The MVC architecture pattern turns complex application development into a much more manageable process. It allows several developers to simultaneously work on the application.

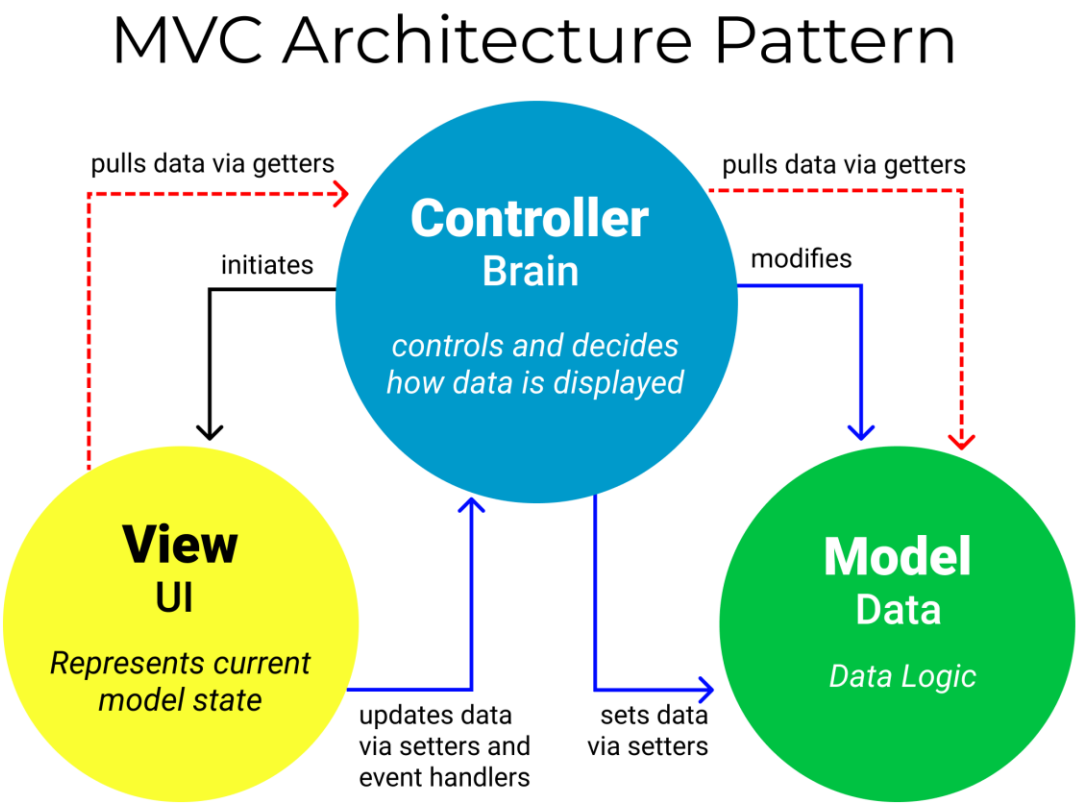
When I first learned about MVC patterns, I was intimidated by all the jargon. And even more so when I started applying these concepts to an actual application.

By taking a step back to focus on what MVC is and what it can accomplish, it's much easier to understand and apply the pattern to any web application.

**What is MVC?**

MVC stands for model-view-controller. Here's what each of those components mean:

- **Model**: The backend that contains all the data logic
- **View**: The frontend or graphical user interface (GUI)
- **Controller**: The brains of the application that controls how data is displayed




The concept of MVCs was first introduced by Trygve Reenskaug, who proposed it as a way to develop desktop application GUIs.

Today the MVC pattern is used for modern web applications because it allows the application to be scalable, maintainable, and easy to expand.

**Why Should You Use MVC?**

Three words: **separation of concerns**, or SoC for short.

The MVC pattern helps you break up the frontend and backend code into separate components. This way, it's much easier to manage and make changes to either side without them interfering with each other.

But this is easier said than done, especially when several developers need to update, modify, or debug a full-blown application simultaneously.

**How to Use MVC**

To better illustrate the MVC pattern, I've included a web application that shows how these concepts all work.

My Car Clicker application is a variation of a well-known Cat Clicker app.

Here are some of the major differences in my app:

1. No cats, **only** muscle cars images (sorry cat lovers!)
2. Multiple car models are listed
3. There are multiple click counters
4. It only displays the selected car

Now let's dive into these three components that make up the MVC architecture pattern.

**Model (data)**

The model's job is to simply manage the data. Whether the data is from a database, API, or a JSON object, the model is responsible for managing it.

Example:

In the Car Clicker application, the model object contains an array of car objects with all the information (data) needed for the app.

It also manages the current car being displayed with a variable that's initially set to null.

```
const model = {
  currentCar: null,
  cars: [
    {
      clickCount: 0,
      name: 'Coupe Maserati',
      imgSrc: 'img/black-convertible-coupe.jpg',
    },
    {
```

```
      clickCount: 0,
      name: 'Camaro SS 1LE',
      imgSrc: 'img/chevrolet-camaro.jpg',
    },
    {
      clickCount: 0,
      name: 'Dodger Charger 1970',
      imgSrc: 'img/dodge-charger.jpg',
    },
    {
      clickCount: 0,
      name: 'Ford Mustang 1966',
      imgSrc: 'img/ford-mustang.jpg',
    },
    {
      clickCount: 0,
      name: '190 SL Roadster 1962',
      imgSrc: 'img/mercedes-benz.jpg',
    },
  ],
};
```

**Views (UI)**

The view's job is to decide what the user will see on their screen, and how.

The Car Clicker app has two views: carListView and CarView.

Both views have two critical functions that define what each view wants to initialize and render.

These functions are where the app decides what the user will see and how.

**carListView**

```
const carListView = {
  init() {
    // store the DOM element for easy access later
    this.carListElem = document.getElementById('car-list');
```

```
    // render this view (update the DOM elements with the right values)
    this.render();
  },

  render() {
    let car;
    let elem;
    let i;
    // get the cars to be render from the controller
    const cars = controller.getCars();

    // to make sure the list is empty before rendering
    this.carListElem.innerHTML = '';

    // loop over the cars array
    for(let i = 0; i < cars.length; i++) {
      // this is the car we've currently looping over
      car = cars[i];

      // make a new car list item and set its text
      elem = document.createElement('li');
      elem.className = 'list-group-item d-flex justify-content-between lh-condensed';
      elem.style.cursor = 'pointer';
      elem.textContent = car.name;
      elem.addEventListener(
        'click',
        (function(carCopy) {
          return function() {
            controller.setCurrentCar(carCopy);
            carView.render();
```

```
            };
          })(car)
        );
        // finally, add the element to the list
        this.carListElem.appendChild(elem);
      }
    },
};
```

**CarView**

```
const carView = {
  init() {
    // store pointers to the DOM elements for easy access later
    this.carElem = document.getElementById('car');
    this.carNameElem = document.getElementById('car-name');
    this.carImageElem = document.getElementById('car-img');
    this.countElem = document.getElementById('car-count');
    this.elCount = document.getElementById('elCount');



    // on click, increment the current car's counter
    this.carImageElem.addEventListener('click', this.handleClick);

    // render this view (update the DOM elements with the right values)
    this.render();
  },

  handleClick() {
    return controller.incrementCounter();
  },

  render() {
```

```
    // update the DOM elements with values from the current car
    const currentCar = controller.getCurrentCar();
    this.countElem.textContent = currentCar.clickCount;
    this.carNameElem.textContent = currentCar.name;
    this.carImageElem.src = currentCar.imgSrc;
    this.carImageElem.style.cursor = 'pointer';
  },
};
```

**Controller (Brain)**

The controller's responsibility is to pull, modify, and provide data to the user. Essentially, the controller is the link between the view and model.

Through getter and setter functions, the controller pulls data from the model and initializes the views.

If there are any updates from the views, it modifies the data with a setter function.

```
const controller = {
  init() {
    // set the current car to the first one in the list
    model.currentCar = model.cars[0];

    // tell the views to initialize
    carListView.init();
    carView.init();
  },

  getCurrentCar() {
    return model.currentCar;
  },

  getCars() {
    return model.cars;
  },
```

```
  // set the currently selected car to the object that's passed in
  setCurrentCar(car) {
    model.currentCar = car;
  },

  // increment the counter for the currently-selected car
  incrementCounter() {
    model.currentCar.clickCount++;
    carView.render();
  },
};

// Let's goooo!
controller.init();
```

**MVC Frameworks**

JavaScript has grown in popularity, and it's taken over the backend in recent years. More and more full-blown JavaScript applications have opted for the MVC architecture pattern in one way or another.

Frameworks come and go, but what has been constant are the concepts borrowed from the MVC architecture pattern.

Some of the early frameworks that applied these concepts were **KnockoutJS**, **Django**, and **Ruby on Rails.**

**Conclusion**

The most attractive concept of the MVC pattern is separation of concerns.

Modern web applications are very complex, and making a change can sometimes be a big headache.

Managing the frontend and backend in smaller, separate components allows for the application to be scalable, maintainable, and easy to expand