Application Architecture: MVC Pattern

> 🎓 **CS Perspective:** MVC (Model-View-Controller) is a foundational **architectural pattern** from software engineering that enforces **separation of concerns**—a design principle stating that each module should have a single responsibility. The Model manages **application state** and business logic (the "truth"), the View handles **presentation** (UI rendering), and the Controller acts as **mediator** between them. This pattern implements a form of the **Observer pattern**: Views subscribe to Model changes. The Publisher-Subscriber pattern used here enables **loose coupling**—components communicate through events rather than direct references. Configuration modules demonstrate the **single source of truth** principle, while helper modules show **DRY (Don't Repeat Yourself)** through code reuse.

📌 MVC Overview

Model-View-Controller separates concerns:

Component	Responsibility	Example
Model	Business logic, state, data	API calls, data transformations
View	Presentation logic	DOM manipulation, rendering
Controller	Coordinates Model & View	Event handlers, app flow

```
User Action → Controller → Model (update state)
                              ↓
                        View ← (new data)
```

📌 Project Structure

```
src/
├── js/
│   ├── controller.js      # Application logic
│   ├── model.js           # State & business logic
│   ├── config.js          # Constants & configuration
│   ├── helpers.js         # Reusable utility functions
│   └── views/
│       ├── View.js        # Parent class
│       ├── recipeView.js  # Specific views
│       └── ...
```

📌 Configuration Module

Centralize magic values:

```
// config.js
export const API_URL = 'https://api.example.com/';
export const TIMEOUT_SEC = 10;
export const RES_PER_PAGE = 10;
export const MODAL_CLOSE_SEC = 2.5;
```

Benefits:

- Easy to find and change values
- Single source of truth
- Self-documenting code

📌 Helper Functions Module

Reusable utilities:

```
// helpers.js
const timeout = function(s) {
  return new Promise((_, reject) => {
    setTimeout(() => {
      reject(new Error(`Request timed out after ${s}s`));
    }, s * 1000);
  });
};

export const AJAX = async function(url, uploadData = undefined) {
  try {
    const fetchPro = uploadData
      ? fetch(url, {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify(uploadData),
        })
      : fetch(url);

    // Race against timeout
    const res = await Promise.race([fetchPro, timeout(TIMEOUT_SEC)]);
    const data = await res.json();

    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
    return data;
  } catch (err) {
    throw err; // Re-throw to handle in caller
  }
};
```

📌 Model: State Management

Centralized application state:

```
// model.js
export const state = {
  recipe: {},
  search: {
    query: '',
    results: [],
    page: 1,
    resultsPerPage: RES_PER_PAGE,
  },
  bookmarks: [],
};

// Data fetching
export const loadRecipe = async function(id) {
  try {
    const data = await AJAX(`${API_URL}${id}`);
    state.recipe = createRecipeObject(data);
  } catch (err) {
    throw err;
  }
};

// Data transformation (API → App format)
const createRecipeObject = function(data) {
  const { recipe } = data.data;
  return {
    id: recipe.id,
    title: recipe.title,
    image: recipe.image_url,      // Rename properties
    servings: recipe.servings,
    // Conditionally add property
    ...(recipe.key && { key: recipe.key }),
  };
};
```

Conditional Object Property

```
// Add property only if it exists
...(recipe.key && { key: recipe.key })

// Equivalent to:
if (recipe.key) obj.key = recipe.key;
```

📌 View: Parent Class Pattern

Base class with shared functionality:

```
// View.js
export default class View {
  _data;
  _parentElement;
  _errorMessage = 'Something went wrong';

  render(data, render = true) {
    if (!data || (Array.isArray(data) && data.length === 0))
      return this.renderError();

    this._data = data;
    const markup = this._generateMarkup();

    if (!render) return markup;  // Return string instead

    this._clear();
    this._parentElement.insertAdjacentHTML('afterbegin', markup);
  }

  _clear() {
    this._parentElement.innerHTML = '';
  }

  renderSpinner() {
    const markup = `<div class="spinner">...</div>`;
    this._clear();
    this._parentElement.insertAdjacentHTML('afterbegin', markup);
  }

  renderError(message = this._errorMessage) {
    const markup = `<div class="error"><p>${message}</p></div>`;
    this._clear();
    this._parentElement.insertAdjacentHTML('afterbegin', markup);
  }

  renderMessage(message = this._message) {
    const markup = `<div class="message"><p>${message}</p></div>`;
    this._clear();
    this._parentElement.insertAdjacentHTML('afterbegin', markup);
  }
}
```

📌 View: Child Classes

Extend parent, implement specifics:

```
// recipeView.js
import View from './View.js';

class RecipeView extends View {
  _parentElement = document.querySelector('.recipe');
  _errorMessage = 'Could not find recipe!';

  // Publisher: accepts handler from controller
  addHandlerRender(handler) {
    ['hashchange', 'load'].forEach(ev =>
      window.addEventListener(ev, handler)
    );
  }

  // Event delegation with data attributes
  addHandlerUpdateServings(handler) {
    this._parentElement.addEventListener('click', function(e) {
      const btn = e.target.closest('.btn--update-servings');
      if (!btn) return;

      const { updateTo } = btn.dataset;
      if (+updateTo > 0) handler(+updateTo);
    });
  }

  _generateMarkup() {
    return `
      <h1>${this._data.title}</h1>
      <img src="${this._data.image}" />
    `;
  }
}

export default new RecipeView(); // Export instance
```

📌 DOM Update Algorithm

Update only changed elements (avoid full re-render):

```
update(data) {
  this._data = data;
  const newMarkup = this._generateMarkup();

  // Create virtual DOM from string
  const newDOM =
document.createRange().createContextualFragment(newMarkup);
  const newElements = Array.from(newDOM.querySelectorAll('*'));
  const curElements =
Array.from(this._parentElement.querySelectorAll('*'));
```

```
  newElements.forEach((newEl, i) => {
    const curEl = curElements[i];

    // Update changed TEXT
    if (
      !newEl.isEqualNode(curEl) &&
      newEl.firstChild?.nodeValue.trim() !== ''
    ) {
      curEl.textContent = newEl.textContent;
    }

    // Update changed ATTRIBUTES
    if (!newEl.isEqualNode(curEl)) {
      Array.from(newEl.attributes).forEach(attr =>
        curEl.setAttribute(attr.name, attr.value)
      );
    }
  });
}
```

📌 Publisher-Subscriber Pattern

Views publish events, Controller subscribes:

```
// VIEW (Publisher)
addHandlerRender(handler) {
  window.addEventListener('hashchange', handler);
}

// CONTROLLER (Subscriber)
const controlRecipes = async function() {
  const id = window.location.hash.slice(1);
  if (!id) return;

  recipeView.renderSpinner();
  await model.loadRecipe(id);
  recipeView.render(model.state.recipe);
};

// Connect publisher to subscriber
const init = function() {
  recipeView.addHandlerRender(controlRecipes);
};
init();
```

Why?

- Views don't import Controller (would create circular dependency)

- Controller passes handler to View
- View calls handler when event occurs

📌 Controller: Orchestration

```
// controller.js
import * as model from './model.js';
import recipeView from './views/recipeView.js';

const controlRecipes = async function() {
  try {
    const id = window.location.hash.slice(1);
    if (!id) return;

    // 1. Loading indicator
    recipeView.renderSpinner();

    // 2. Load data (Model)
    await model.loadRecipe(id);

    // 3. Render (View)
    recipeView.render(model.state.recipe);

  } catch (err) {
    recipeView.renderError();
  }
};

// Initialize
const init = function() {
  recipeView.addHandlerRender(controlRecipes);
  searchView.addHandlerSearch(controlSearchResults);
  paginationView.addHandlerClick(controlPagination);
};
init();
```

📌 Pagination Logic

```
// model.js
export const getSearchResultsPage = function(page = state.search.page) {
  state.search.page = page;

  const start = (page - 1) * state.search.resultsPerPage;
  const end = page * state.search.resultsPerPage;

  return state.search.results.slice(start, end);
};
```

```
// Page 1: slice(0, 10)  → items 0-9
// Page 2: slice(10, 20) → items 10-19
```

📌 Local Storage Persistence

```
// Save
const persistBookmarks = function() {
  localStorage.setItem('bookmarks', JSON.stringify(state.bookmarks));
};

// Load on init
const init = function() {
  const storage = localStorage.getItem('bookmarks');
  if (storage) state.bookmarks = JSON.parse(storage);
};
init();
```

📌 Hash-Based Routing

```
// Get ID from URL hash
const id = window.location.hash.slice(1);
// URL: example.com/#abc123 → id = 'abc123'

// Listen for hash changes
window.addEventListener('hashchange', handler);
window.addEventListener('load', handler);

// Update URL without reload
window.history.pushState(null, '', `#${newId}`);
```

📌 Data Attributes for Actions

```
<button data-update-to="4" class="btn--update-servings">+</button>
```

```
addHandlerUpdateServings(handler) {
  this._parentElement.addEventListener('click', function(e) {
    const btn = e.target.closest('.btn--update-servings');
    if (!btn) return;

    const { updateTo } = btn.dataset;  // '4'
    handler(+updateTo);                // 4
```

```
  });
}
```

📌 Error Handling Pattern

```
// Helpers - throw error
export const AJAX = async function(url) {
  try {
    const res = await fetch(url);
    if (!res.ok) throw new Error(`${res.status}`);
    return await res.json();
  } catch (err) {
    throw err;  // Re-throw!
  }
};

// Model - propagate error
export const loadRecipe = async function(id) {
  try {
    const data = await AJAX(`${API_URL}${id}`);
    state.recipe = createRecipeObject(data);
  } catch (err) {
    throw err;  // Re-throw!
  }
};

// Controller - handle error
const controlRecipes = async function() {
  try {
    await model.loadRecipe(id);
    recipeView.render(model.state.recipe);
  } catch (err) {
    recipeView.renderError();  // User feedback
  }
};
```

🧪 Quick Reference Cheatsheet

```
// Config
export const API_URL = 'https://...';

// State object
export const state = { data: {}, ui: {} };

// View class
class MyView extends View {
  _parentElement = document.querySelector('.container');
```

```
  _generateMarkup() { return `<div>${this._data.title}</div>`; }
  addHandlerClick(handler) {
    this._parentElement.addEventListener('click', handler);
  }
}
export default new MyView();

// Controller
const controlAction = async function() {
  try {
    view.renderSpinner();
    await model.loadData();
    view.render(model.state.data);
  } catch (err) {
    view.renderError();
  }
};

// Init
const init = function() {
  view.addHandlerClick(controlAction);
};
init();
```

✅ Exam Tips

1. **MVC separates concerns**: Model (data), View (UI), Controller (logic)
2. **Config module** centralizes constants
3. **Helper module** contains reusable utilities
4. **State is single source of truth** in Model
5. Use **class inheritance** for shared View logic
6. **Publisher-Subscriber** prevents circular dependencies
7. Views are **exported as instances** (singleton pattern)
8. Use `createContextualFragment` for virtual DOM diffing
9. **Re-throw errors** to propagate up the chain
10. `...(cond && { prop })` conditionally adds object properties
11. `window.location.hash.slice(1)` gets URL hash without `#`
12. `history.pushState` changes URL without page reload