

# Full Stack Application Development

Application Development- Introduction

Akshaya Ganesan



# Module 1.2 Web Application Architecture



# Activity



- Web site: <https://whatismyipaddress.com/>
- API : <https://api.ipify.org/?format=json>



# Layered Pattern

- Context: Separation of concern
- Problem: Modules of the system may be independently developed and maintained, supporting portability, modifiability, and reuse.
- **Solution:** The layered pattern divides the software into units called layers.
- Each layer is a grouping of modules that offers a cohesive set of services.
- Each partition is exposed through a interface.
- Layers interact according to a strict ordering relation and unidirectional.



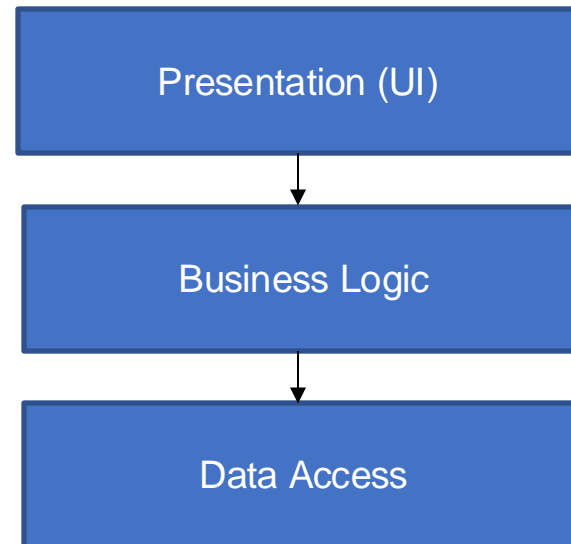
# Web Application

- Web application follows the Layered Architecture.
- Two-layer systems
  - Typical client-server system
  - The client held the user interface and other application code, and the server was usually a relational database.
  - Embed the logic directly into the UI screens
  - Alternative: put the domain logic in the database as stored procedures



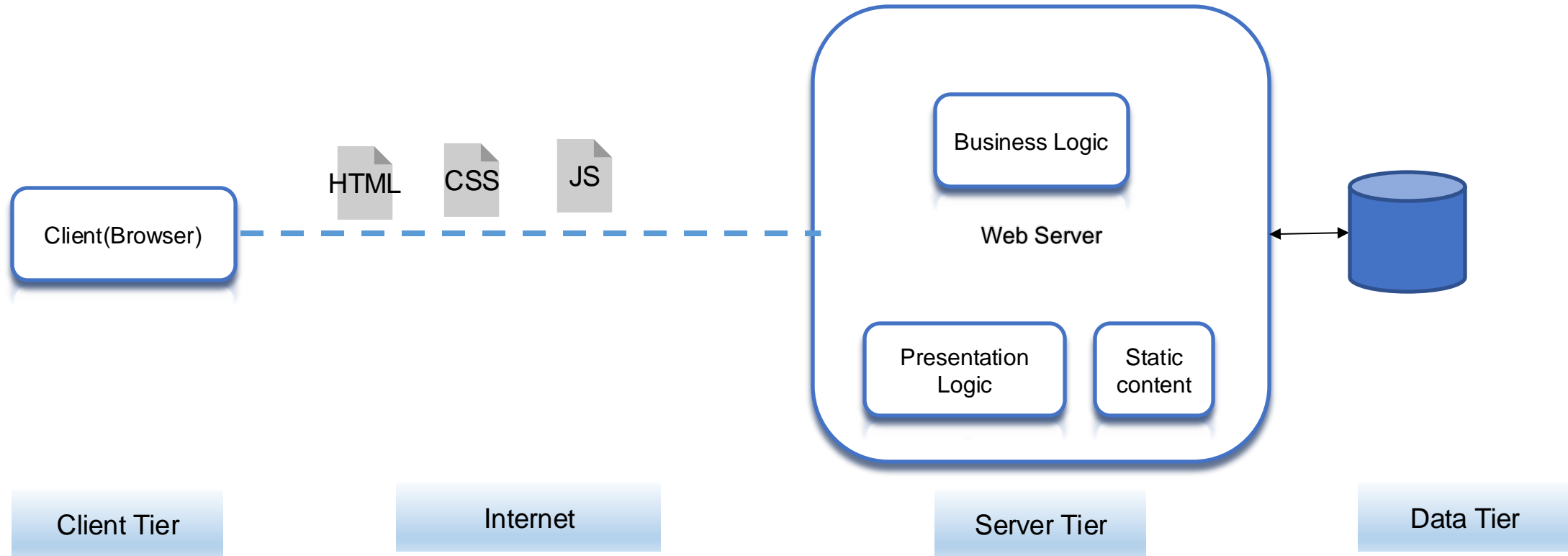
# 3-Tier Architecture

- A typical Web Application:



# Traditional Web Application

- Three Tier





# Ensuring a clear separation of concern

- **Scenario 1:** A list of products in which all the products that sold over 10 percent more than they did the previous month were colored in red.
- **Method 1:** (putting domain logic into the presentation)
- Developers placed logic in the presentation layer that compared this month's sales to last month's sales, and if the difference was more than 10 percent, they set the color to red
- **Method 2:**
- To properly separate the layers, you need a method in the domain layer to indicate if a product has improved sales. This method makes the comparison between the two months and returns a Boolean value.
- The presentation layer then calls this Boolean method and, if true, highlights the product in red.

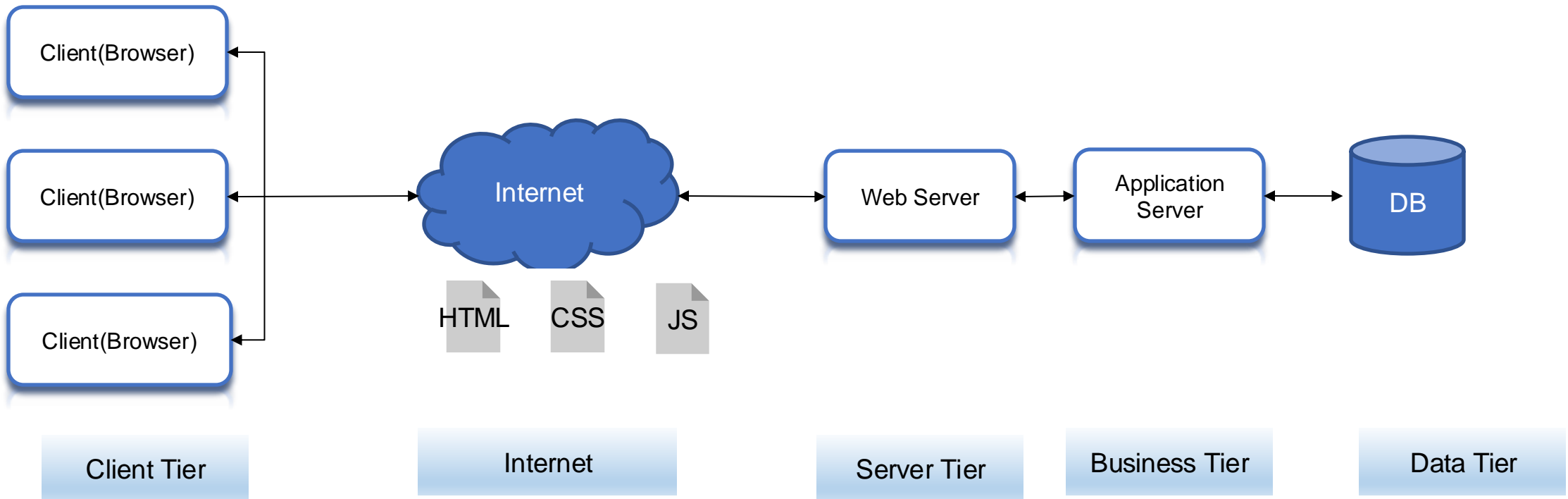




# Ensuring a clear separation of concern

- **Scenario 2:** You need to add different layers to an application, such as a command-line interface to a Web application.
- Method: If there's any functionality you have to duplicate to do this, that's a sign of where domain logic has leaked into the presentation
- Similarly, do you have to duplicate logic to replace a relational database with an XML file?

# N-Tier Web Application

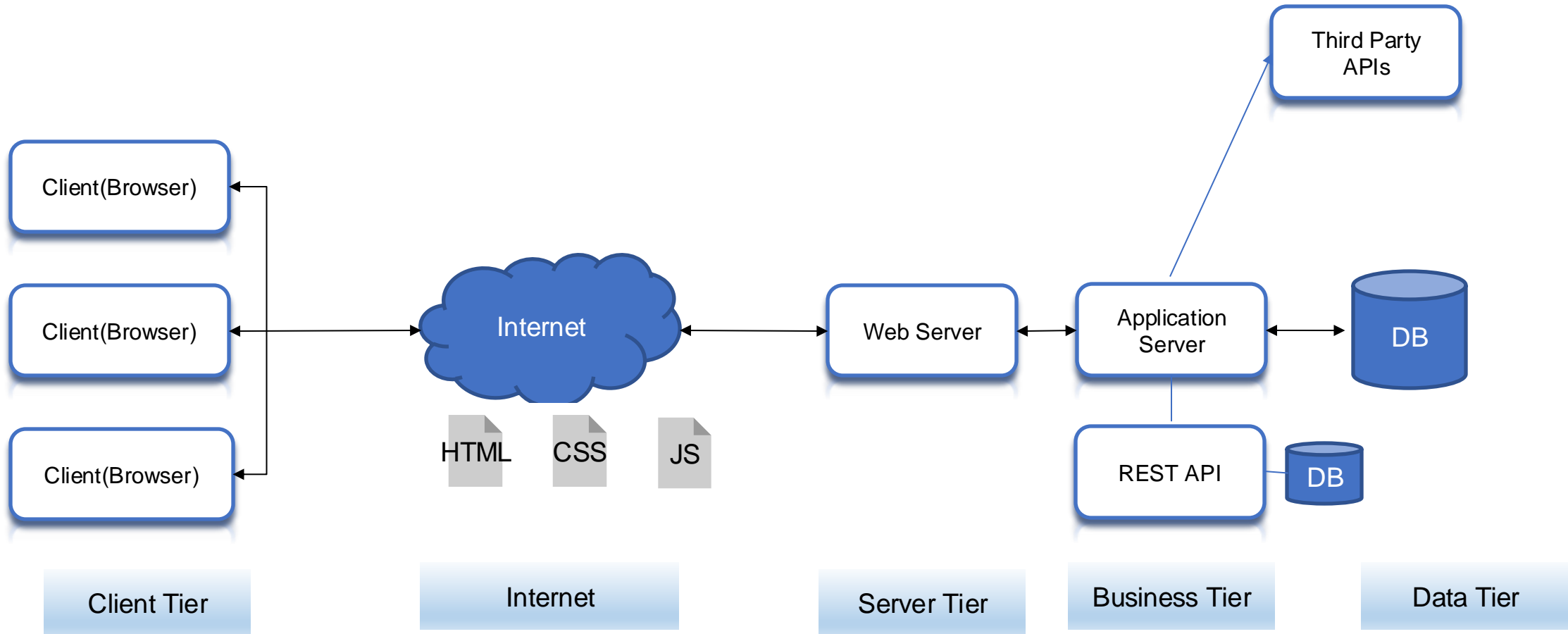




# Characteristics

- Each tier is completely independent.
- The  $n$ th tier only has to know how to handle a request from the  $n+1$ th tier
- Tiers make it easier to ensure security and to optimize performance and availability in specialized ways.

# Service Based Web Application





# Monolithic Application

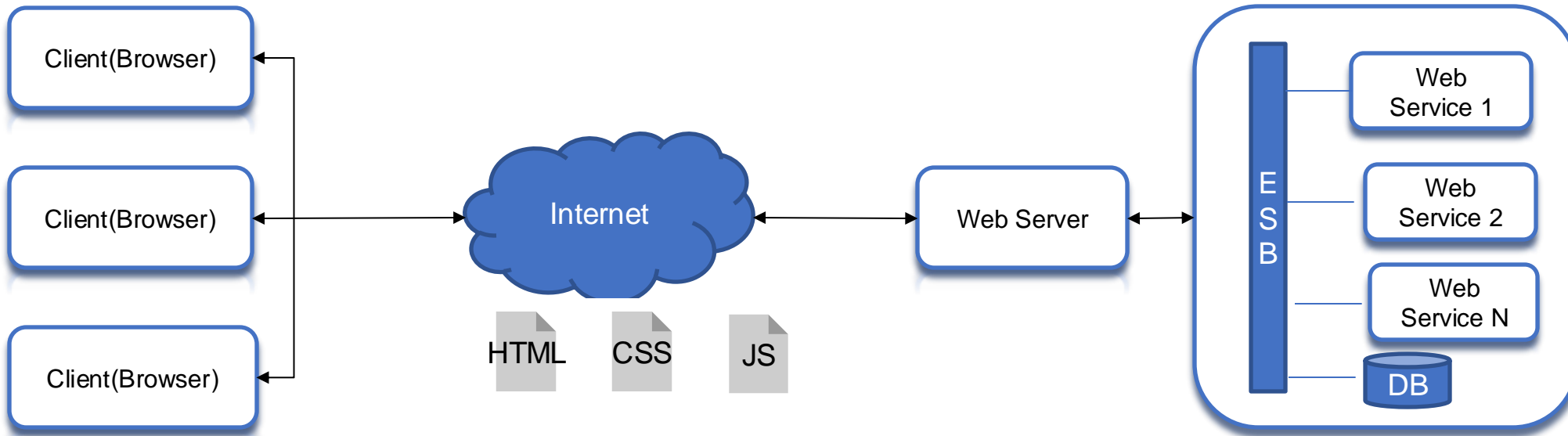
- The application is deployed as a single monolithic application.
- For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat
- Difficulty to scale
- Redeployment of the entire application in case of change



# Service Oriented Architecture

- Service-oriented architecture (SOA) is a business-centric IT architectural approach that supports integrating your business as linked, repeatable business tasks or services.
- SOA exposes business functionalities as services to be consumed by applications.
- Services are
  - loosely coupled
  - Autonomous
- **Robert C. Martin's Single Responsibility Principle**
- Gather together the things that change for the same reasons. Separate those things that change for different reasons.

# SOA Based Web Application



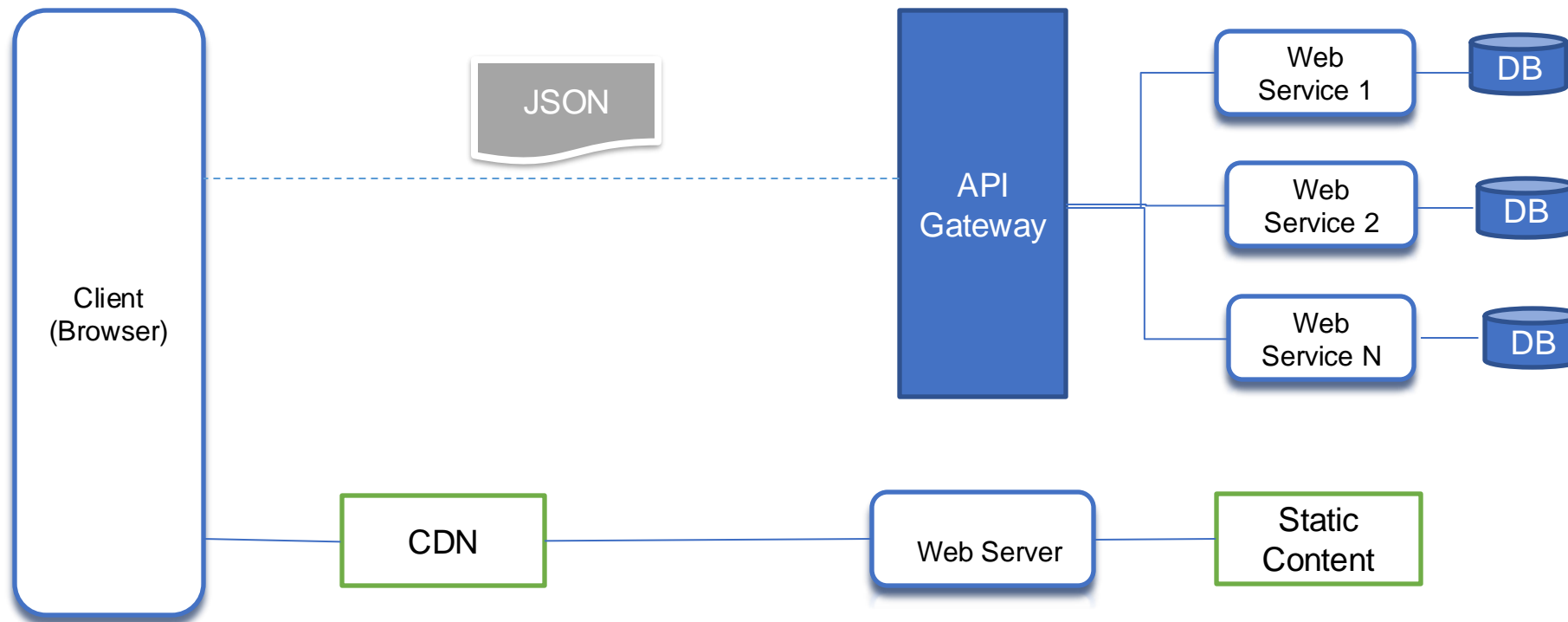


# Microservice Architecture

- Microservice Application
  - ✓ is composed of many small, independent services
  - ✓ each service implements a single business capability
  - ✓ are loosely coupled, communicating through API contracts
  - ✓ can be built by a small, focused development team
- More complex to build and manage
  - ✓ requires a mature development and DevOps culture
  - ✓ can lead to higher release velocity, and a more resilient architecture



# Microservices Application



# API



- API stands for 'Application Programming Interface'
- API: an interface to access whatever resource it points to: data, server software, or other applications
- In an internet-connected world, web and mobile applications are designed for humans to use
- While APIs are designed for other digital systems and applications to use



# What is a API?

- An API is a software intermediary that allows two applications to talk to each other
  - ✓ API is the messenger that delivers your request to the provider that you're requesting it from and then delivers the response back to you
- An API is independent of their respective implementations.
- Changing the underlying implementation can be done without affecting the users
- APIs make it possible to integrate different systems together
  - ✓ like Customer Relationship Management systems, databases, or even school learning management systems

# API



- In this metaphor, a customer is like a user, who tells the waiter what she wants
- The waiter is like an API,
  - ✓ receiving the customer's order
  - ✓ translating the order into easy-to-follow instructions that the kitchen then uses to fulfill that order—often following a specific set of codes
  - ✓ or input, that the kitchen easily recognizes
- The kitchen is like a server that creates the order in the manner the customer wants it, hopefully!
- When the food is ready, the waiter picks up the order and delivers it to the customer.
- Similarly, the API delivers the response.

# Types



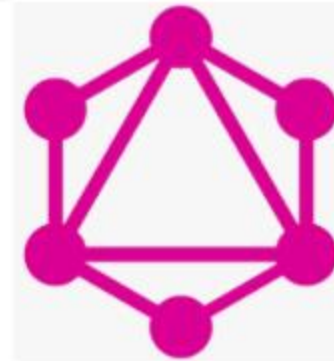
- Public API
  - Twitter API, Facebook API, Google Maps API, and more
  - Granting Outside Access to Your Assets
  - Provide a set of instructions and standards for accessing the information and services being shared
  - Making it possible for external developers to build an application around those assets
  - Much more restricted in the assets they share, given they're sharing them publicly with developers around the web
- Private API
  - Self-Service Developer & Partner Portal API
  - Far more common (and possibly even more beneficial, from a business standpoint)
  - Give developers an easy way to plug right into back-end systems, data, and software
  - Letting engineering teams do their jobs in less time, with fewer resources
  - All about productivity, partnerships, and facilitating service-oriented architectures



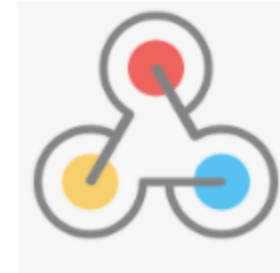
# API Paradigm

- Over the years, multiple API paradigms have emerged such as
  - ✓ REST
  - ✓ RPC
  - ✓ GraphQL
  - ✓ WebHooks
  - ✓ and WebSockets
- Broadly can be classified as
  - ✓ Request-Response APIs
  - ✓ Event Driven APIs

{ REST }



gRPC





# Request-Response APIs

- Typically expose an interface through a HTTP-based web server
- APIs define a set of endpoints
  - ✓ Clients make HTTP requests for data to those endpoints
  - ✓ Server returns responses
- The response is typically sent back as JSON or XML
- Three common paradigms used to expose request–response APIs:
  - ✓ REST
  - ✓ RPC
  - ✓ and GraphQL



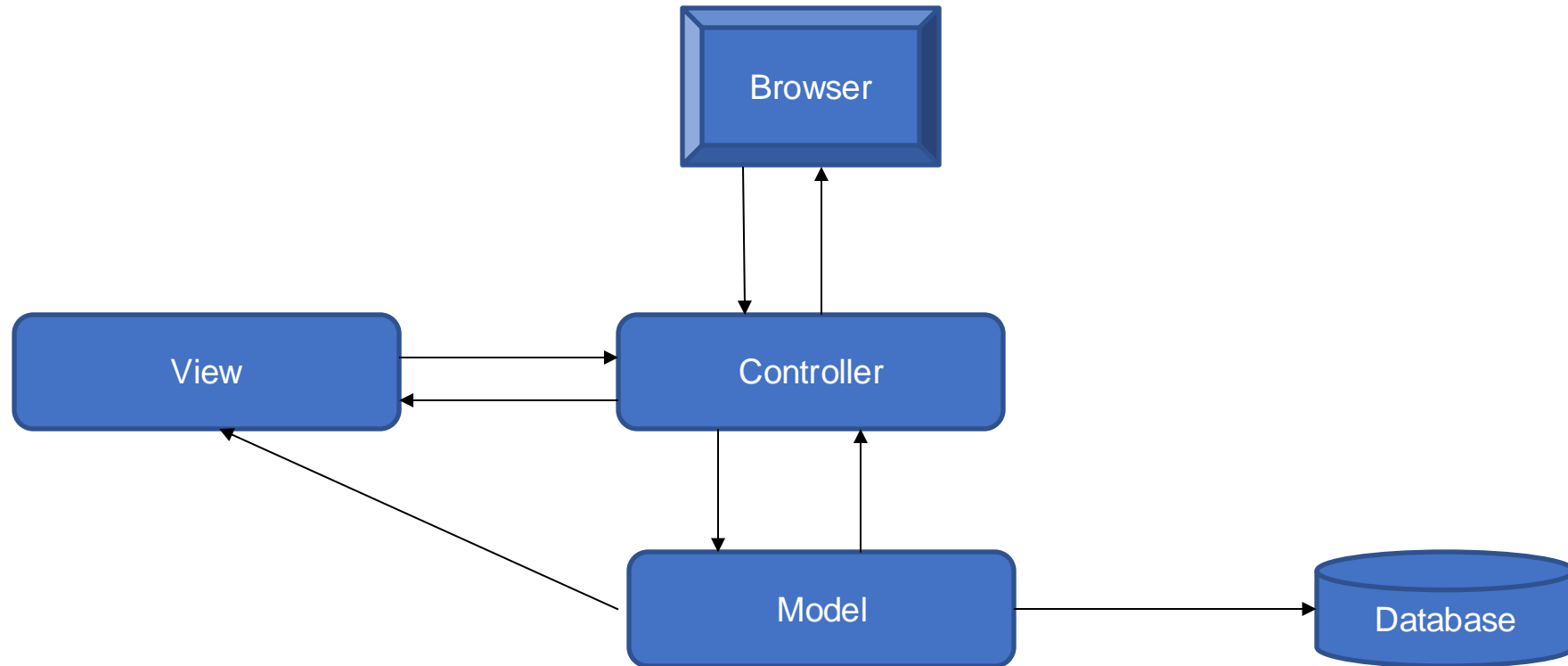
# Model View Controller (MVC)

- Software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements
- Supported well in JavaScript, Python, Ruby, PHP, Java, C#
- Three main logical components:
  - the **model**,
  - the **view**, and
  - the **controller**.
- Separates functionality
- Promotes organized programming



# Model View Controller (MVC)

- Separating the presentation from the model and
- Separating the controller from the view.





# Model View Controller (MVC)

- Data related Logic
- Interacts with the Database
- Communicated with controller
- Updates view (on few Frameworks)



# Model View Controller (MVC)

- User Interface
- Communicates with controller
- Template engines
- Implementing the view in MVC
  - Template View and
    - *Renders information into HTML by embedding markers in an HTML page.*
  - Transform View
    - writing a program that looks at domain-oriented data and converts it to HTML.



# Model View Controller (MVC)

- Receives input( from view)
- Process request( GET, POST)
- Gets data from the model
- Pass data to view
- Load a plain view(static HTML pages)
- Implementing Controller
  - Page controller
  - Front controller
  - Application controller



# Example

- /routes
  - `Users/profiles/id = Users.getUser(id)`
- /controllers

```
Class Users{
  function getUser(id){
    Profile=this.UserModel.getUser(id)
    renderView("/users/profiles", profile)
  }
}
```
- /Models

```
Class UserModel{
  getUser(id){ sql query to db , return data}
}
```



# Example

- /views

/users

/profile

```
<h1> {{profile.name}}</h1>
```

```
<ul>
```

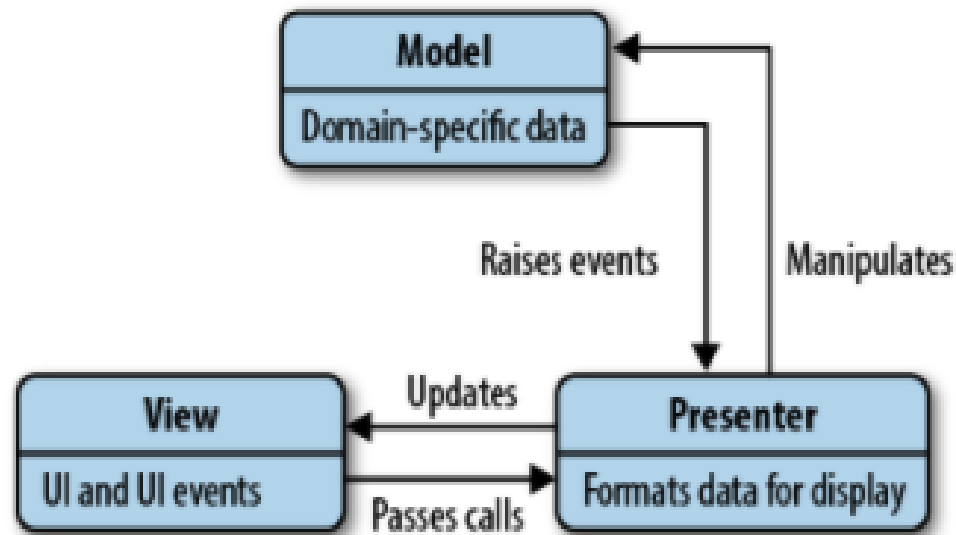
```
  <li>email : {{profile.email}}</li>
```

```
  <li>Phone: {{profile:phone}}</li>
```

```
</ul>
```

# Model View Presenter

- The view is a passive interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data.





# MTV (Model, Template, and View)

- This Model similar to MVC acts as an interface for your data
- The View executes the business logic and interacts with the Model and renders the template.
- It accepts HTTP request and then return HTTP responses.
- The Template is the component which makes MVT different from MVC.
- Templates act as the presentation layer and are basically the HTML code that renders the data.
- The content in these files can be either static or dynamic.



# Thank You!

