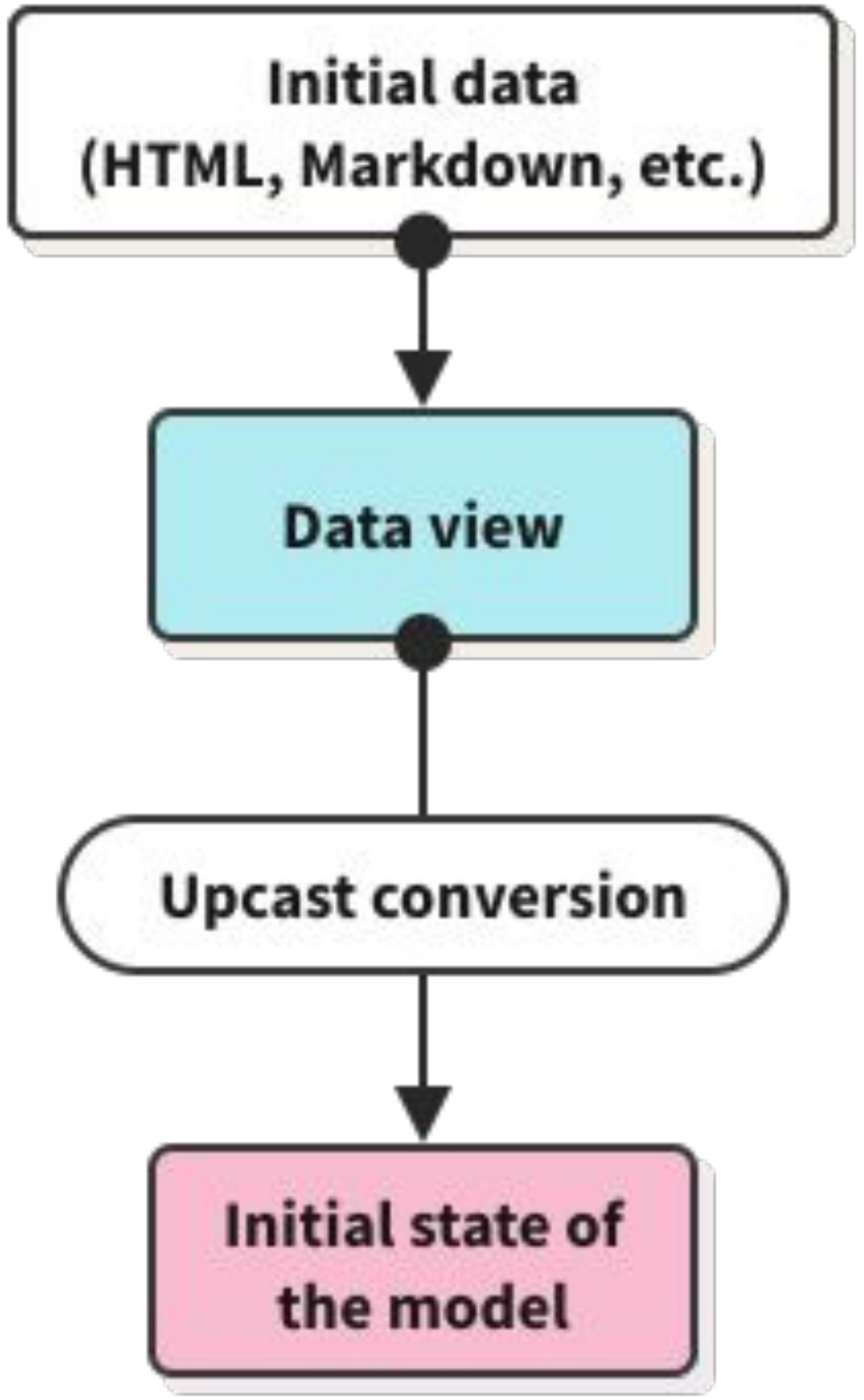
```
<div aria-label="Editor editing area: main" class="ck ck-content ck-edit
rounded-corners ck-editor__editable_inline ck-
blurred" contenteditable="true" dir="ltr" lang="en" role="textbox">
  <p>
    <a class="demo-link" href="#url">
      <span class="text">
        Text
      </span>
      <small class="file-extension">
        File extension
      </small>
    </a>
  </p>
</div>
```

Upcast Conversion

View → Model

1. **View** is created out of the markup.
2. With the help of the **upcast converters**, the **model** is created.
3. The model becomes the editor state.

The whole process is called **upcast conversion.**



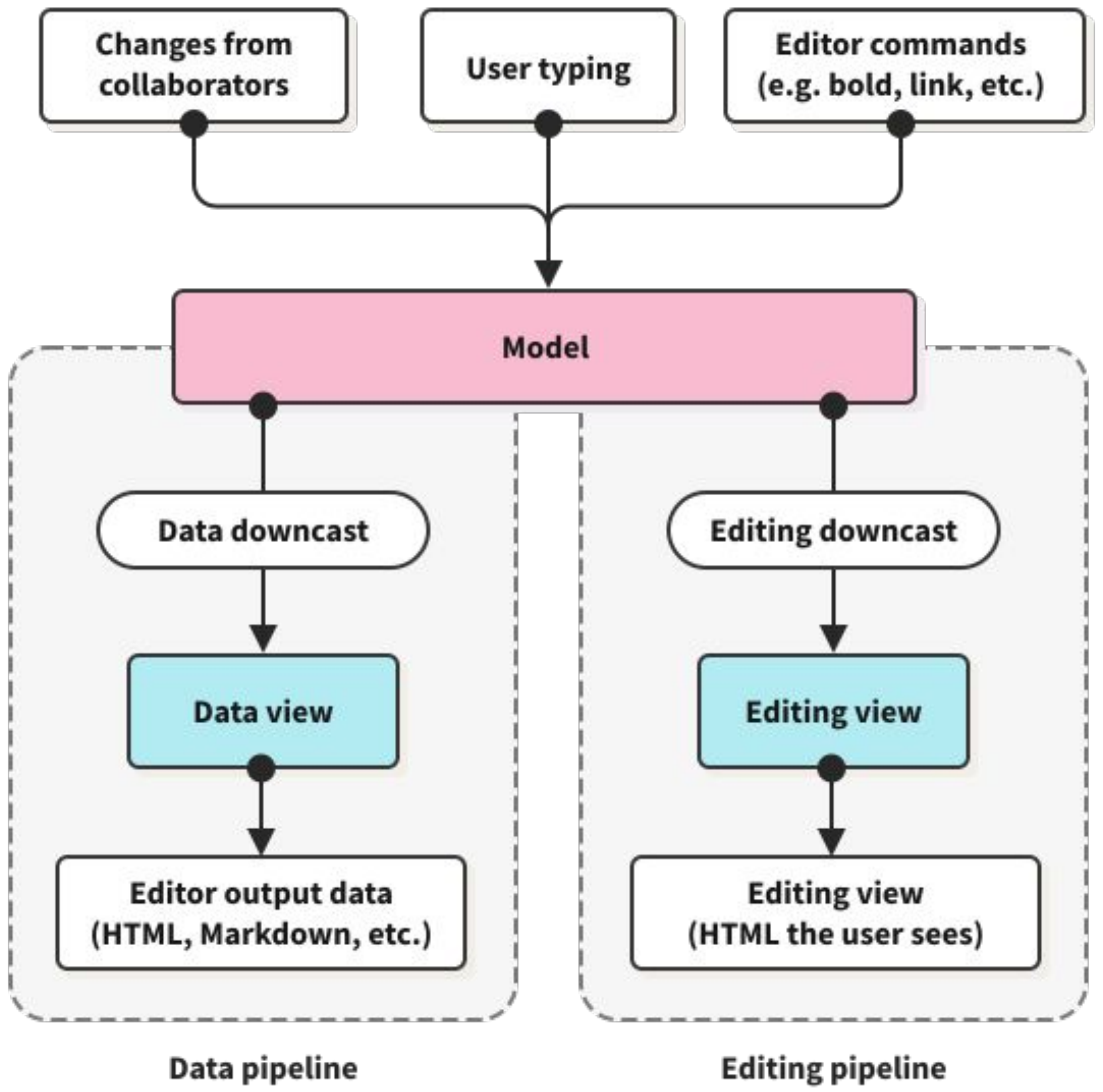
Downcast Conversion

Model → View

1. Changes (typing or pasting) are applied to the **model**.
2. To update the **editing view** (the layer being displayed to the user) the engine **transforms** these **changes** in the **model to the view.**

Editing pipeline. How the editor sees the plugin HTML

Data pipeline. How the end user sees the plugin HTML


Changes from collaborators
User typing
Editor commands (e.g. bold, link, etc.)
Model
Data downcast
Editing downcast
Data view
Editing view
Editor output data (HTML, Markdown, etc.)
Editing view (HTML the user sees)
Data pipeline
Editing pipeline

Conversion

The editing engine of CKEditor 5 works on **two separate layers** — **model** and **view**. The process of transforming one into the other is called **conversion**.

- Upcasting

  <span class="text"> to `demoLinkText`

  model element

- Downcasting

  `demoLinkText` model element to *<span class="text">*

```
// demoLinkText. View -> Model.
conversion.for('upcast').elementToElement({
 view: {
   name: 'span',
   classes: 'text',
 },
 model: (viewElement, { writer }) => {
   return writer.createElement('demoLinkText');
 }
});

// demoLinkText. Model -> View.
conversion.for('downcast').elementToElement({
 model: 'demoLinkText',
 view: (modelElement, {writer: viewWriter}) => {
   return viewWriter.createContainerElement(
     'span',
     {class: 'text'}
   );
 }
});
```

UI plugin

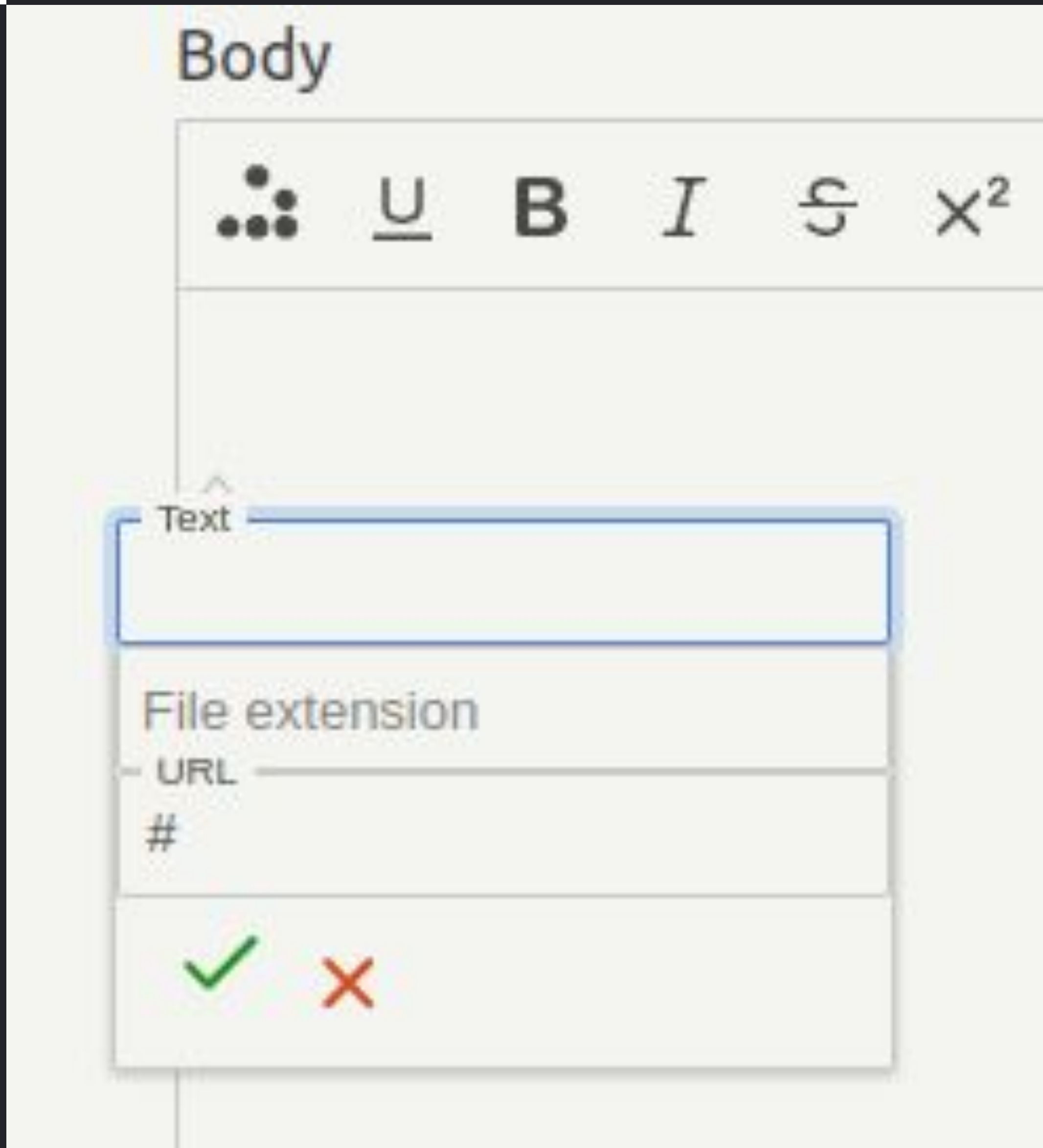
1. Toolbar button
2. Form
3. Selection

UI PLUGIN

Toolbar button, Form & Selection

- **Toolbar button.** On click - opens the Form
- **Form** - plugin configuration form. On submit executed the **command**
- **Selection.** Reacts on the **mouse click** or **arrow key** inside the plugin

```
/**
* @inheritDoc
*/
init() {
 // Create the balloon.
 this._balloon = this.editor.plugins.get( ContextualBalloon );

 this._addToolbarButton();
 this.formView = this._createFormView();
 this._handleSelection();
}
```



Toolbar button

Adds the toolbar button.

- Create new `ButtonView`
- Assign properties
- Link the command
  - Disable button depending on the command `isEnabled` property.
- Show UI (Form) on `execute`

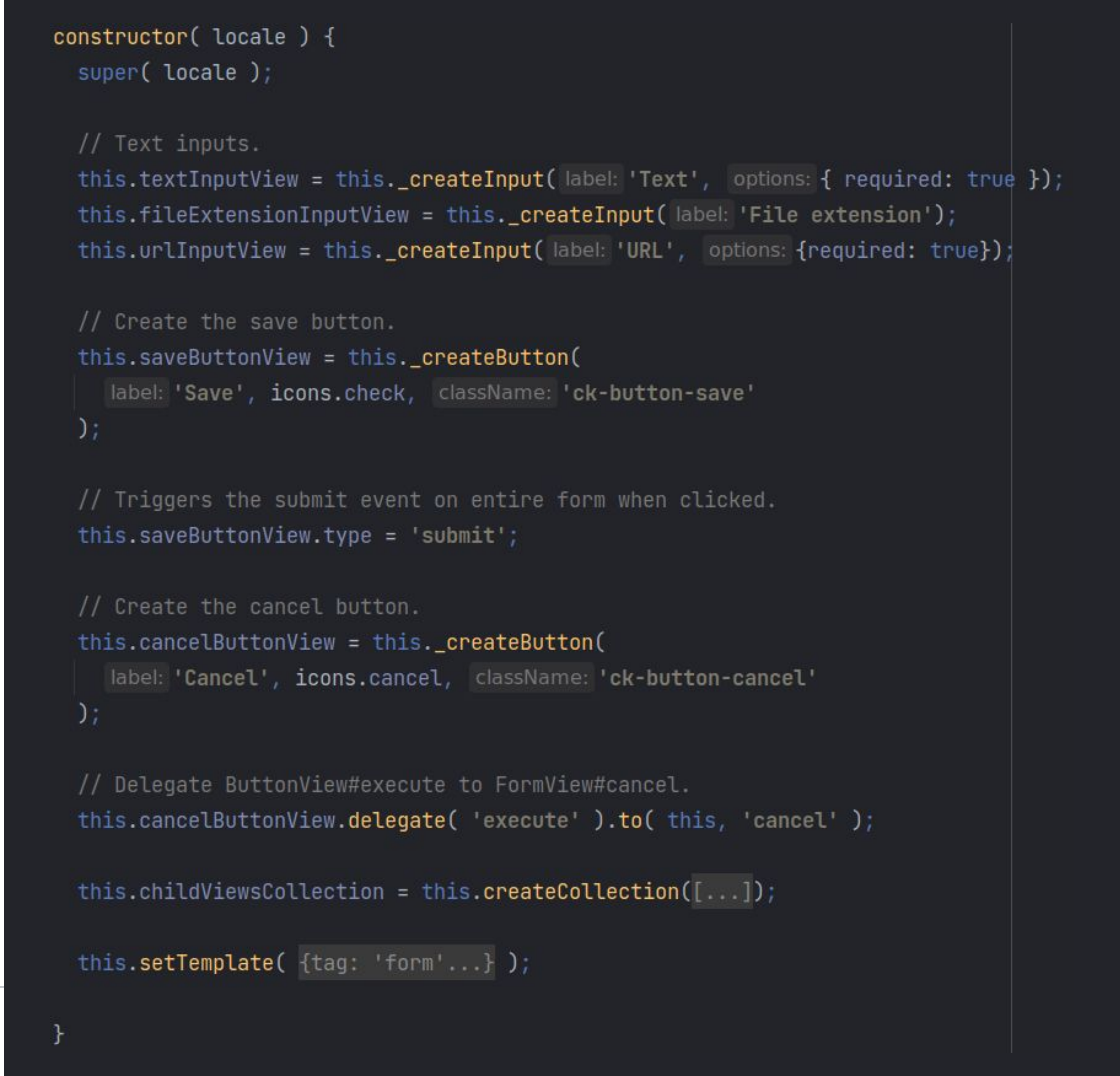
```
_addToolbarButton() {
 const editor = this.editor;

 editor.ui.componentFactory.add('demoLink', (locale) => {
   const buttonView = new ButtonView(locale);

   // Create the toolbar button.
   buttonView.set({
     label: editor.t('demoLink'),
     icon: demoLinkIcon,
     tooltip: true
   });

   // Bind button to the command.
   // The state on the button depends on the command values.
   const command = editor.commands.get('demoLink');
   buttonView.bind( 'isEnabled' ).to( command, 'isEnabled' );
   buttonView.bind( 'isOn' ).to( command, 'value', value => !!value );

   // Execute the command when the button is clicked.
   this.listenTo(buttonView, 'execute', () =>
     this._showUI(),
   );

   return buttonView;
 });
}
```

UI PLUGIN

FormView

Helper class to create the form.

- Create **text input** fields
- Create **buttons**
  - Save
  - Cancel
- Put form fields into the `ViewsCollection`
- Pass `ViewsCollection` to the `Template`



```
constructor( locale ) {
  super( locale );

  // Text inputs.
  this.textInputView = this._createInput( label: 'Text', options: { required: true });
  this.fileExtensionInputView = this._createInput( label: 'File extension');
  this.urlInputView = this._createInput( label: 'URL', options: {required: true});

  // Create the save button.
  this.saveButtonView = this._createButton(
    label: 'Save', icons.check, className: 'ck-button-save'
  );

  // Triggers the submit event on entire form when clicked.
  this.saveButtonView.type = 'submit';

  // Create the cancel button.
  this.cancelButtonView = this._createButton(
    label: 'Cancel', icons.cancel, className: 'ck-button-cancel'
  );

  // Delegate ButtonView#execute to FormView#cancel.
  this.cancelButtonView.delegate( 'execute' ).to( this, 'cancel' );

  this.childViewsCollection = this.createCollection([...]);

  this.setTemplate( {tag: 'form'...} );

}
```

Creating the Form

- Create new `FormView` (custom helper)
- `submit` handler
  - Collect form **values**
  - Pass it to the **command**
  - **Hide** the **UI** (Form)
- `cancel` handler
  - Hide the UI (Form)
- Click outside of the plugin handler
  - Hide the UI (Form)

```
_createFormView() {
 // The FormView defined in src/ui/demolinkformview.js
 const formView = new FormView(this.editor.locale);

 // Form submit handler.
 this.listenTo(formView, 'submit', () => {

   let values = {
     demoLinkText: formView.textInputView.fieldView.element.value,
     demoLinkFileExtension: formView.fileExtensionInputView.fieldView.element.value,
     demoLinkUrl: formView.urlInputView.fieldView.element.value,
   };

   this.editor.execute('demoLink', values);

   // Hide the form view after submit.
   this._hideUI();
 });

 // Hide the form view after clicking the "Cancel" button.
 this.listenTo( formView, 'cancel', () => {
   this._hideUI();
 } );


 // Hide the form view when clicking outside the balloon.
 clickOutsideHandler( {
   emitter: formView,
   activator: () => this._balloon.visibleView === formView,
   contextElements: [ this._balloon.view.element ],
   callback: () => this._hideUI()
 } );

 return formView;
}
```

Adding the values

Adds the **form** to the balloon and **populates** its fields.

- **Add** the form to the `balloon`
- **Iterate** through the form elements
  - **Get** the **value** for the form element **from the command**
  - **Assign the value** to the form element
- **Set** the form **focus**

```
_addFormView() {

  this._balloon.add({
    view: this.formView,
    position: this._getBalloonPositionData()
  });

  const command = this.editor.commands.get('demoLink');

  const modelToFormFields = {
    demoLinkText: 'textInputView',
    demoLinkFileExtension: 'fileExtensionInputView',
    demoLinkUrl: 'urlInputView',
  };

  // Handle text input fields.
  Object.entries(modelToFormFields).forEach(([modelName, formElName]) => {

    const formEl = this.formView[formElName];

    // Needed to display a placeholder of the elements being focused before.
    formEl.focus();

    const isEmpty = !command.value || !command.value[modelName] || command.value[modelName] === '';

    // Set URL default value.
    if (modelName === 'demoLinkUrl' && isEmpty) {
      formEl.fieldView.element.value = '#';
      formEl.set('isEmpty', false);
      return;
    }

    if (!isEmpty) {
      formEl.fieldView.element.value = command.value[modelName];
    }
    formEl.set('isEmpty', isEmpty);

  });

  // Reset the focus to the first form element.
  this.formView.focus();
}
```

UI PLUGIN

Selection

`selectionChange` **event** listener:

- Check if the **selected element is not outside** the `demoLink`
- **Identifies** the **last child** element (`demoLinkText` or `demoLinkFileExtension`)
- **Identifies the boundaries** of the `demoLink` element
- If the **“border”** is selected (right before or after the element) - **move the selection** to the element’s ancestor (*paragraph*)

```
_handleSelection() {
 const editor = this.editor;

 this.listenTo(editor.editing.view.document, 'selectionChange', (eventInfo, eventData) => {
   const selection = editor.model.document.selection;

   let el = selection.getSelectedElement() ?? selection.getFirstRange().getCommonAncestor();

   // The selected element is outside of a demo link.
   if (!['demoLinkText', 'demoLinkFileExtension'].includes(el.name)) {
     this._hideUI();
     return;
   }

   this._showUI();

   const positionBefore = editor.model.createPositionBefore(el);
   const positionAfter = editor.model.createPositionAfter(el);

   const position = selection.getFirstPosition();

   // Define which child element will be used for afterTouch;
   const demoLinkEl = findElement(selection, 'demoLink');
   var hasFileExtension = false;
   for (const child of demoLinkEl.getChildren()) {
     if (child.name === 'demoLinkFileExtension') {
       hasFileExtension = true;
       continue;
     }
   }
   const afterTouchChildElName = hasFileExtension ? 'demoLinkFileExtension' : 'demoLinkText';

   const beforeTouch = el.name == 'demoLinkText' && position.isTouching( positionBefore );
   const afterTouch = el.name == afterTouchChildElName && position.isTouching( positionAfter );

   // Handle the "border" selection.
   if (beforeTouch || afterTouch) {
     editor.model.change(writer => {
       writer.setSelection(el.findAncestor('demoLink'), 'on');
     });
   }

 });
}
```

Command

Modified the model element.

Command

Commands are the main way to **manipulate** the editor **contents and state**. They are mostly **used by UI elements** (or by other commands) to **make changes in the model.** Commands are available in every part of the code that has access to the editor instance.

- refresh() - Refreshes the command. The command should **update** its isEnabled and value **properties** in this method
  - Command value property is used to **keep the configuration form values up to date**
- execute() - **Adds or modifies** a plugin instance based on the values received from the **plugin configuration form**

COMMAND

refresh() method

Updates `isEnabled` and `value` properties.

- **Initialize** `isEnabled` and `value` properties.
- **Verify** that the element is in the `selection`
- Assign the `demoLink` model **attributes** (`demoLinkUrl` and `demoLinkClass`) to the **value** property (used by the form)
- Assign the `demoLink` **child elements** values (`demoLinkText` and `demoLinkFileExtension`) to the **value** property (used by the form)

```
refresh() {
  // Demo link Toolbar button is always enabled.
  this.isEnabled = true;

  // Init the empty command value.
  this.value = null;

  // Find the element in the selection.
  const { selection } = this.editor.model.document;
  const demoLinkEl = findElement(selection, 'demoLink');
  if (!demoLinkEl) {
    return;
  }

  // Populate command value.
  this.value = {};

  // Process demoLink attributes (demoLinkUrl & demoLinkClass).
  for (const [attrKey, attrValue] of demoLinkEl.getAttributes()) {
    this.value[attrKey] = attrValue;
  }

  // Process demoLink children (demoLinkText & demoLinkFileExtension).
  for (const childNode of demoLinkEl.getChildren()) {
    const childTextNode = childNode.getChild(0);
    const dataNotEmpty = childTextNode && childTextNode._data;
    this.value[childNode.name] = dataNotEmpty ? childTextNode._data : '';
  }
}
```

COMMAND

execute() method

Modifies the **model** element.

On `model.change()` event:

- **Find** an **existing** element or **create** new
- `this._editElement()` modified the element
- **Insert** the element of **new**

```
execute(values) {
 const { model } = this.editor;

 model.change((writer) => {

   // If a new button is created or an existing one is being edited.
   var isNew = false;

   // Find an existing demo link if it is being edited.
   var demoLinkEl = findElement(model.document.selection, 'demoLink');

   // Create new demoLink.
   if (!demoLinkEl) {
     demoLinkEl = writer.createElement('demoLink');
     isNew = true;
   }

   // Editing the model element and its children to match the form
values.
   this._editElement(writer, demoLinkEl, values);

   // Insert a new button.
   if (isNew) {
     model.insertContent(demoLinkEl);
   }

 });
}
```

COMMAND

Deeper look

Re-creates model **attributes** and **children**.

- **Clear** model **attributes**
- **Set** new model **attributes**
- **Re-create child** elements (`demoLinkText` and `demoLinkFileExtension`)
- **Append child** elements **to** the **parent** model element

```
_editElement(writer, modelEl, values) {
 // Clear modelEl attributes.
 writer.clearAttributes(modelEl);

 // Set modelEl attributes.
 var modelAttrs = {};
 modelAttrs.demoLinkUrl = values['demoLinkUrl'];
 modelAttrs.demoLinkClass = 'demo-link';
 writer.setAttributes(modelAttrs, modelEl);

 // Get modelEl children elements names.
 const children = [];
 Array.from(modelEl.getChildren()).forEach((el) => {
   children.push(el.name);
 });

 // Get or create child elements.
 const demoLinkText = this._processChildTextEl(writer, values, children,
modelEl, 'demoLinkText');
 const demoLinkFileExtension = this._processChildTextEl(writer, values,
children, modelEl, 'demoLinkFileExtension');

 // Append child element in a proper order.
 if (demoLinkText) {
   writer.append(demoLinkText, modelEl);
 }
 if (demoLinkFileExtension) {
   writer.append(demoLinkFileExtension, modelEl);
 }

}
```

Thank you

evolvingweb