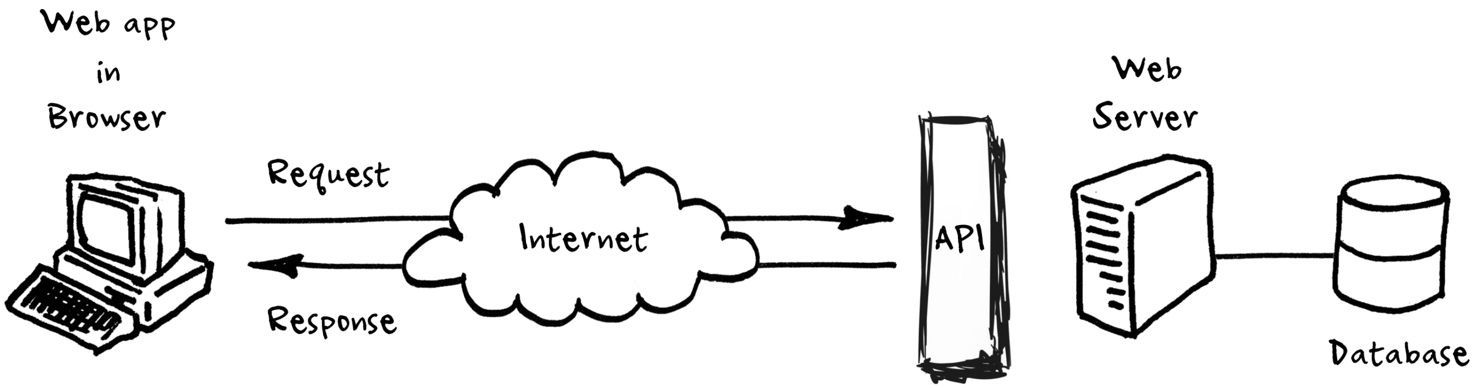
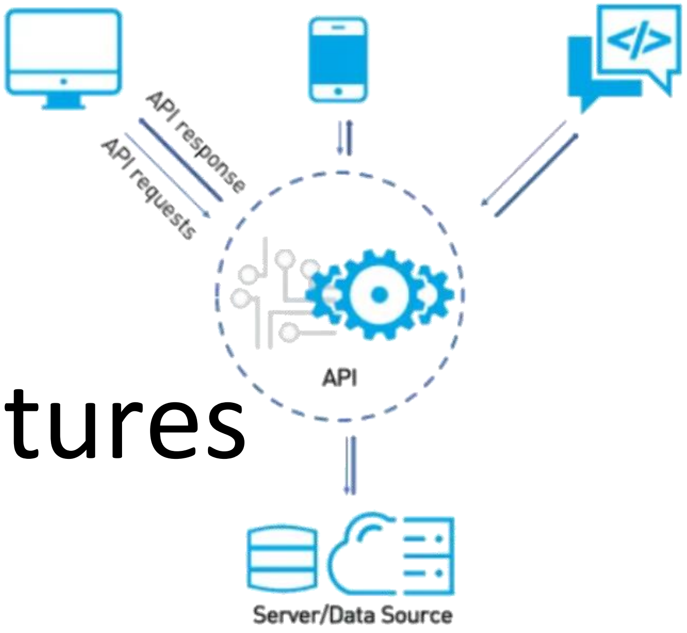
Web App Architectures

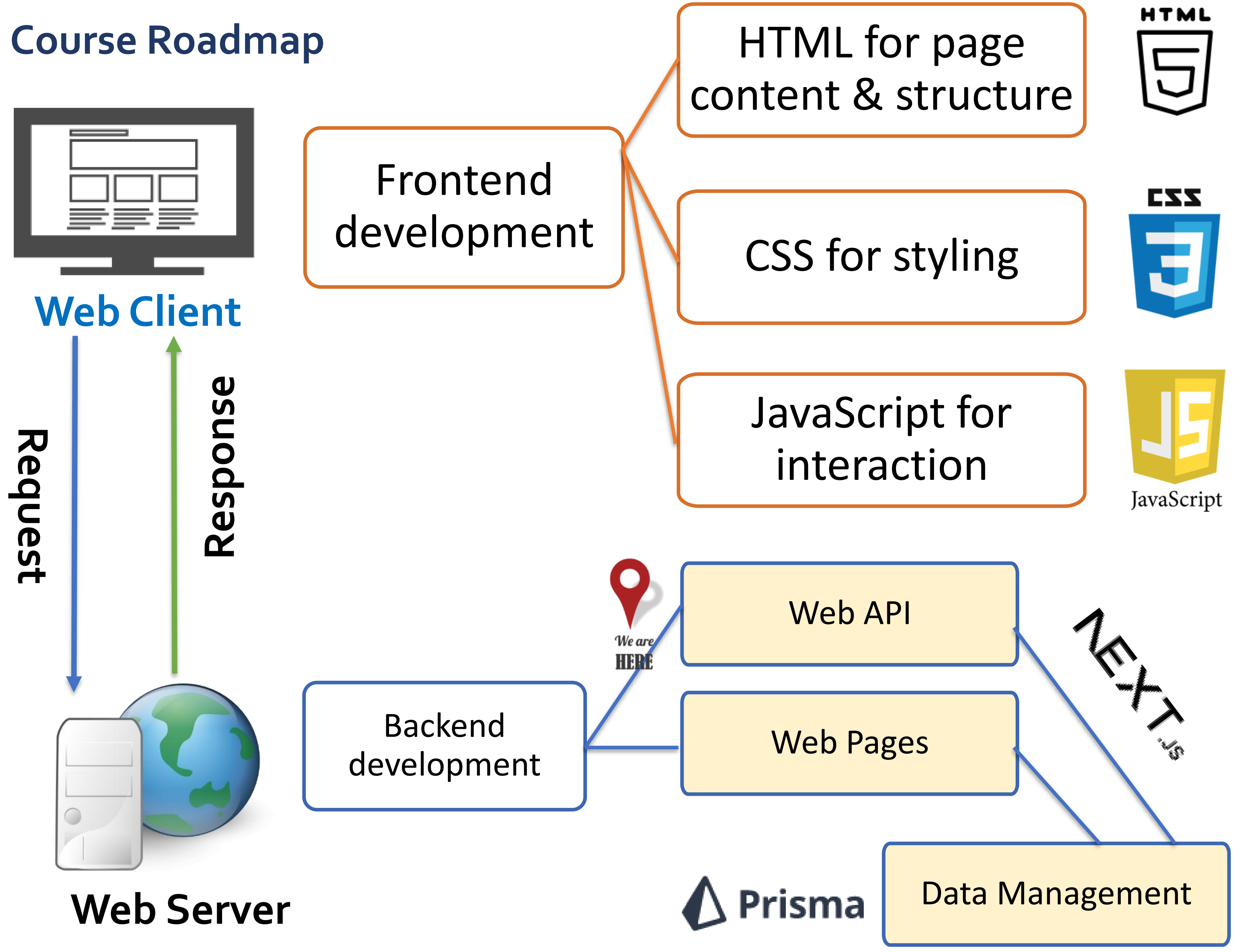


Outline

1. Web App Architectures



Course Roadmap



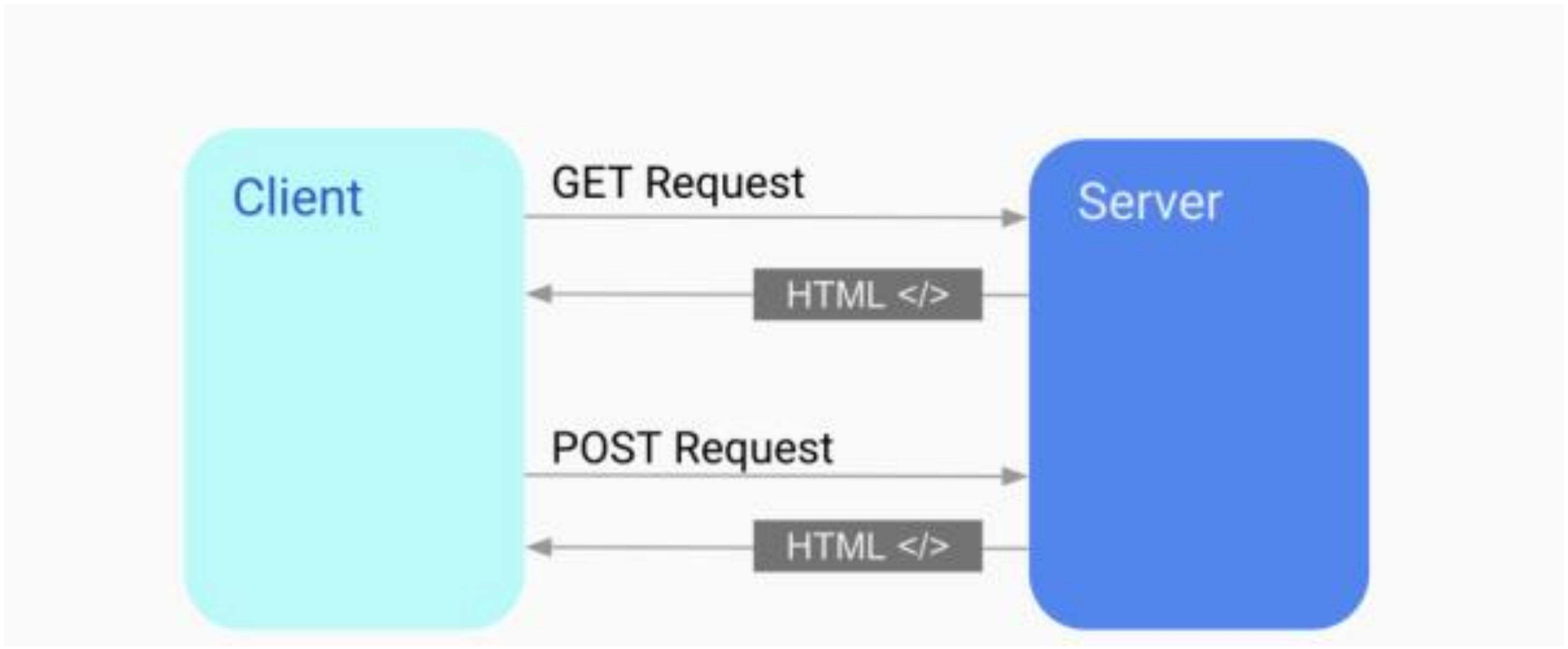
Software Architecture

- Architecture = Components + Connectors
- Architectural **Patterns**
- Enable to achieve the desired **non-functional** requirements (scalability, performance, extensivity, ease maintenance)
- Web software architecture defines how to organize app data, UI, and logic to ensure scalability, performance, and maintainability

Multi-Page Application (MPA)

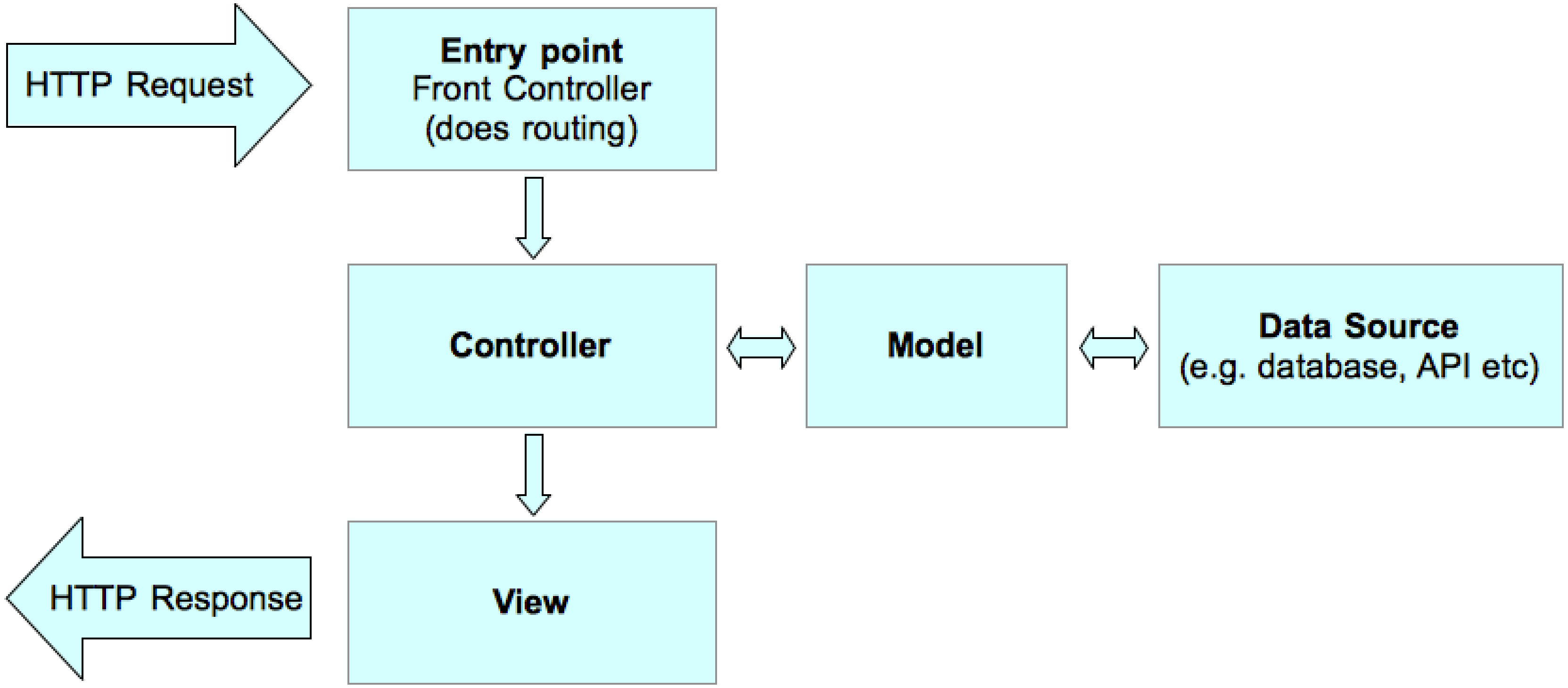
- **Concept**: Each interaction loads a new page from the server (traditional web model)
- **Key Technologies**: PHP, Rails, Django, Next.js (SSR)
- **Use Cases**: Blogs, e-commerce sites, SEO-heavy apps

Multi-Page Application (MPA)



1

MVC-based Web App

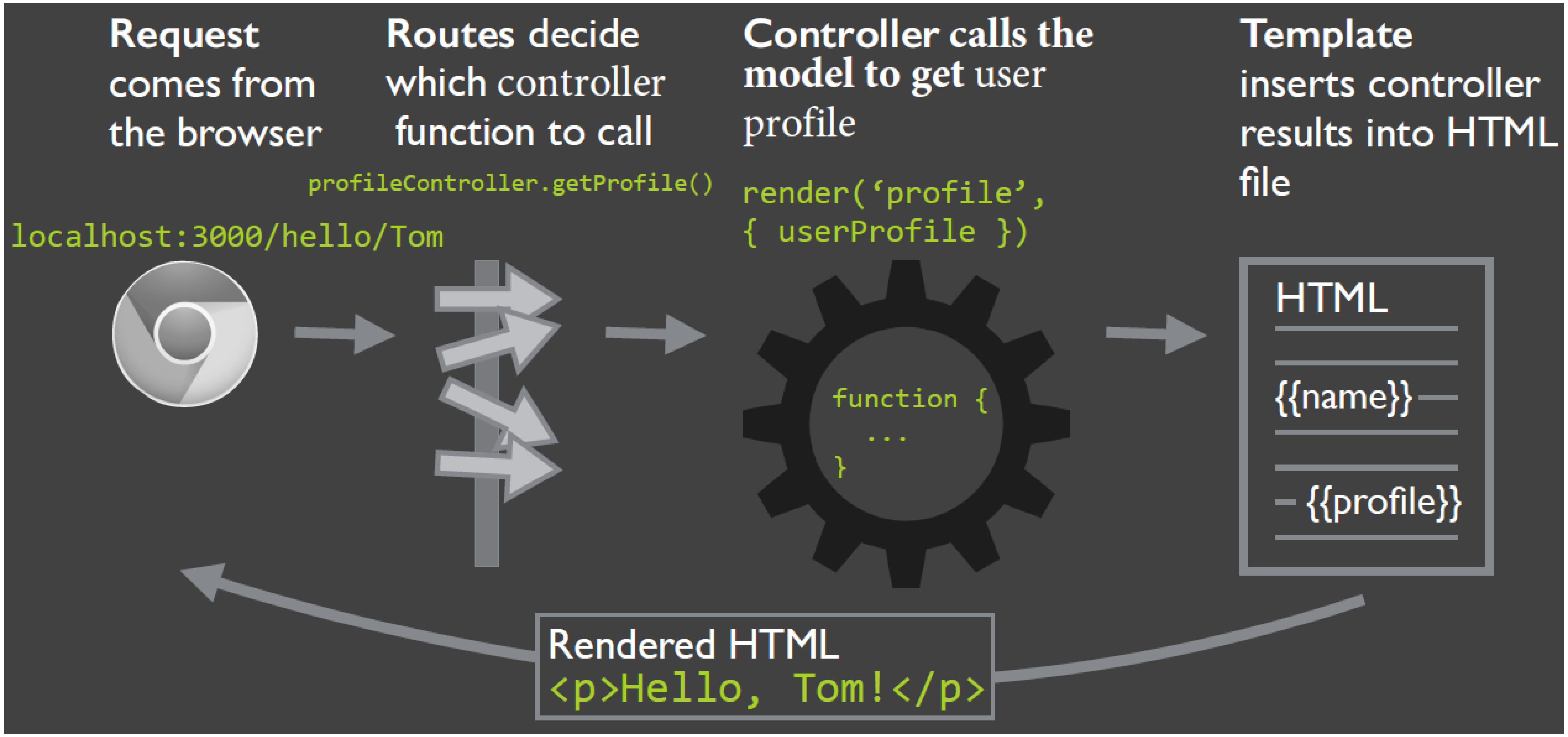


Back

MVC

- **Concept**: Separates concerns into three layers: Model (Data & Business Logic),View (User Interface),Controller (Handles User Requests)
- **Key Technologies**: Express.js, Ruby on Rails, Laravel, Django
- **Use Cases**: Traditional web apps, CRUD-based systems

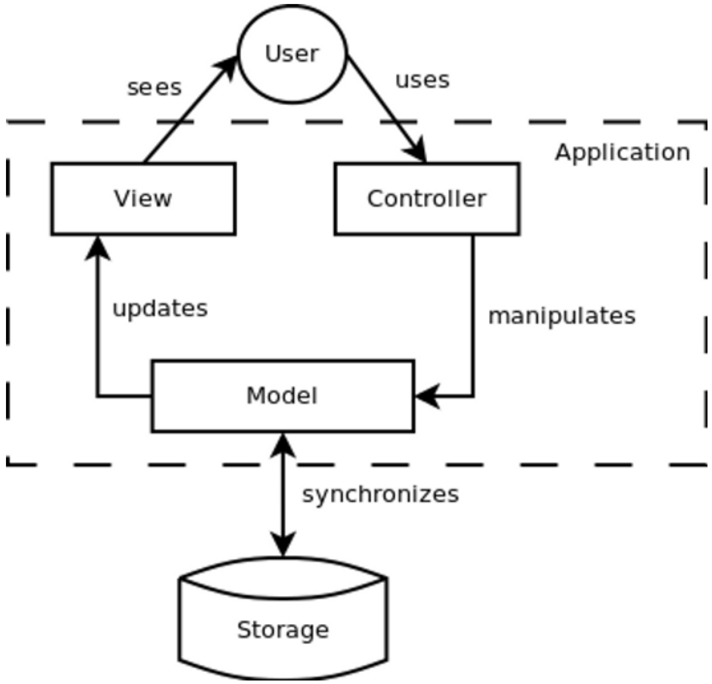
Interaction between App Modules


Request comes from the browser
Routes decide which controller function to call
Controller calls the model to get user profile
Template inserts controller results into HTML file
profileController.getProfile()
localhost:3000/hello/Tom
render('profile', { userProfile })
function { ... }
HTML
{{name}}
{{profile}}
Rendered HTML
<p>Hello, Tom!</p>

MVC-based Web application

Controller

- accepts incoming requests and user input and **coordinates** request handling
- instructs the model to perform actions based on that input
  - e.g. add an item to the user's shopping cart
- decides what view to display for output



Model : implements business logic and computation, and manages application's data

View : responsible for

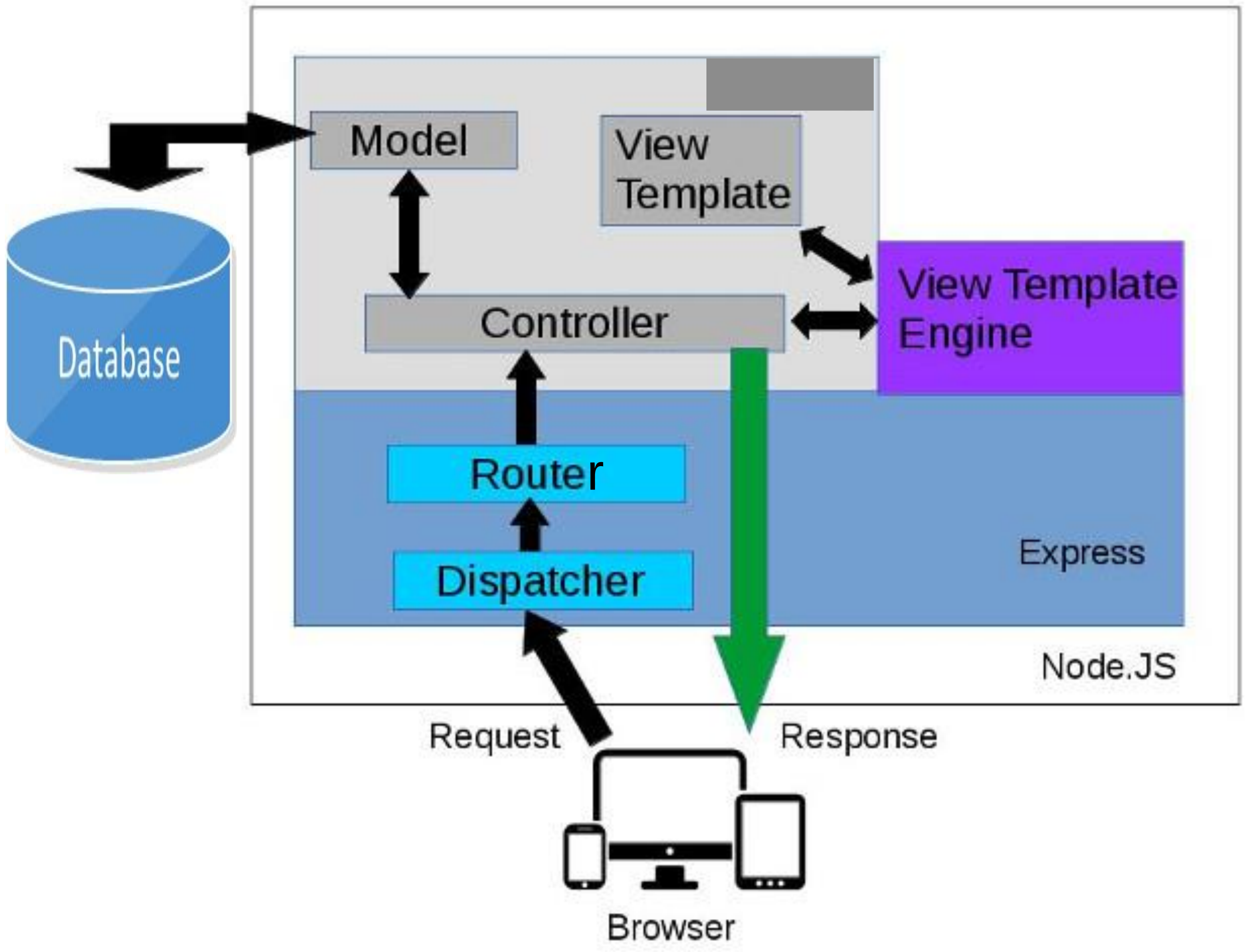
- collecting input from the user
- displaying output to the user

Advantages of MVC

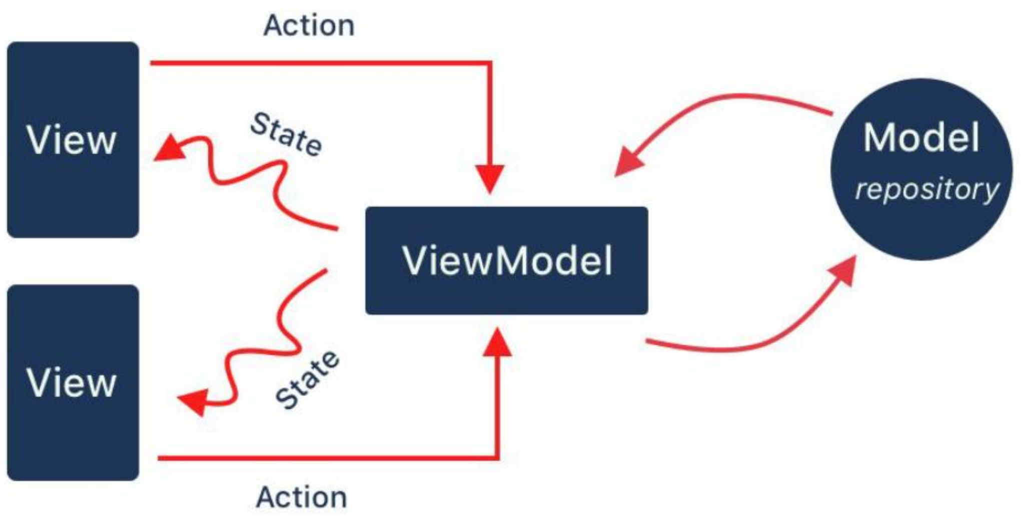
- ***Separation of concerns***
  - Views, controller, and model are separate components. This allows modification and change in each component without significantly disturbing the others.
    - Computation is not intermixed with Presentation. Consequently, code is cleaner and easier to understand and change.
- **Flexibility**
  - The view component, which often needs changes and updates to keep the users continued interests, is separate
    - The UI can be completely changed without touching the model in any way
- **Reusability**
  - The same model can used by different views (e.g., Web view and Mobile view)
- **Allows for parallel teamwork**, e.g., a UI designer can work on the View while a software engineer works on the Controller and Model

MVC is widely used and recommended particularly for interactive web-applications

MVC using Node.js Express



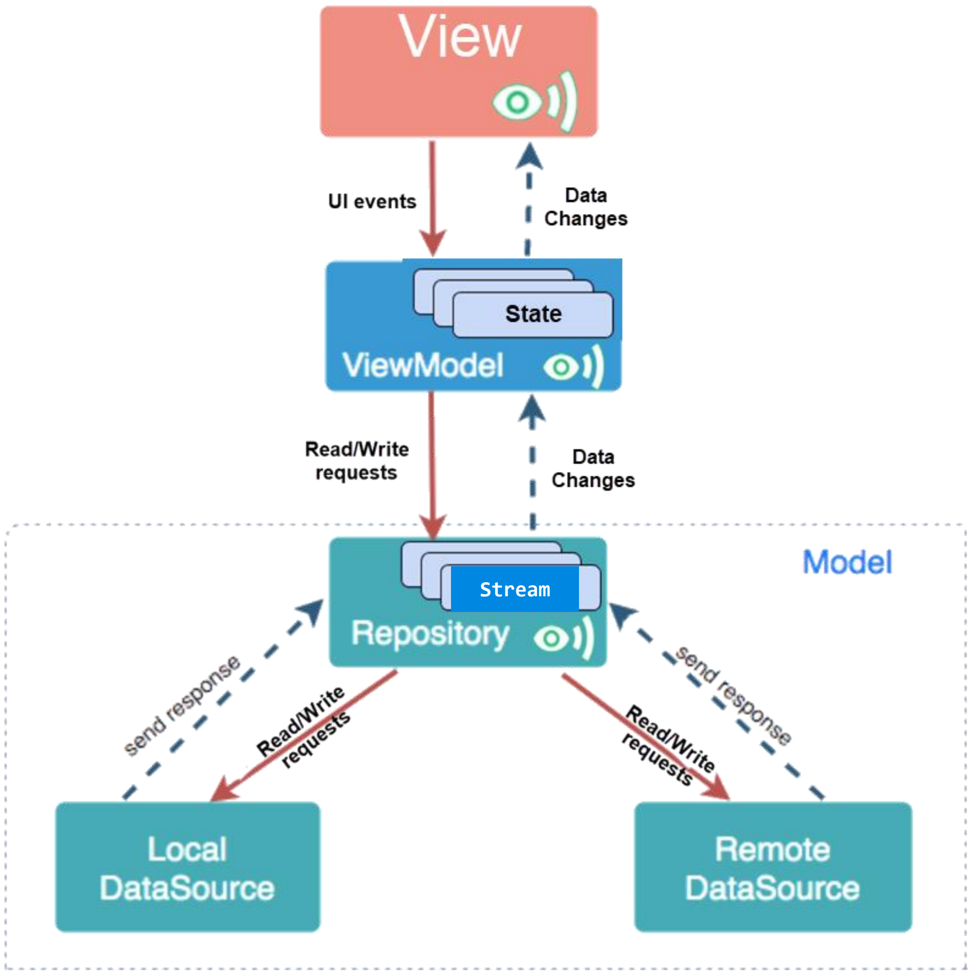
MVVM Architecture



Model-View-ViewModel (MVVM)

- **Concept**: Similar to MVC, but introduces a ViewModel to handle UI logic separately
- **Key Technologies**: React (Hooks), Vue.js (Composition API)
- **Use Cases**: Scalable frontend-heavy applications, state-driven UIs

Model-View-ViewModel (MVVM) Architecture



View = UI to display state & collect user input

- It **observes** state changes from the ViewModel to update the UI accordingly

- Calls the ViewModel to handle events such as button clicks, form input, etc.

ViewModel

- Manages **state** (i.e., data needed by the UI)
  - Interacts with the Model to read/write data based on user input
  - Expose the state as **Observables** that the UI can subscribe-to to get data changes
- Implements UI logic / computation (e.g., Filtering or Sorting Data, Validate user input, check correct email format or check both the password and confirm password fields match)

Model - handles data operations

- Model has **entities** that represent app data
- **Repositories** read/write data from either a Local Database or a Remote Web API
- Implements data-related logic / computation

MVVM Key Principles

- Separation of concerns:
  - View, ViewModel, and Model are **separate components** with distinct roles
- Loose coupling:
  - ViewModel **has no direct reference to the View**
  - View never accesses the model directly
  - Model unaware of the view
- Observer pattern:
  - View observes the ViewModel (to get data changes)
  - ViewModel observes the Model (to get data changes)

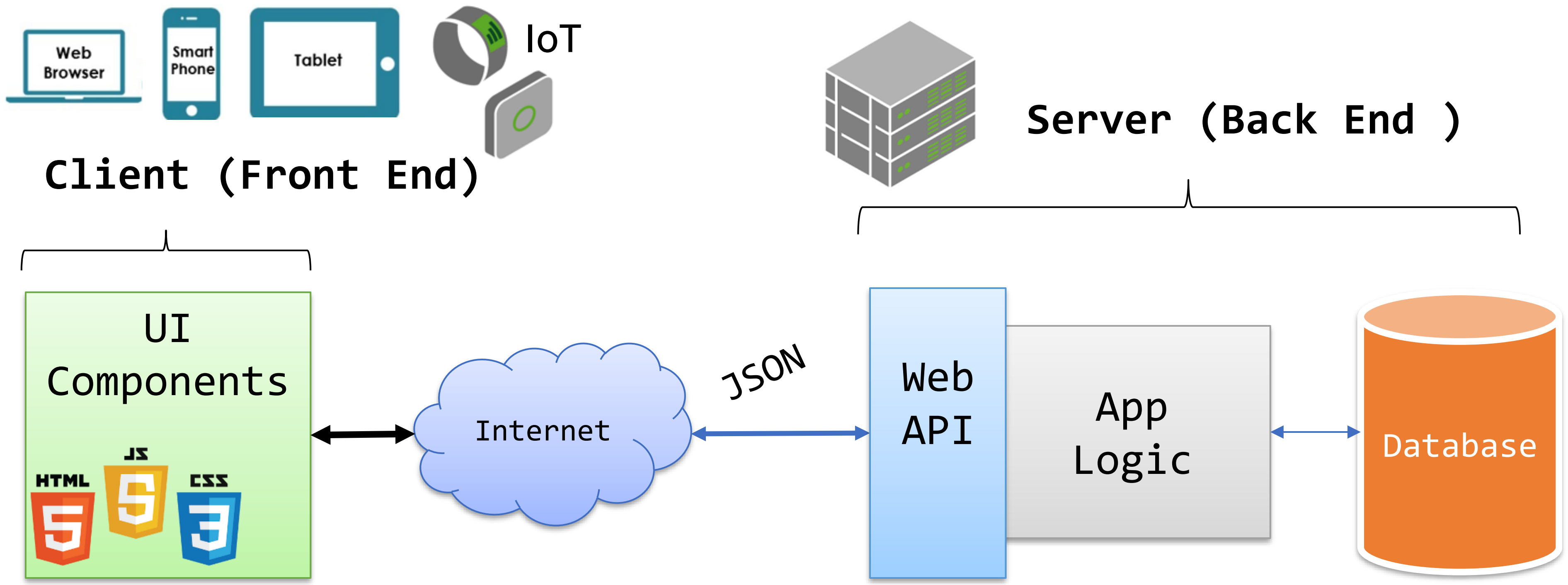
Advantages of MVVM

- ***Separation of concerns* = separate UI from app logic**
  - App logic is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change
  - Allow changing a component without significantly disturbing the others (e.g., View can be completely changed without touching the model)
  - Easier **testing** of the App components

MVVM => Easily maintainable and testable app

Single Page Application (SPA)

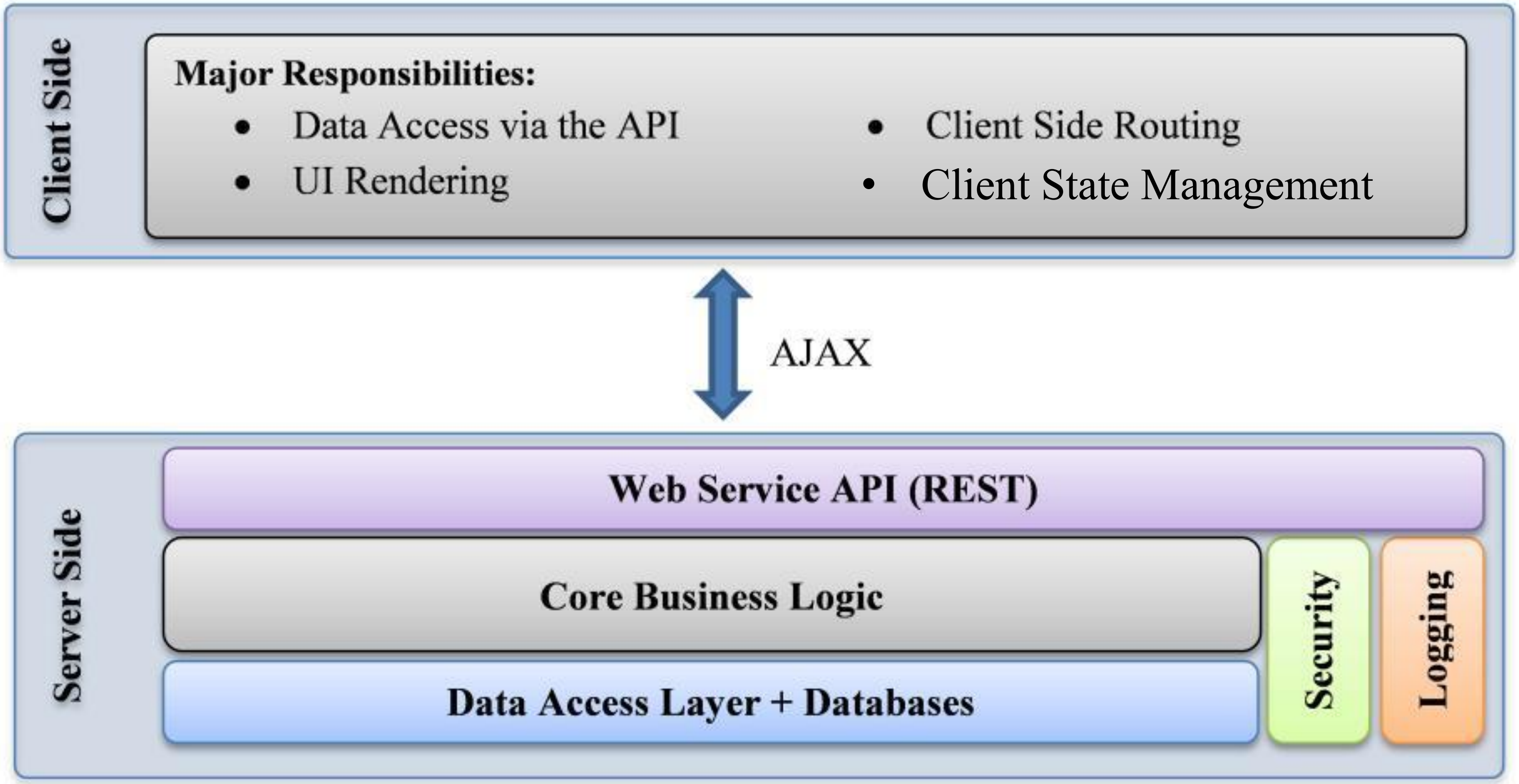
- Front-end made-up of **multiple UI components loaded** in response to user actions
- Back-end Web API



Single Page Application (SPA)

- **Concept**: Loads a single HTML page and dynamically updates content without full-page reloads
    - Uses client-side rendering (CSR)
- **Key Technologies**: React, Next.js (Client Components), Vue.js, Angular
- **Use Cases**: Dashboards, social media apps, real-time applications
    - Mobile & web apps with different UI needs

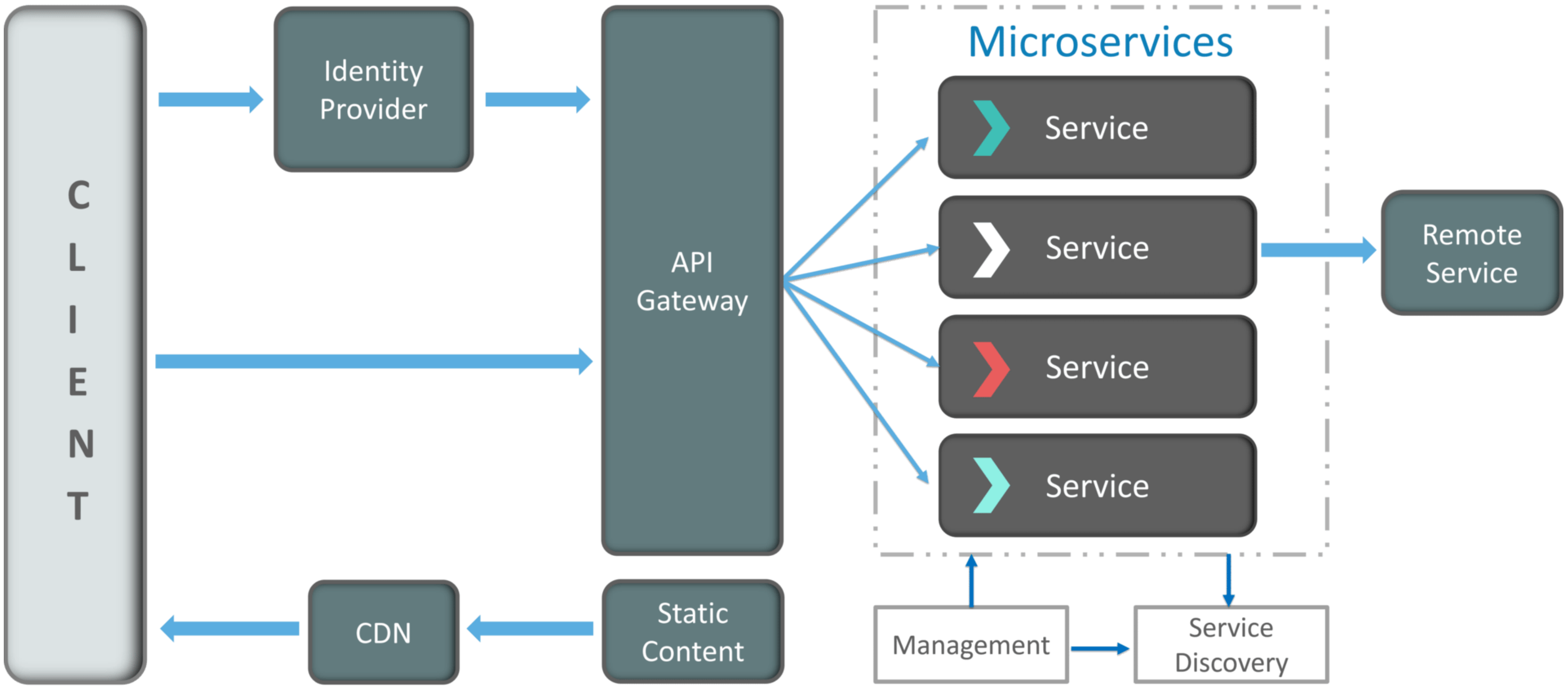
Role of Client and Server in SPA



Microservices Architecture

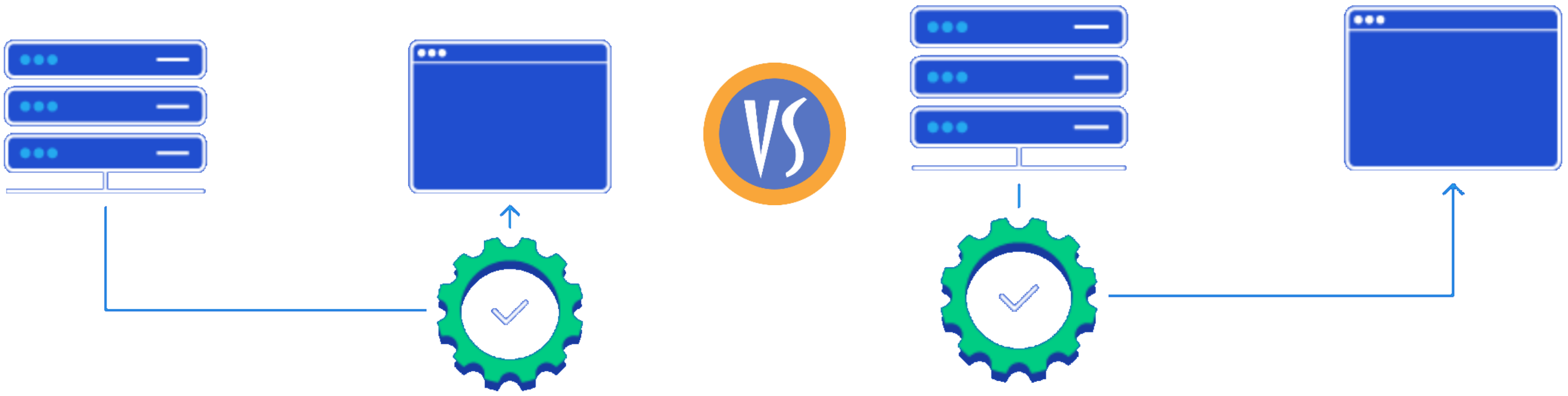
- **Concept**: Divides an application into small, independent services that communicate via APIs
- **Key Technologies**: Docker, Kubernetes, gRPC, REST APIs
- **Use Cases**: Large-scale apps (e.g., Netflix, Uber).

Microservices Architecture



Source: https://www.edureka.co/blog/microservice-architecture/

Client-side vs. Server-side Rendering of Views



< Back

Client-side rendering (CSR)?

1. The user sends a request to a website (usually via a browser)
2. The browser downloads the HTML (containing HTML templates and static content), CSS and JS
3. Client-side JS makes Web API requests via AJAX to fetch dynamic data from the server
4. After the server responds, Client-side JS renders the Html template using the received data (The data from the API fill the template placeholders) then updates the page using DOM processing on the client browser

CSR using Handlebars

- Add Handlebars script

```
<script src="path/to/handlebars.js"></script>
```

- Create a template

```
const studentTemplate = '<p>{{firstname}} {{lastname}}</p>'
```

- Render the template

```
const student = {id: '…', firstname: '…', lastname: '…'};
const htmlTemplate = Handlebars.compile(studentTemplate);
studentDetails.innerHTML = htmlTemplate(student);
```

Server-side rendering (SSR)?

Concept: Pages are generated on the server per request, improving SEO and initial load speed:

1. The user sends a request to a website (usually via a browser)
2. The server (more precisely a **model** object) performs the necessary computation to get/compute the results data
3. The server renders the Html template using the produced data
4. The produced Html content is sent to the client's browser

- **Key Technologies**: Next.js (Server Components), Nuxt.js, SvelteKit
- **Use Cases**: Content-heavy sites, blogs, e-commerce

Server-Side Rendering (1 of 2)

1. Configure the View Engine

```
import handlebars    from 'express-handlebars';
const app            = express();
/* Configure handlebars:
 set extension to .hbs so handlebars knows what to look for
 set the defaultLayout to 'main'
 the main.hbs defines common page elements such as the menu
 and imports all the common css and javascript files
*/
app.engine('hbs', handlebars({ defaultLayout: 'main',
  extname: '.hbs'}));

// Register handlebars as our view engine as the view engine
app.set('view engine', 'hbs');

//Set the location of the view templates
app.set('views', `${currentPath}/views`);
```

Server-Side Rendering (2 of 2)

2. Call **res.render** method to perform server-side rendering and return the generated html to the client

```
app.get('/cart', (req, res) => {
    const shoppingCart = shoppingRepository.getShoppingCart();
    res.render('shopCart', { shoppingCart });
});
```

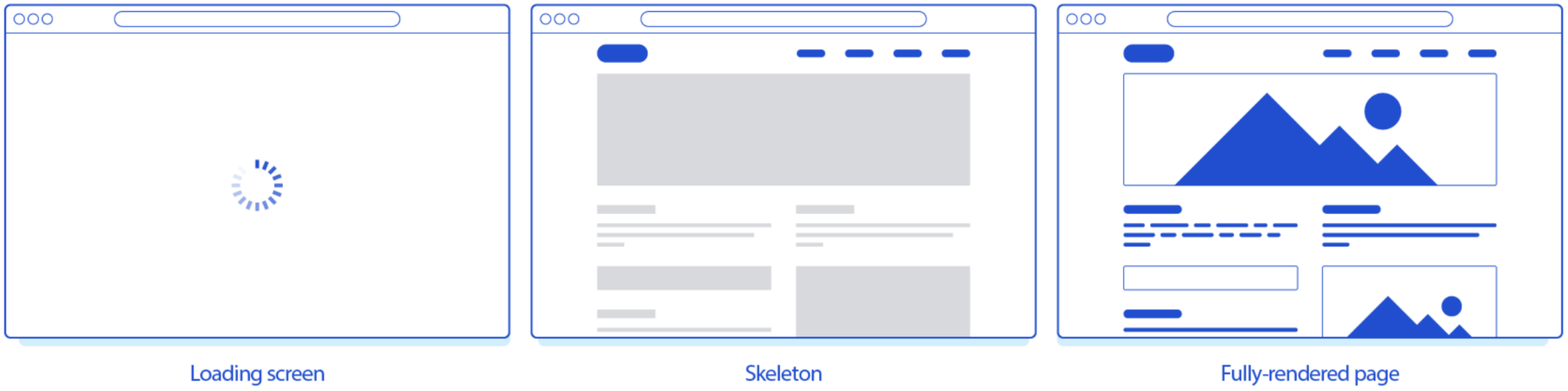
The above example passes the shopping cart to the **'shopCart'** template to generate the html to be returned to the browser

Client-side vs. Server-side Rendering of Views

- 👍 CSR **frees the server** from the rendering burden and enhances the app **scalability** (increased server ability to handle more concurrent requests)
  - 👎 But one of the main disadvantages is slower initial loading speed **of the first page** as the client receive a lot of JavaScript files to handle views rendering
  - 👍 Fast rendering after initial loading: second and further page load time is lesser, since all the supporting scripts are loaded in advance for CSR

- 👍 SSR **reduces** the amount of client-side JavaScript and speed-up the initial page loads particularly for slow clients
  - 👍 Web servers (having higher compute power) may render the page faster than a client-side rendering. As a result, the initial loading is quicker.
  - 👎 But this puts the rending burden on the server
  - 👎 Does **full page reload** to update the page

Client-side Rendering slow initial page load



Choosing the Right Architecture

Architecture	**Best For**
SPA	Interactive UIs, dashboards
MPA	Traditional websites, SEO-heavy apps
MVC	CRUD apps, monolithic web apps
MVVM	Scalable UIs, frontend-heavy apps
Microservices	Large, scalable applications