الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

مقرر بنيان البرمجيات | كلية الهندسة المعلوماتية | Week 5

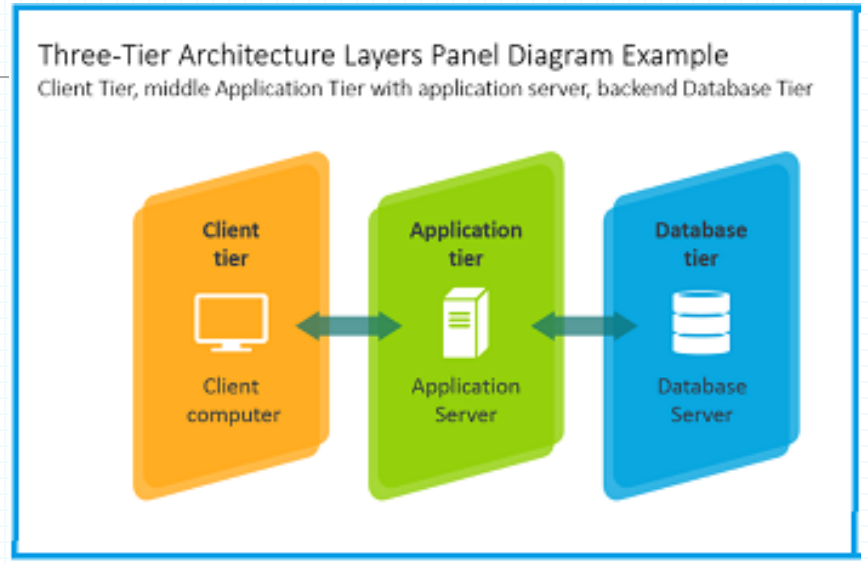
Software Architecture Style

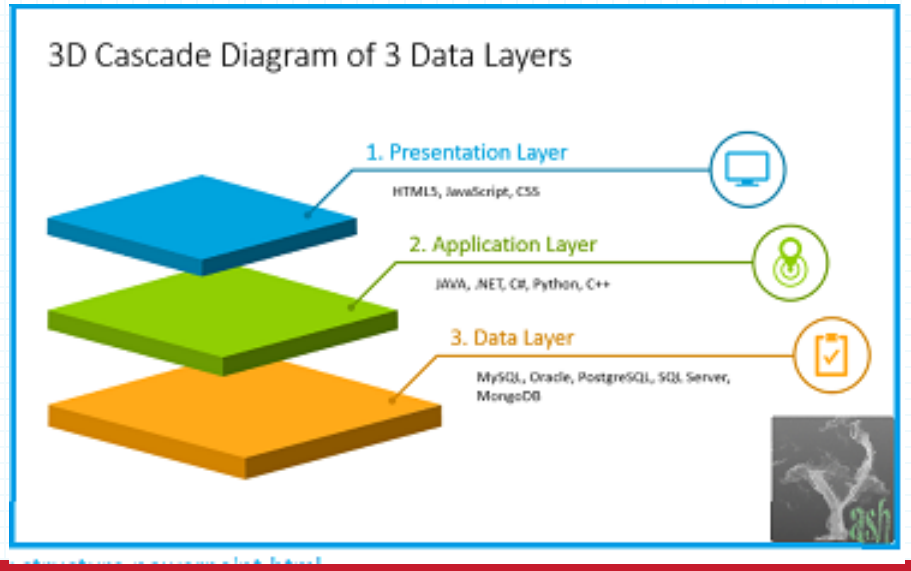
n-Tier, Client-server, Peer-to-Peer, MVC

د. رياض سنبل

3-Tier architecture

- **Layers** are **logical** separation of related-functional code within the application.
- **Tiers** are **Physical** separation of layers which get hosted on Individual servers in an individual computer(process).
- n-Tier advantages:
  - Better Security
  - Scalability : e.g. You can scale up your DB-Tier with DB-Clustering if needed.
  - Maintainability
  - Easily Upgrade or Enhance





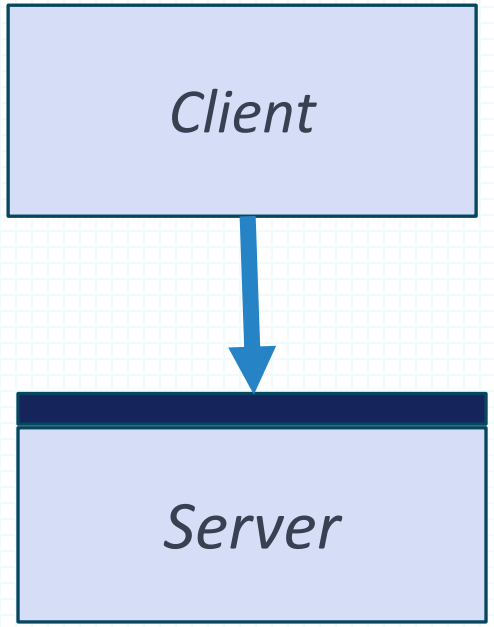
3-Tier architecture

- **Presentation tier:**
  - Focus on user interface concerns: Logic to render the user interface, including the placement of data, formatting the data, and hiding/showing UI components as required
  - Basic validation.
- **Business tier:**
  - Provides the implementation for the BL of the application: Business rules, validations, and calculation logic, Business entities for the application's domain.
  - serves as an intermediary between the presentation and data tiers.
- **Data tier:** The data tier provides functionality to access and manage data.

Client-server architecture (2-tier architecture)

- In a distributed application that uses a client-server architecture, also known as a two-tier architecture, clients and servers communicate with each other directly.
- A client requests some resource or calls some service provided by a server and the server responds to the requests of clients.
- There can be multiple clients connected to a single server.

contains the user interface code

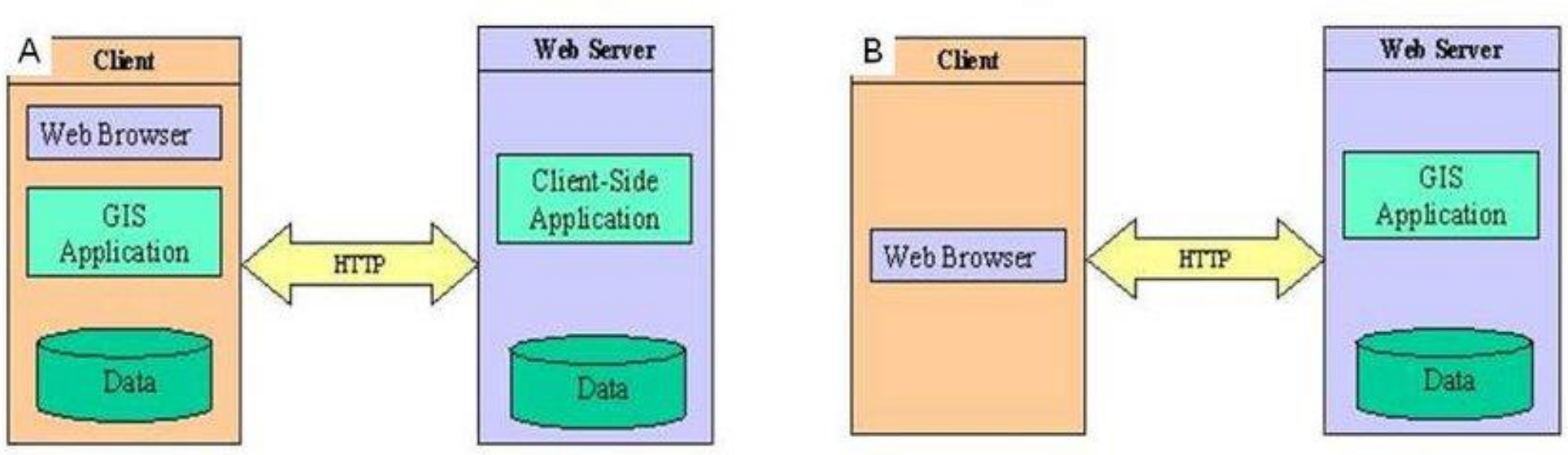


Contains the relational database management system (RDBMS)

The majority of application logic

Client-server architecture (2-tier architecture)

- When the client contains a significant portion of the logic and is handling a large share of the workload, it is known as a thick, or fat, client. When the server is doing that instead, the client is known as a thin client.

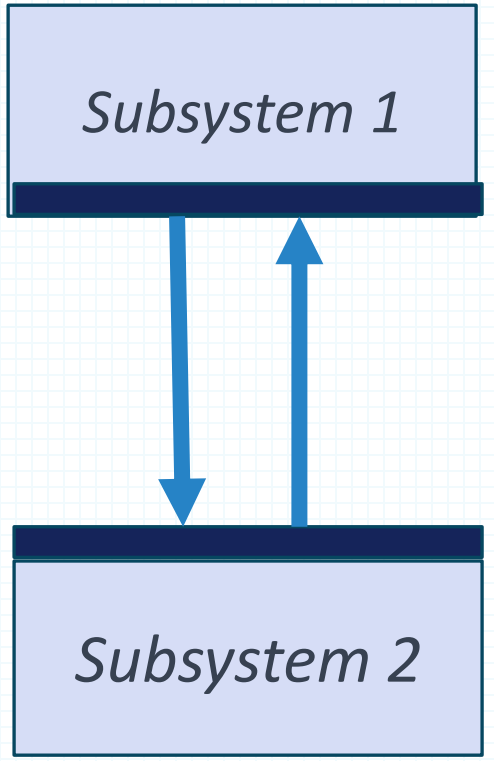


a) Thick client architecture

b) Thin client architecture

Peer-to-Peer Architectural Style

- A generalization of the client/server architectural style.
- Each subsystems can act as client or as servers, in the sense that each subsystem can request and provide services.
- Sample Applications?
  - Easy file sharing.
  - Efficient instant messaging.
  - Smooth voice communication.

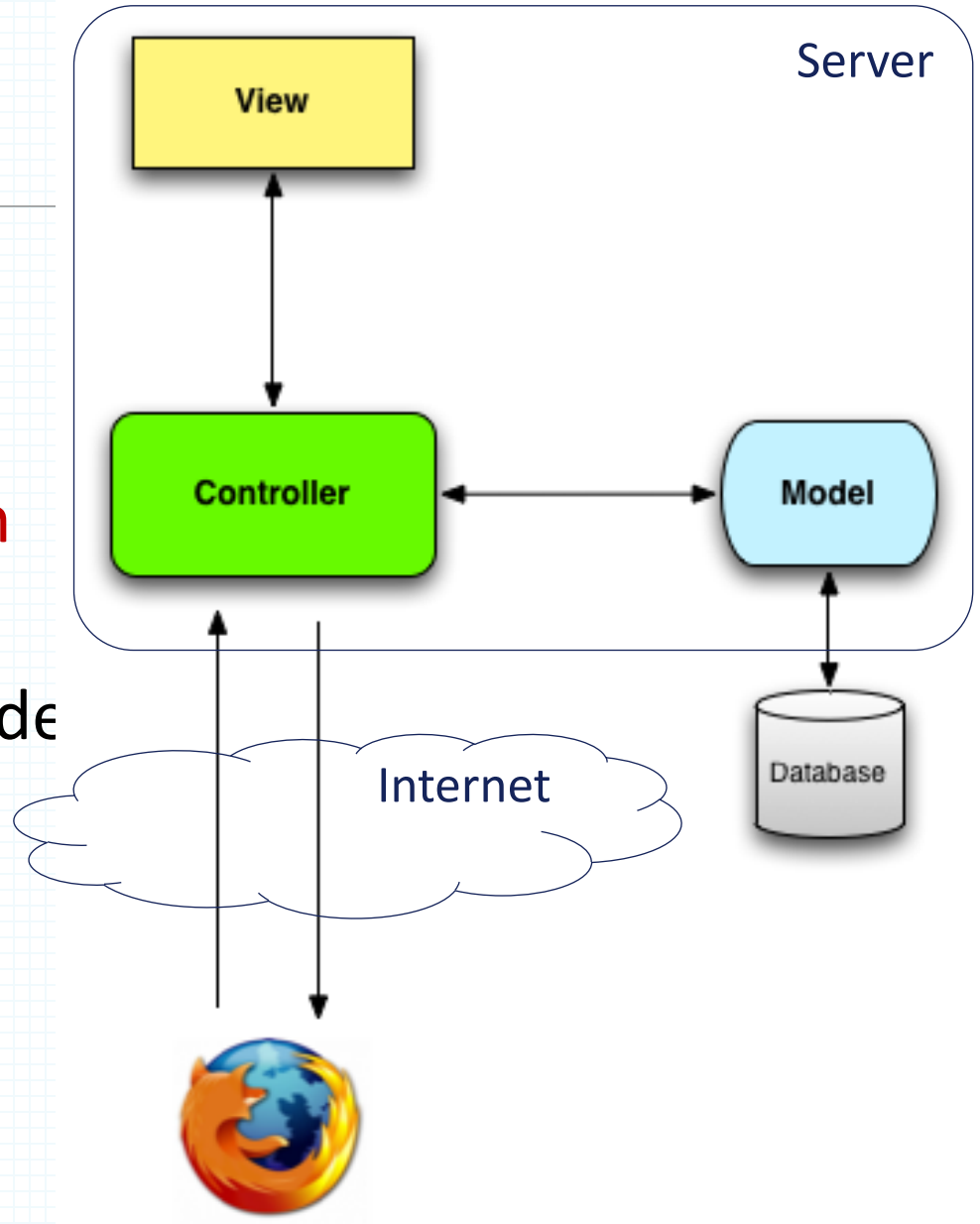


Architecture	Description	Advantages	Disadvantages	Sample Usage
Layered Architecture	Organized into layers where each layer performs a specific role (e.g., presentation, business logic, data access)	- Modular and promotes separation of concerns - Easy to develop, test, and maintain - Supports reusability	- Can lead to performance overhead due to multiple layers - May increase latency in layer communication	Web applications, enterprise software
Client-Server	Divides the system into client and server, where clients request services, and servers provide them	- Centralized control and management - Easy to secure and monitor - Scales well for small to medium systems	- Server can become a bottleneck in high load - Can lead to single point of failure without redundancy	Email systems, online games
n-Tier	Extends client-server with additional layers (e.g., presentation, application, and data tiers)	- Highly scalable and flexible - Separates functionality into distinct services - Easy to distribute layers	- Complex to implement - Higher latency as requests go through multiple layers	E-commerce platforms, Banking systems
Peer-to-Peer (P2P)	Each node (peer) in the network can act as both client and server, sharing resources directly	- Decentralized, no single point of failure - Efficient resource sharing - Scalable without central server	- Difficult to secure and manage - Complex to maintain consistency across peers	File sharing networks, blockchain, VoIP

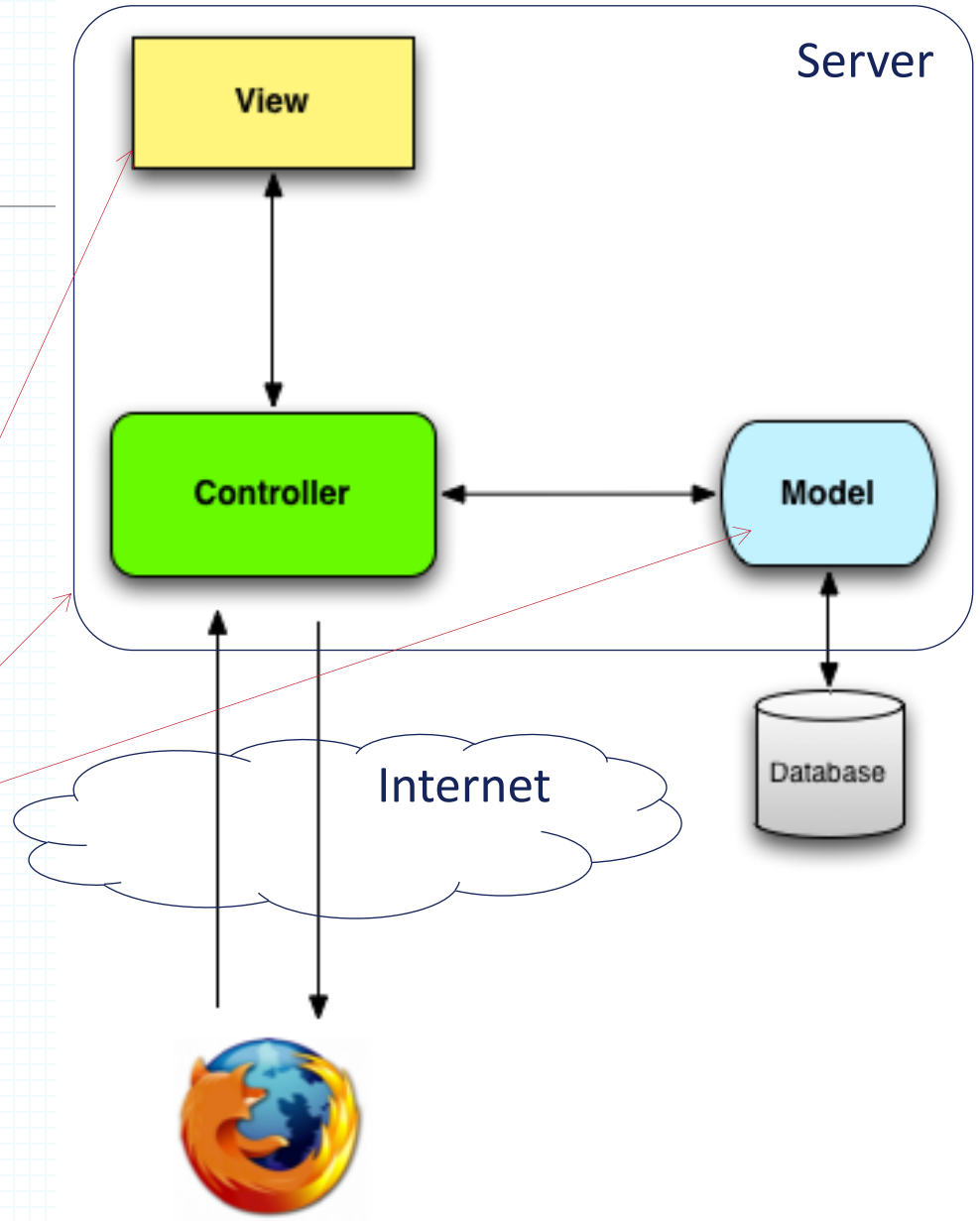
MVC Architecture

Model-View-Controller

- Model–view–controller (MVC) is a software architecture pattern which separates the representation of information from the user's interaction with it.
- A Model View Controller pattern is made up of the following three parts:
  - Model
  - View
  - Controller



Model-View-Controller



Being "architectural", these components may contain many subcomponents (classes, etc.)

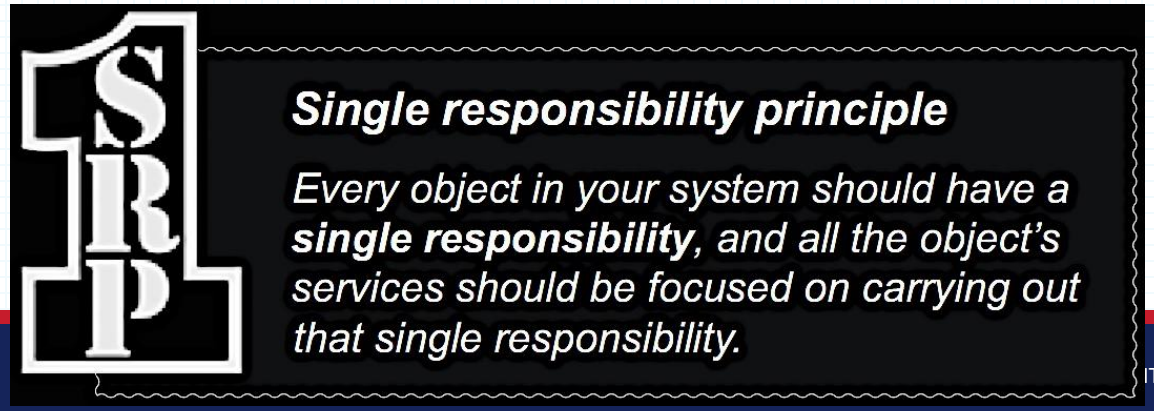
Model-View-Controller

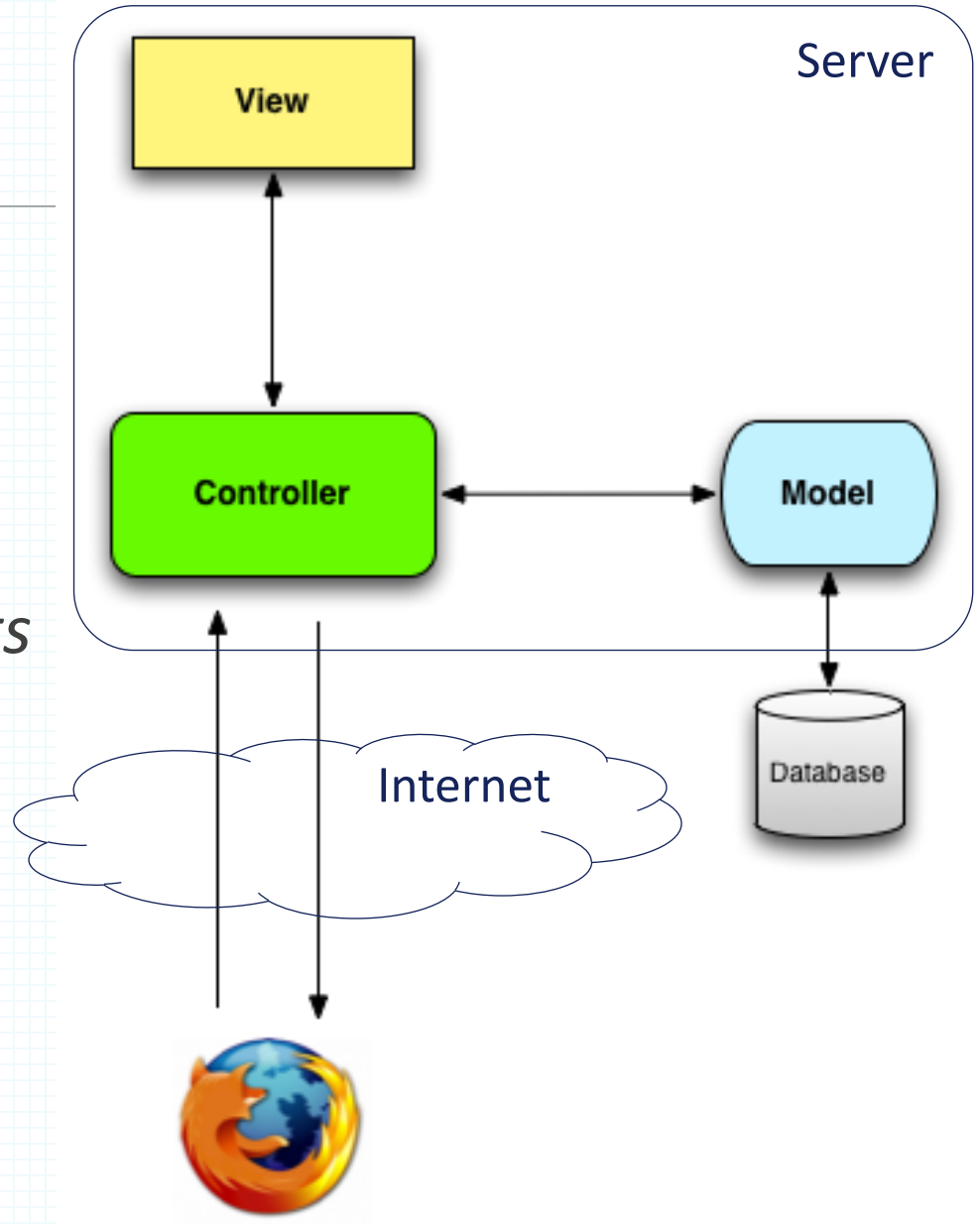
Remember:

Component = clear Responsibilities

"Responsibility-Driven Design"

- ✓ *How to assign responsibilities to objects*
- ✓ *How objects should collaborate*
  - ✓ *What role each object should play in a collaboration*



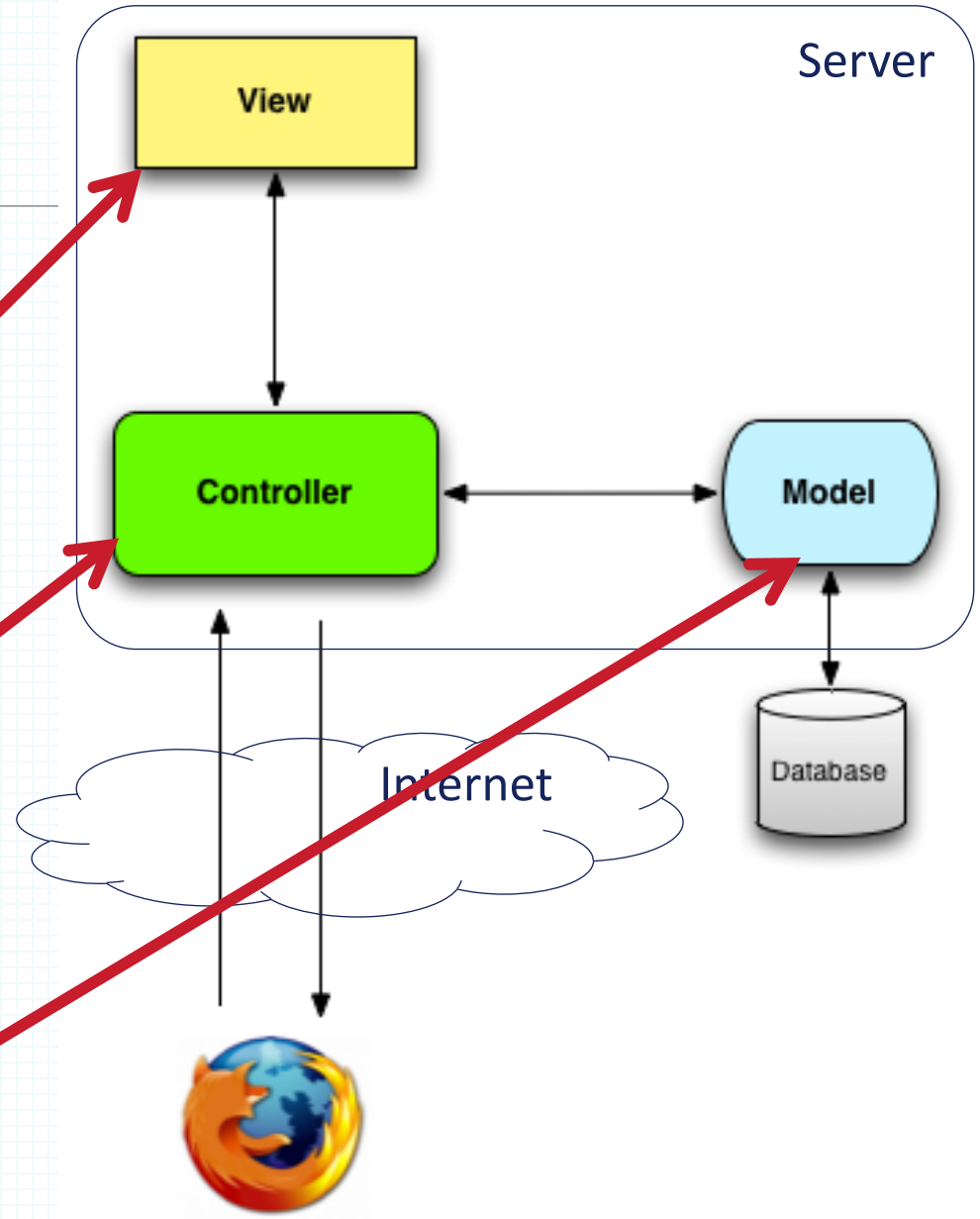


Responsibilities

View
Responsible for UI
(generating HTML)

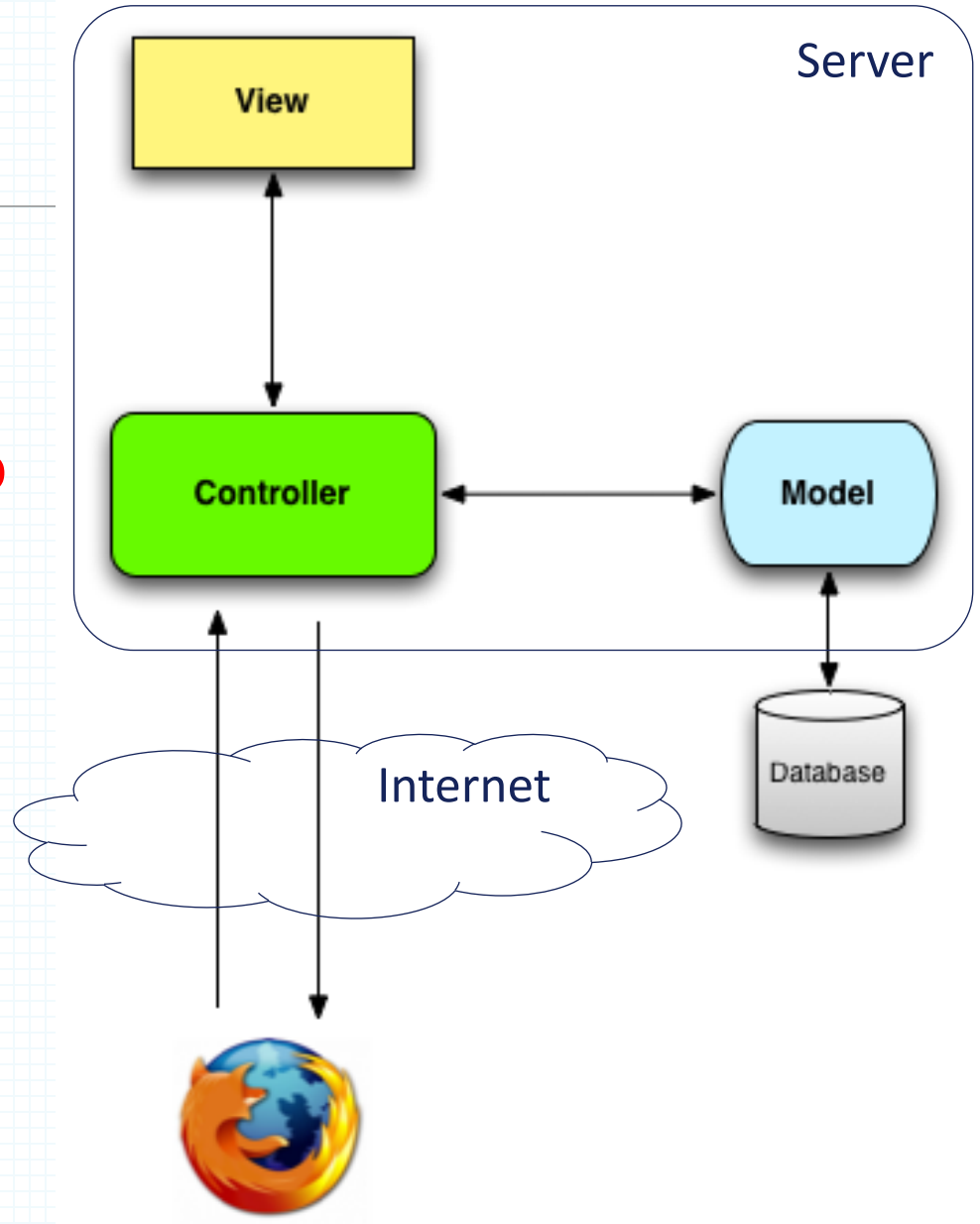
Controller: Translates UI actions into operations on domain objects

Model
Business logic, domain objects



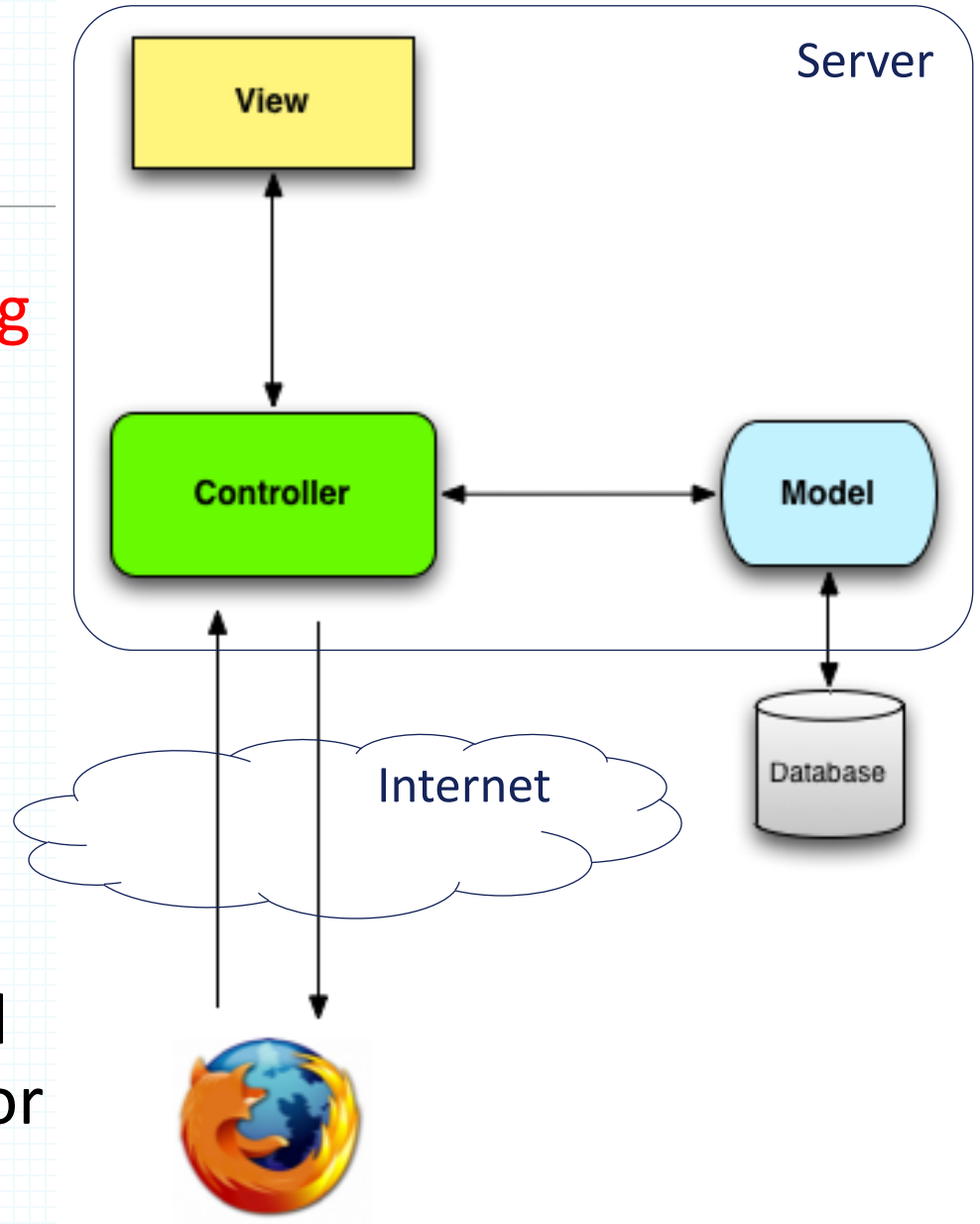
Responsibilities: View

- The View displays the data.
- The View knows about the Model. It will poll the Model about the state, to know what to display.
- The View knows nothing about the Controller.



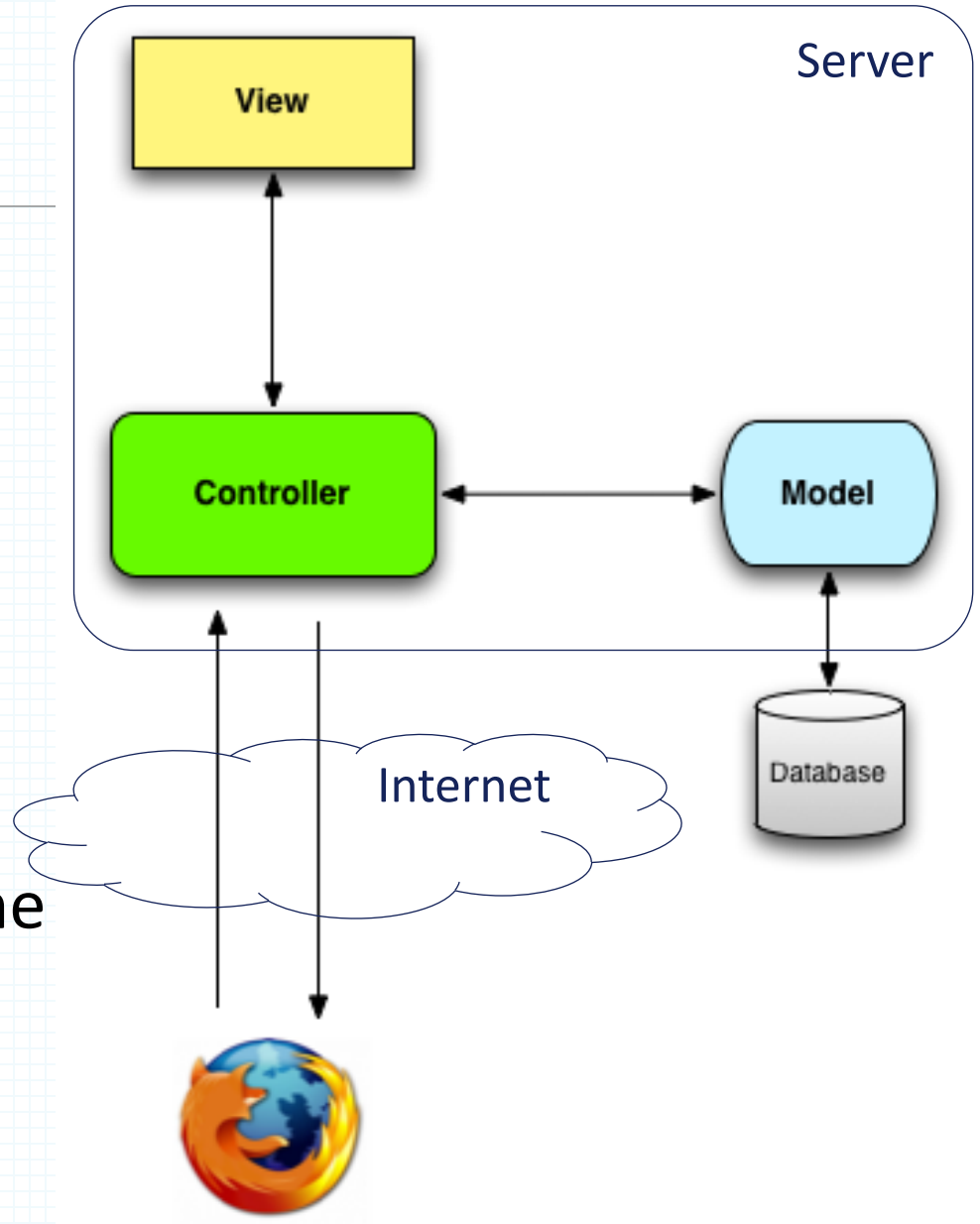
Responsibilities: Model

- The model is responsible for managing and maintaining the data of the application.
- The Model represents the application core (for instance a list of database records).
- It is also called the “domain layer”
- The Model knows only about itself.
- That is, the source code of the Model has no references to either the View or Controller.

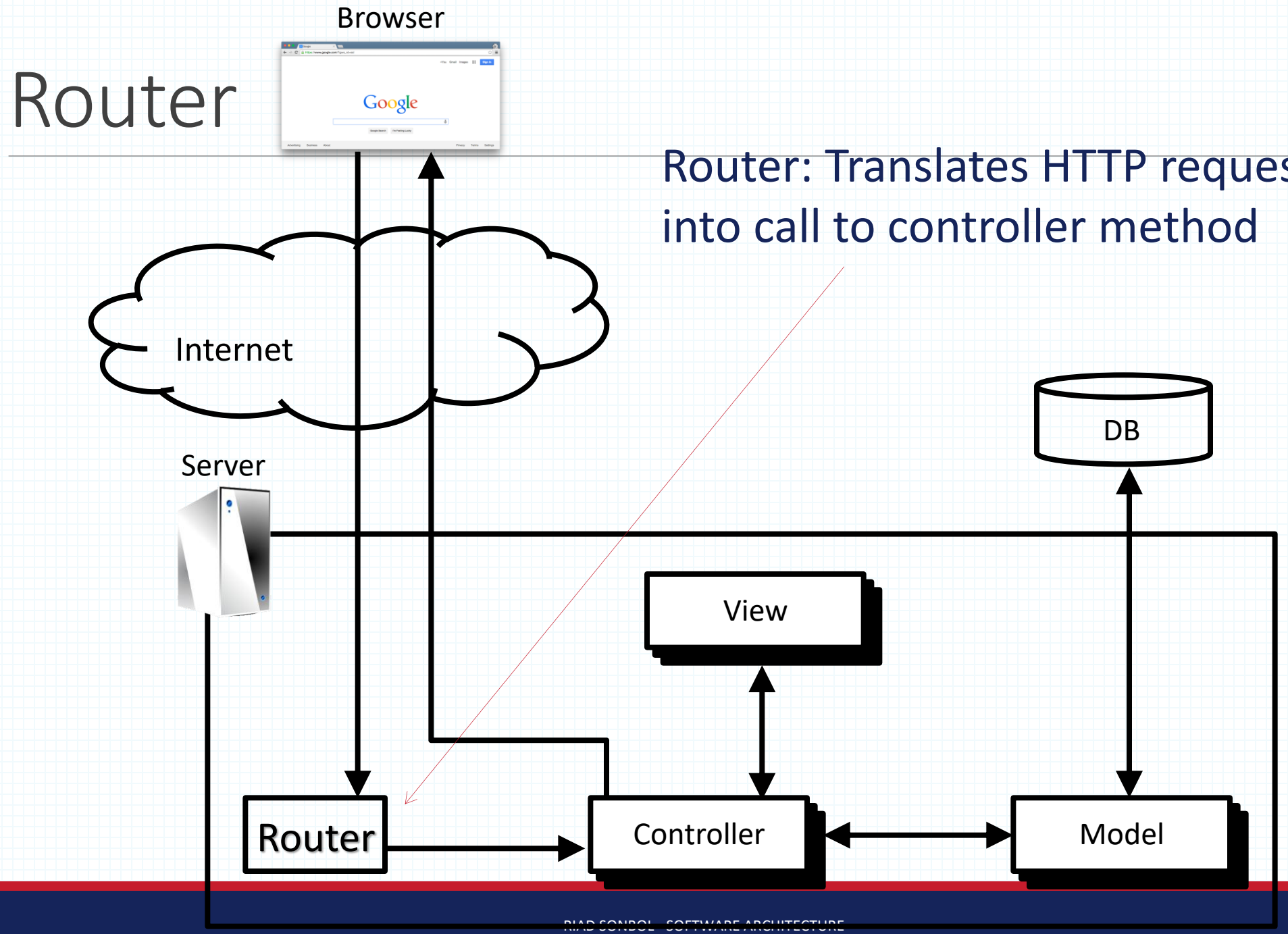


Respo...:Controller

- The Controller is the part of the application that handles user interaction.
- Typically controllers read data from a view, control user input, and send input data to the model.
- It handles the input, typically user actions and may invoke changes on the model and view.
- The Controller knows about both the Model and the View.



Router

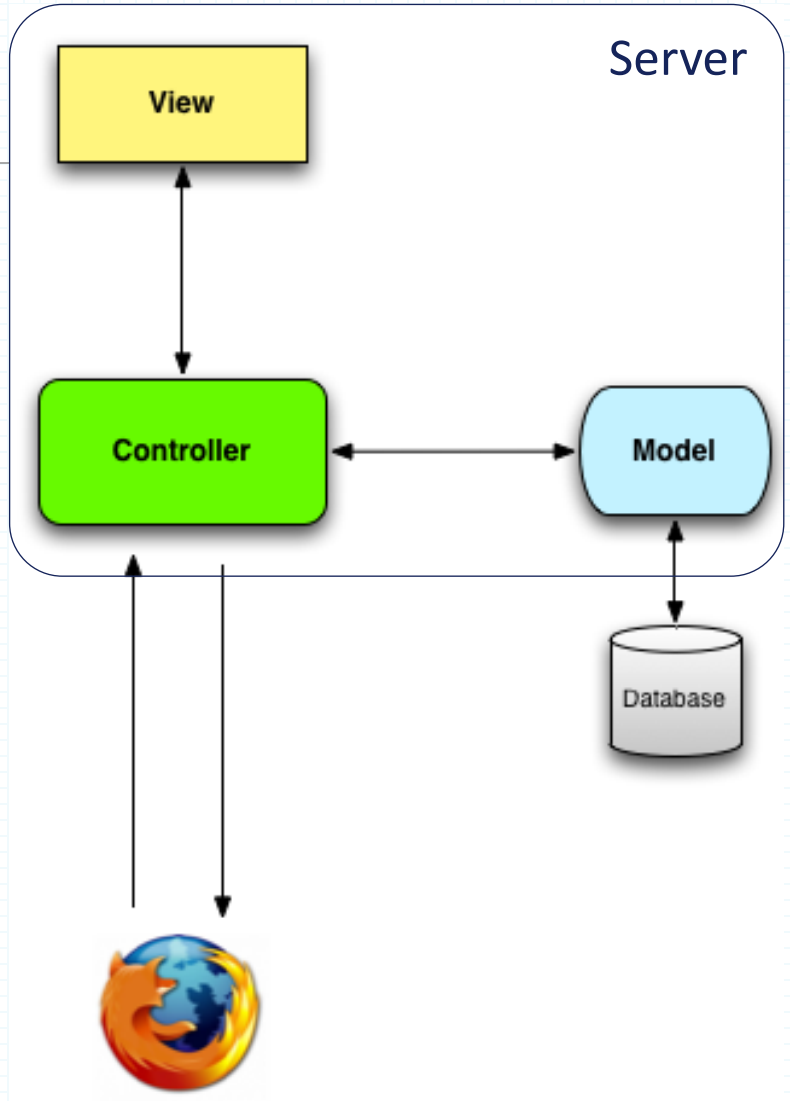

Browser
Google
Router: Translates HTTP request into call to controller method
Internet
Server
DB
View
Router
Controller
Model

Workflow in MVC - Example

The control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g., user presses a button)
2. A controller handles the input event from the user interface, often via a registered handler or callback.
3. The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart).
4. A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents).

The view gets its own data from the model. The model has no direct knowledge of the view.



Advantages

- Clear separation between presentation logic and business logic.
  - No matter how the View class is modified, the Model will still work.
  - Even if the system is moved from a desktop operating system to a smart phone, the Model can be moved with no changes.
  - But the View probably needs to be updated, as will the Controller.
- The MVC programming model is a lighter alternative to traditional Web Page/Forms.
- It is a lightweight, highly testable framework, integrated with all existing features, such as Security, and Authentication.
- parallel development
- easy to maintain and future enhancements

Example

model/Book.php

```php
<?php
class Book {
    public $title;
    public $author;
    public $description;

    public function __construct($title, $author,
$description)
    {
        $this->title = $title;
        $this->author = $author;
        $this->description = $description;
    }
}
?>
```

Example

model/Model.php

```php
<?php
include_once("model/Book.php");

class Model {
    public function getBookList()
    {
        // here goes some hardcoded values to simulate the database
        return array(
            "Jungle Book" => new Book("Jungle Book", "R. Kipling", "A classic book."),
            "Moonwalker" => new Book("Moonwalker", "J. Walker", ""),
            "PHP for Dummies" => new Book("PHP for Dummies", "Some Smart Guy", "")
        );
    }
    public function getBook($title)
    {
        // we use the previous function to get all the books
        // and then we return the requested one.
        // in a real life scenario this will be done through
        // a database select command
        $allBooks = $this->getBookList();
        return $allBooks[$title];
    }
}
?>
```

Example

view/viewbook.php

```
<html>
<head></head>
<body>
    <?php

        echo 'Title:' . $book->title . '<br/>';
        echo 'Author:' . $book->author . '<br/>';
        echo 'Description:' . $book->description . '<br/>';

    ?>
</body>
</html>
```

Example

view/booklist.php

```
<html>
<head></head>
<body>
    <table>
        <tbody>
          <tr><td>Title</td><td>Author</td><td>Description</td></tr>
        </tbody>
        <?php

            foreach ($books as $book)            {
                echo '<tr><td><a href="index.php?book=' .
                    $book->title . '">' . $book->title .
'</a></td><td>' .
                    $book->author . '</td><td>' . $book-
>description . '</td></tr>';
            }
        ?>
    </table>
</body>
</html>
```

Example

controller/Controller.php

```
 public function invoke()
{
        if (!isset($_GET['book']))
        {
            // no special book is requested, we'll show a
list of all available books
            $books = $this->model->getBookList();
            include 'view/booklist.php';
        }
        else
        {
            // show the requested book
            $book = $this->model->getBook($_GET['book']);
            include 'view/viewbook.php';
        }
    }
}
?>
```

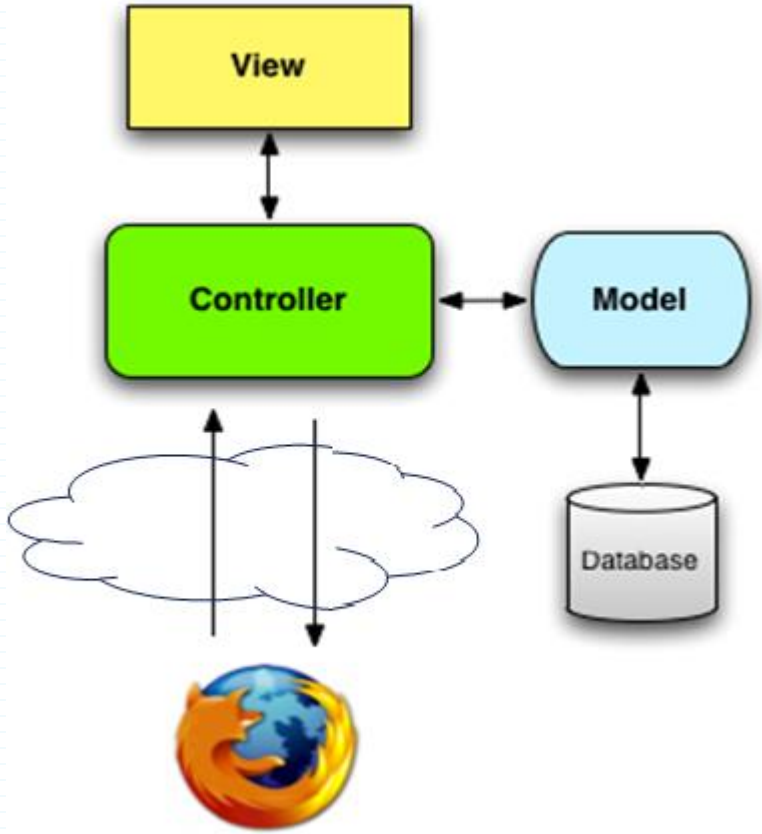
Example

index.php

```
<?php

// All interaction goes through the index and is
forwarded
// directly to the controller

include_once("controller/Controller.php");

$controller = new Controller();
$controller->invoke();

?>
```

Where should you place your **business logic!**

- The placement of business logic largely depends on the **complexity** and **scope** of that logic.

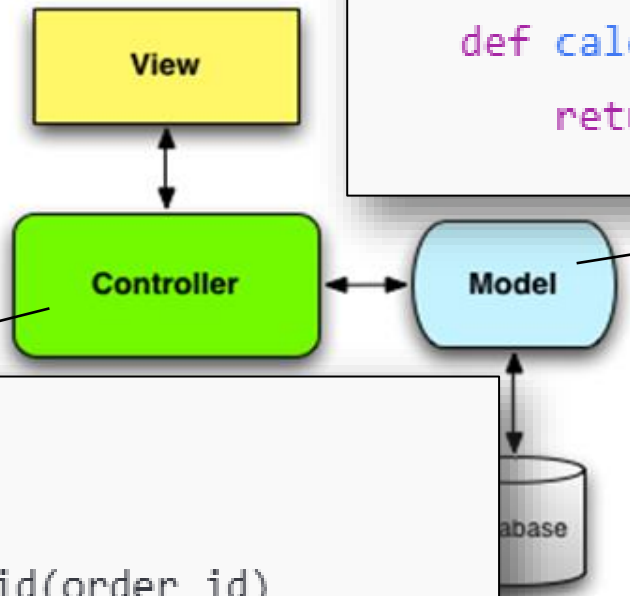


Where should you place your **business logic!**

(1) Small or Simple Business Logic

- If the business logic is relatively simple (e.g., validating input data, basic calculations, or fetching filtered data),

```python
class Order:
    def calculate_total(self):
        return sum(item.price for item in self.items)
```

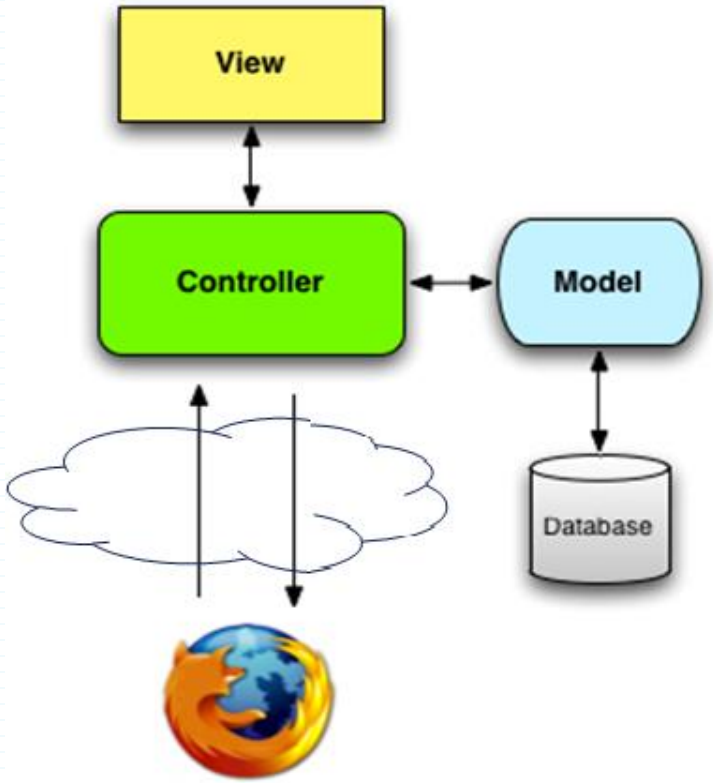



```python
class OrderController:
    def get_order_total(self, order_id):
        order = OrderModel.get_order_by_id(order_id)
        total_price = sum(item.price for item in order.items)
        return total_price
```

Where should you place your **business logic!**

(2) Medium Complexity Business Logic

- If the business logic involves multiple models or complex workflows (e.g., handling orders, applying discounts, or processing payments).
- You should **avoid** placing it directly in Controllers or Models to keep your code maintainable.
- Solution!



Where should you place your **business logic!**

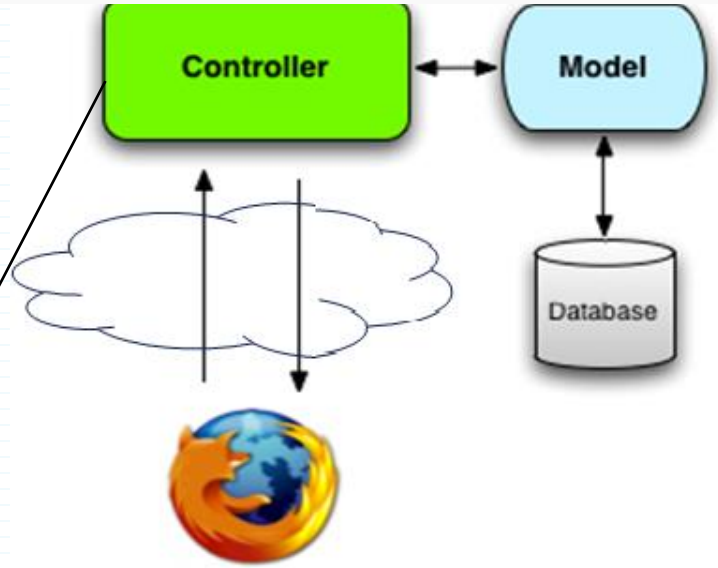
(2) Medium Complexity Business Logic

- (A) Create a separate services module for handling these operations.
- dedicated Service Layer helps to decouple business logic from Controllers and Models, making it easier to test and maintain.

```python
class OrderController:
    def show_order_summary(self, order_id):
        order = OrderModel.get_order_by_id(order_id)
        total = OrderService.calculate_discounted_total(orde
        return render_view('order_summary', {'total': total}
```

The Controller would then use this service!

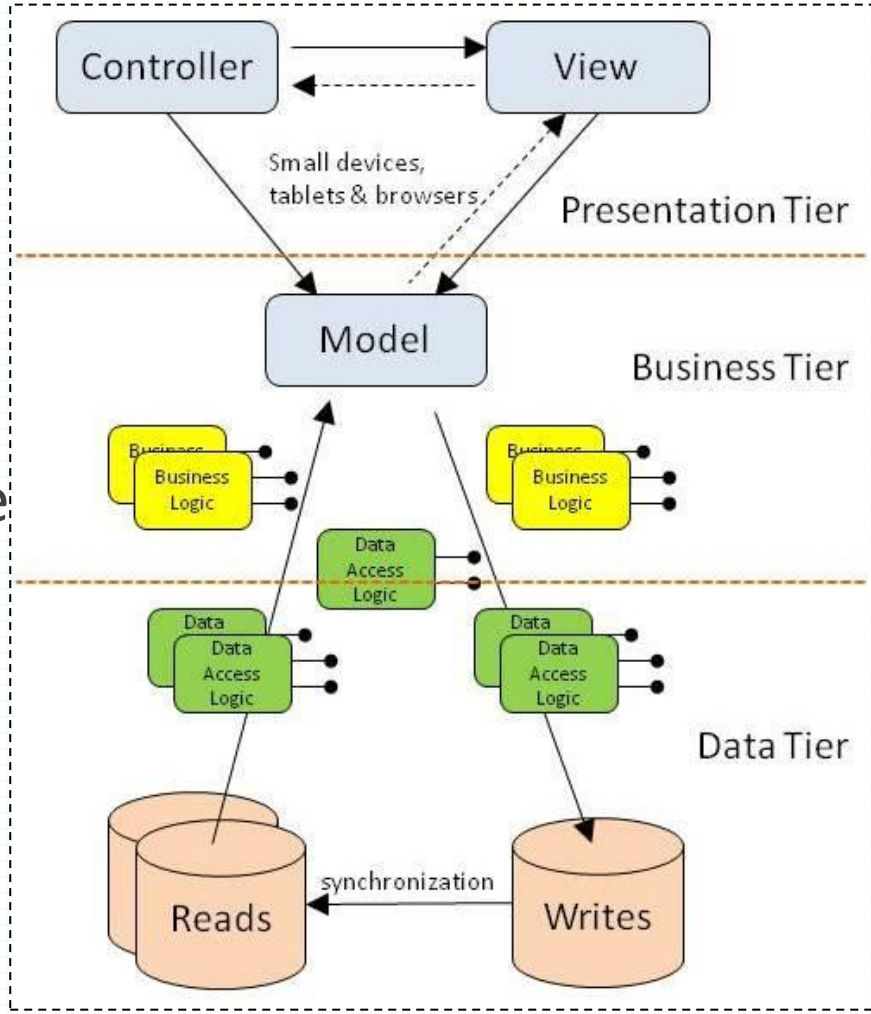
```python
class OrderService:
    def calculate_discounted_total(order):
        total = sum(item.price for item in order.items)
        return total * 0.9 if order.is_eligible_for_discount() else total
```



Where should you place your **business logic!**

(2) Medium Complexity Business Logic

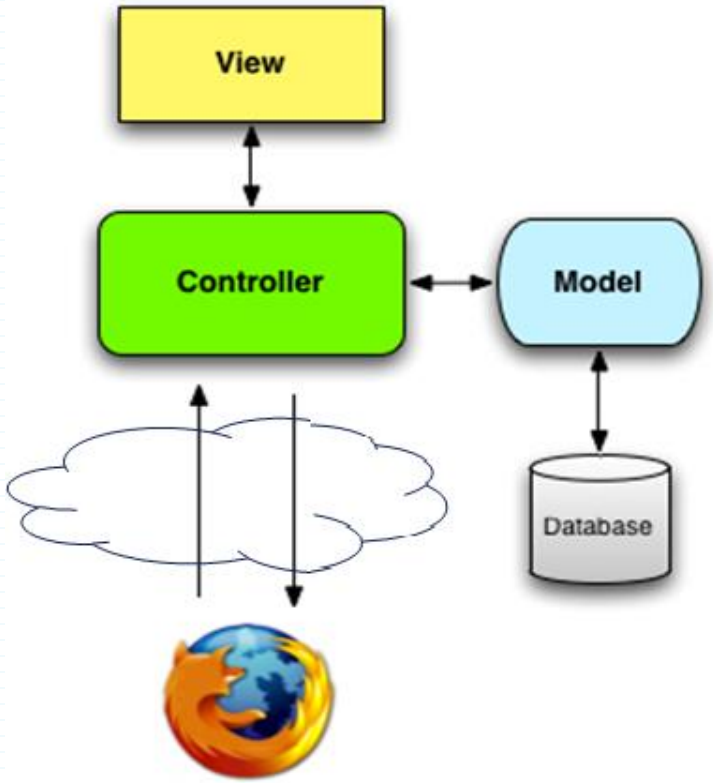
- (B) Treat the **Model** as a **Domain Entity** or **Business Object (BO)** that encapsulates business logic related to the data stored in the database.
- **Models as Domain Entities/BOs** are best when the logic is **data-centric** and directly related to the entity itself.
- However, if the business logic is more complex or involves **cross-cutting concerns** (like external service integrations, multiple models, or workflows that span different entities), it's better to use a **Service Layer**.



Where should you place your **business logic!**

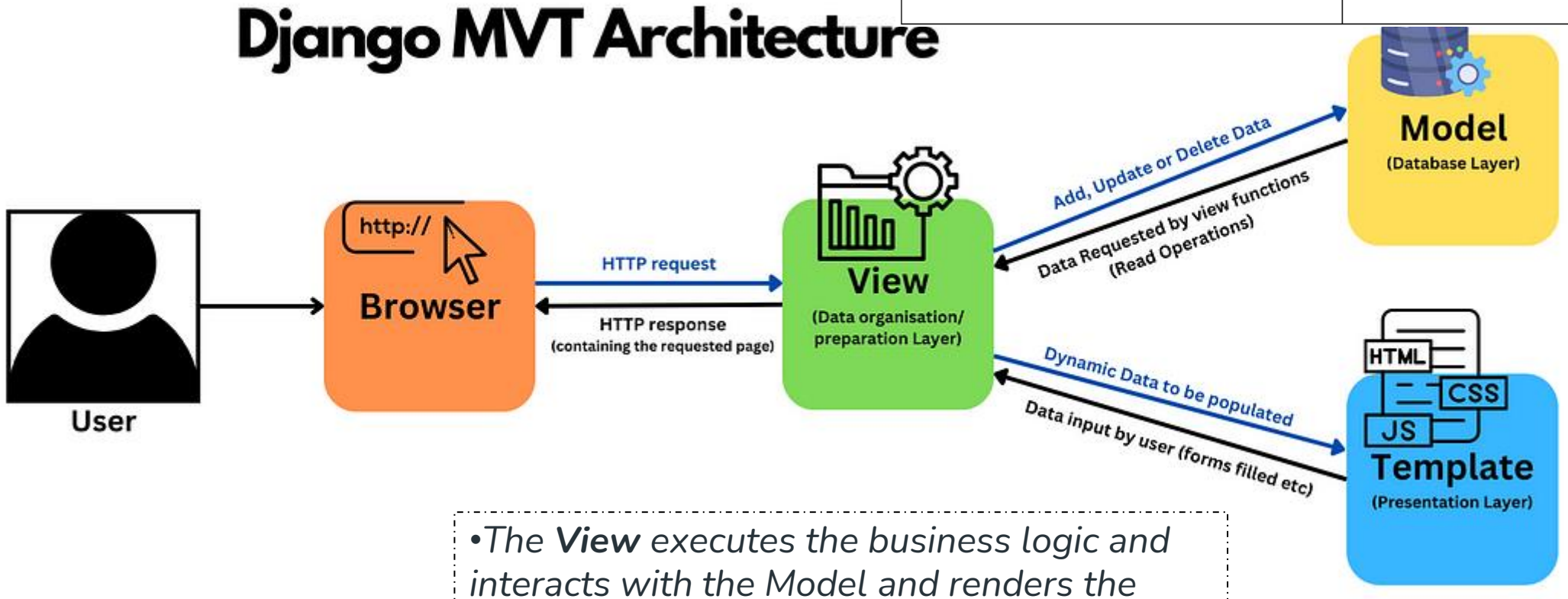
(3) Heavy or Long-Running Business Logic

- For complex operations that are computationally intensive or need to be executed asynchronously (e.g., generating reports, batch processing, or sending emails) it's best to decouple them from the request/response cycle.
- Offloading heavy processing to background workers improves performance and scalability.



MVC Variations

MVC	MVT
Controller	View
Model	Model
View	Template



- *The **View** executes the business logic and interacts with the Model and renders the template. It accepts HTTP request and then return HTTP response.*

- *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.*

حسن التصميم السابق باستخدام ما تعرفنا عليه في محاضرة اليوم!

حالة عملية للنقاش

شركة تطوير تعمل على بناء نظام برمجي لإدارة خدمات الرعاية الصحية عن بُعد، ويتميز النظام المراد بالخصائص التالية:

- يتكامل النظام مع أجهزة قياس العلامات الحيوية للمرضى لتحصيل البيانات والقياسات الحيوية مباشرة لتحليلها.
- يوفر واجهة مستخدم تفاعلية للأطباء والممرضين تتيح لهم مراقبة الحالة الصحية للمرضى بشكل لحظي عند الحاجة وتقديم الاستشارات عبر الإنترنت.
- يتيح النظام للمرضى حجز المواعيد والتواصل مع الأطباء عبر المحادثات النصية أو مكالمات الفيديو.
- يقدم النظام توصيات طبية مخصصة بناءً على بيانات المرضى وسجلهم الصحي باستخدام تقنيات الذكاء الصنعي.
- يوفر النظام خدمة إدارة الوصفات الطبية الإلكترونية، حيث يمكن للأطباء إصدار الوصفات وتحويلها مباشرة إلى الصيدليات.
- يدعم النظام تخزين الملفات الطبية بشكل آمن مع إمكانية مشاركتها مع مقدمي الرعاية الصحية بناءً على موافقة المريض.