Osman Ghani Granday

FA21-BSE-024 |

Software Design and Architecture

Assigmnment # 1

**A Detailed Information About MVC**

The Model-View-Controller (MVC) is a popular architectural pattern used in software development. It helps developers organize their code more efficiently and create applications with better maintainability and scalability.

**History:**

The MVC pattern was first introduced by Trygve Reenskaug in 1979 while working on the Smalltalk project at Xerox PARC. The goal was to create an architectural pattern that would separate an application's data (Model) user interface (View), and control logic (Controller) into distinct components, making it easier to manage and maintain the codebase.

**Pros of MVC architecture:**

1. Separation of concerns: MVC separates the application logic into three distinct components, making it easier to manage and maintain the codebase.
2. Reusability: The modular design of MVC allows developers to reuse components, reducing development time and effort.
3. Easier testing: Since the components are separated, it's easier to test individual parts of the application without affecting others.
4. Improved collaboration: MVC allows multiple developers to work on different components simultaneously, improving overall productivity.
5. Scalability: The modular design of MVC makes it easier to scale applications as they grow in complexity.

**Cons of MVC architecture:**

1. Increased complexity: MVC can be more complex than other architectural patterns, especially for smaller applications or projects with tight deadlines.
2. Overhead: The separation of concerns in MVC can lead to additional overhead, such as creating and managing multiple files and directories.
3. Learning curve: Developers new to MVC might find it challenging to understand and implement the pattern correctly.
4. Components of MVC:
   * Model: The Model represents the application's data and business logic. It encapsulates the data, handles data manipulation, and interacts with the database or other data sources.
   * View: The View is responsible for the presentation layer of the application. It defines how the data should be displayed to the user and handles user interface elements.
   * Controller: The Controller acts as an intermediary between the Model and the View. It processes user inputs, manipulates data from the Model, and updates the View accordingly.
5. Workflow in MVC:
   * The user interacts with the application through the View (e.g., clicking a button or submitting a form).
   * The Controller receives the user input, processes it, and communicates with the Model to perform the necessary data manipulation.
   * The Model updates the data and sends it back to the Controller.
   * The Controller then updates the View with the new data, and the user sees the changes in the user interface.
6. Variations of MVC:
   * There are several variations of the MVC pattern, such as Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), and Model-View-Adapter (MVA). These variations aim to address specific concerns or improve upon the original MVC pattern.
7. MVC in modern web development:
   * MVC is widely used in web development, particularly in frameworks like Ruby on Rails, Django, ASP.NET MVC, and Laravel. These frameworks provide built-in tools and conventions to help developers implement the MVC pattern efficiently.

**Examples of MVC usage:**

MVC is widely used in various software applications, particularly in web development. Some popular web frameworks that implement MVC include:

1. **Ruby on Rails**: A popular web application framework for the Ruby programming language that follows the MVC pattern.
2. **Django**: A high-level Python web framework that encourages rapid development and clean, pragmatic design, using the MVC pattern.
3. **ASP.NET MVC:** A Microsoft web framework that implements the MVC pattern for building dynamic, data-driven web applications using the .NET platform.
4. **Laravel:** A popular PHP web framework that follows the MVC pattern and is designed for building robust web applications.

**REFRENCES**

1. History of MVC:
2. "A Brief History of MVC" by Pete LePage: <https://www.scholarhat.com/tutorial/mvc/a-brief-history-of-aspnet-mvc-framework>
3. Pros and Cons of MVC:
   * "MVC Architecture: Advantages and Disadvantages" by educative.io:  <https://www.educative.io/answers/what-are-the-pros-and-cons-of-different-software-architectures>
4. Examples of MVC usage in web frameworks:
   * Ruby on Rails: <https://rubyonrails.org/>
   * Django: <https://www.djangoproject.com/>
   * ASP.NET MVC: <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions>
   * Laravel: <https://laravel.com/>
5. Variations of MVC:
   * "MVC, MVP, MVVM, and MVA: What's the Difference?" by GeeksforGeeks: <https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/>
6. MVC in modern web development:
   * Ruby on Rails: <https://rubyonrails.org/>
   * Django: <https://www.djangoproject.com/>
   * ASP.NET MVC: <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions>
   * Laravel: <https://laravel.com/>

**Data-centred Architectural Style**

**Failed Example:** The United States' **HealthCare.gov** launch (initial phase)

HealthCare.gov was launched in 2013 as a part of the Affordable Care Act (ACA) to provide a platform for Americans to purchase health insurance. The initial launch of the website was marred by numerous issues, many of which were related to its data-centric architecture.

**Reasons for failure:**

1. Inadequate data management: The website was designed to handle large volumes of data, but it struggled with data validation, consistency, and integrity. This resulted in incorrect calculations of subsidies, enrollment failures, and other issues that negatively impacted user experience.
2. Complex data integration: HealthCare.gov required integration with multiple data sources, such as the Internal Revenue Service (IRS), Department of Homeland Security (DHS), and various state-level systems. The complexity of integrating these disparate systems led to data exchange problems, causing delays and errors in the enrollment process.
3. Insufficient testing: The data-centric architecture of HealthCare.gov was not adequately tested before its launch. This resulted in numerous performance issues, data inconsistencies, and system crashes, which made it difficult for users to navigate and enroll in health insurance plans.
4. Scalability issues: HealthCare.gov was expected to handle a high volume of users, but its data-centric architecture was not initially designed to scale efficiently. This led to slow response times, system crashes, and overall poor performance during the initial weeks of the launch.

REFRENCES

1. A report from the U.S. Department of Health and Human Services (HHS) Office of Inspector General on the challenges faced by HealthCare.gov, including contractor performance and oversight issues: <https://oig.hhs.gov/oei/reports/oei-06-14-00350.asp>
2. An article from The New Yorker discussing the technical challenges faced by HealthCare.gov and the efforts to fix them: <https://www.newyorker.com/tech/annals-of-technology/healthcare-gov-it-could-be-worse>
3. A look back at technical issues with Healthcare.gov by Brookings:

<https://www.brookings.edu/articles/a-look-back-at-technical-issues-with-healthcare-gov/>

1. Real-time outages and **problems** **for** **Healthcare**.**gov**. Is the website down? Can't log in or get insurance? Here you see what is going in. <https://downdetector.com/status/healthcare-gov/>
2. Congress: HealthCare.gov Technical Problems May Impact Enrollment:<https://healthpayerintelligence.com/news/congress-healthcare.gov-technical-problems-may-impact-enrollment>
3. Why healthcare.gov has so many problems

Steven Bellovin, Special to **CNN:** <https://edition.cnn.com/2013/10/14/opinion/bellovin-obamacare-glitches/index.html>

**Successful Example: Netflix**

Netflix is a prime example of a successful software built using a data-centric architecture. Netflix's architecture is designed to collect, process, and analyze massive amounts of data to provide personalized recommendations, stream videos seamlessly, and offer a reliable user experience.

Here's a simple explanation of how they did it:

1. Moving to the cloud: Netflix moved their entire system to a virtual space called the cloud, provided by Amazon Web Services (AWS). This allowed them to store and process huge amounts of data and make sure their service was always available.
2. Breaking it down into small parts: Netflix divided their large system into smaller pieces called microservices. Each microservice has a specific job, like managing user information or suggesting movies. This made it easier for Netflix to handle data and improve their services.
3. Personalized suggestions: Netflix collects and analyzes lots of data about what users watch and like. They use this information to suggest movies and shows that each user might enjoy. This is an important part of their design and helps keep users happy and engaged.
4. Storing and processing data: Netflix uses different tools to store and process all the data they collect. These tools help them manage huge amounts of information efficiently and make sure their services work well, even when lots of people are using them at the same time.
5. Delivering videos quickly: Netflix uses a special network called Open Connect to send videos to users. By placing servers close to users, they can deliver videos quickly and without interruptions, even during busy times.
6. Making sure it always works: Netflix designed their system to be strong and reliable. They test it regularly to find and fix any problems, so users can enjoy their service without any issues.
7. Using data to make decisions: Netflix pays close attention to the data they collect. They use this information to make smart decisions about things like which movies to buy or produce, and how to improve their service.

**Reasons for success:**

1. Scalability: Netflix's data-centric architecture allows it to handle an enormous amount of data and scale according to the increasing user base and data size.
2. Personalization: By leveraging data, Netflix can provide personalized content recommendations, which significantly improves user engagement and satisfaction.
3. Resilience: Netflix's architecture is designed to be fault-tolerant and highly available, ensuring that users can access the service without interruptions.
4. Data-driven decision making: Netflix relies on data to make informed decisions about content creation, acquisition, and marketing strategies.

**REFRENCES**

1. Netflix's blog post on their microservices architecture: <https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/>
2. Netflix's blog post on their recommendation system: <https://recoai.net/netflix-recommendation-system-how-it-works/>
3. how Netflix's data-driven culture and architecture contribute to its success: <https://www.linkedin.com/pulse/3-minute-breakdown-how-netflix-data-driven-culture-them-phil-araujo>
4. Netflix's use of data to personalize its service and improve customer satisfaction: <https://www.econtribune.com/post/how-netflix-uses-big-data-for-customer-satisfaction>
5. how Netflix's data-centric architecture enables it to scale and innovate rapidly: <https://www.linkedin.com/pulse/inside-netflix-deep-dive-its-cutting-edge-system-architecture>

***And more information and how the Netflix works visit the link***

[***https://research.netflix.com/***](https://research.netflix.com/)