Ростелеком

3.2.2 Шаблон Model View Controller

Модуль 3. Интеграция приложений и prompt engineering

Оглавление

1. Введение **3**

2. Шаблон MVC **3**

3. MVC в библиотеке Flask **4**

4. Компонент Model **6**

5. LLM service **7**

6. Компонент View **9**

7. Компонент Controller **10**

8. Взаимодействие компонентов **11**

9. Расширенные возможности **13**

10. Альтернативные фреймворки и архитектуры **15**

11. Заключение **16**

1. Введение

Добрый день, уважаемые слушатели! Сегодня мы поговорим о шаблоне Model-View-Controller (MVC) и интеграции языковых моделей с ним. Этот паттерн — один из ключевых в разработке программного обеспечения, и понимание его принципов поможет вам создавать более структурированные и эффективные решения.

Конечно, MVC подходит не для всех задач. Если ваше приложение очень простое, внедрение MVC может излишне усложнить архитектуру. И наоборот, MVC не всегда оптимален для высоконагруженных real-time приложений, где важна минимальная задержка. В таких случаях могут использоваться другие паттерны.

Но мы рассматриваем его, т.к. для профессионального prompt-инженера понимание архитектуры приложения поможет лучше выполнять свою работу. По итогу занятия вы получите готовый шаблон для чата, который можно адаптировать под другие задачи с учетом разделения логики, представления и управления. Давайте разберемся, как это работает более подробно.

2. Шаблон MVC

Начнем с определения. MVC — это архитектурный паттерн, который разделяет приложение на три основных компонента: Model (модель), View (представление) и Controller (контроллер). Идея MVC появилась еще в 1970-х годах, но широкую популярность обрела с развитием веб-разработки, где она помогает четко разделять логику, интерфейс и управление данными.

Model отвечает за сохранение и извлечение данных. Она не знает, как эти данные будут отображаться или обрабатываться пользователем — ее задача хранить, обрабатывать и возвращать информацию. View — это визуальное представление данных, то, что видит пользователь. View не должно содержать сложной логики, только отображение. Controller выступает посредником между Model и View: он принимает пользовательский ввод, обрабатывает его, обновляет Model и выбирает, какое View показать.

Использование MVC в веб-разработке дает несколько преимуществ. Во-первых, код становится более организованным и поддерживаемым. Во-вторых, разделение ответственностей упрощает командную работу: один разработчик может работать с Model, другой — с View, не мешая друг другу. В-третьих, MVC облегчает тестирование, так как каждый компонент можно проверять отдельно.

3. MVC в библиотеке Flask

Перед началом работы с любым проектом сначала нужно создать виртуальное окружение, чтобы в него установить все необходимые библиотеки. Для этого можно использовать терминал, куда нужно ввести команду `python -m venv venv`, а затем активировать окружение. После активации установите Flask и дополнительные пакеты, например, Flask-SQLAlchemy для работы с базой данных, командой `pip install flask flask-sqlalchemy`. На практической работе мы разберем как это можно сделать с помощью мыши в редакторе VS code.

Flask – это библиотека для создания веб-приложений, в том числе с использованием паттерна MVC. Минимальное приложение на flask должно содержать основной файл, обычно он называется app.py.

Файл `models.py` будет содержать файлы с классами Model, которые описывают структуру данных и бизнес-логику. В папке `templates` размещаются компоненты View – это обычные html файлы с возможностью добавлять в них код на python. Контроллеры обычно находятся в основном файле приложения или в папке `controllers`, если проект большой.

Конфигурация простого приложения задается в основном файле `app.py`. Там вы создаете экземпляр Flask-приложения, настраиваете подключение к базе данных, регистрируете маршруты.

Flask может работать в режиме development (разработки) или production (работа на сервере с реальными пользователями и данными). По умолчанию запускается режим разработчика, о чем можно прочитать в командной строке.

Рассмотрим пример простого приложения - чат. В `models.py` вы определяете класс ChatHistory с историей сообщений. В `app.py` создаете маршруты: главная страница получает все запросы из истории чата и передает их в шаблон `index.html`. Маршрут `/chat` обрабатывает форму добавления нового запроса от пользователя. Шаблон представления View определен в файле `index.html` и хранится в папке `templates`.

Таким образом, правильная структура проекта помогает поддерживать четкое разделение на Model, View и Controller даже в небольших приложениях. В prompt-инжиниринге понимание этой

структуры позволит вам лучше организовывать сложные запросы, разбивая их на логические компоненты.

4. Компонент Model

Model — это компонент паттерна MVC, который отвечает за данные и бизнес-логику приложения. Его основная задача — управлять информацией: хранить, обрабатывать, проверять и предоставлять ее другим частям системы. Model не зависит от того, как данные будут отображаться или как пользователь будет с ними взаимодействовать — это зона ответственности View и Controller.

В веб-разработке Model часто работает с базой данных. Например, в нашем приложении для чата, Model будет отвечать за хранение истории запросов пользователя и ответов LLM. Она также может содержать правила валидации — например, проверять, что запрос не пустой или не содержит ненормативной лексики.

Для языка python слой Model можно реализовать с помощью библиотеки SQLAlchemy, которая предоставляет ORM (Object-Relational Mapping). ORM позволяет работать с базой данных как с обычными объектами Python, без написания сложных SQL-запросов. Например, класс ` ChatHistory`, будет соответствовать таблице в базе данных, а его атрибуты — колонкам этой таблицы. Когда вы сохраняете статью, ORM автоматически преобразует объект в SQL-запрос и выполнит его.

Связь Model с базой данных — ключевой момент. ORM упрощает эту связь, избавляя вас от ручного написания запросов и уменьшая

вероятность ошибок. Кроме того, ORM обеспечивает безопасность, защищая от SQL-инъекций.

5. LLM service

Помимо запросов к базе данных, в компоненте Model происходит формирование промпта и обращение к API LLM.

Формирование промптов - это процесс создания текста, который мы отправляем языковой модели. Сначала мы определяем шаблон запроса, куда можно подставить различные параметры. Например, если нам нужно чтобы модель отвечала, как оператор технической поддержки, мы добавляем системный промпт в запрос к LLM.

Вызовы — API LLM - это отправка сформированного запроса к серверу с языковой моделью. Мы используем специальный ключ доступа для аутентификации и отправляем наш промпт через защищенный канал связи. Сервер обрабатывает запрос и возвращает нам ответ в определенном формате.

Предварительная обработка ответов - это проверка полученного ответа перед его использованием. Мы проверяем, что ответ пришел в правильном формате, не содержит ошибок или неожиданных символов. Также проверяем, соответствует ли ответ нашим ожиданиям по структуре и содержанию. При необходимости проводим дополнительную очистку текста.

Работа с кэшем запросов - это сохранение полученных ответов для повторного использования. Когда мы получаем ответ от модели, мы сохраняем его в специальном хранилище вместе с исходным

запросом. Перед отправкой нового запроса мы проверяем, есть ли уже готовый ответ в кэше. Если да, то используем его, что позволяет сэкономить ресурсы и время, так как не нужно делать повторный запрос к модели.

Обработка ошибок – это механизм, который помогает корректно реагировать на возможные проблемы при работе с моделью. Если возникает ошибка подключения, проблема с форматом запроса или ответ от модели не соответствует ожиданиям, система фиксирует это и предпринимает действия для исправления ситуации. Например, может попытаться отправить запрос повторно или вернуть специальное сообщение об ошибке пользователю. При этом система всегда старается предотвратить аварийное завершение работы и сохранить стабильность приложения.

Все эти процессы работают последовательно: сначала формируется промпт, затем отправляется запрос, полученный ответ проверяется и обрабатывается, при необходимости используется кэш, а в случае проблем активируется система обработки ошибок. Такой подход обеспечивает надежную и эффективную работу с языковыми моделями.

Controller использует эту модель для взаимодействия с LLM, когда это необходимо для обработки пользовательского запроса. View при этом только отображает конечный результат, не участвуя в процессе взаимодействия с API.

Такой подход позволяет:

Сохранять принцип разделения ответственности

Легко заменять или добавлять новые LLM-модели

Не смешивать работу с LLM с традиционной бизнес-логикой

Обеспечивать централизованное управление ошибками и кэшированием.

Таким образом, Model — это фундамент вашего приложения, где сосредоточена вся логика работы с данными. Чем сильнее она отделена от View и Controller, тем проще поддерживать и масштабировать код. В prompt-инжиниринге понимание Model поможет вам лучше структурировать запросы, связанные с обработкой данных и бизнес-правилами.

6. Компонент View

View в паттерне MVC отвечает за отображение данных пользователю. Это то, что видит и с чем взаимодействует конечный пользователь — веб-страницы, интерфейсы, формы. Главная задача View — показать информацию, полученную от Model, в удобном формате, не вмешиваясь в логику обработки данных.

Во фреймворке Flask для работы с View используются шаблоны (templates). Шаблоны включают в себя HTML-разметку и Python-код в одном файле. Например, у вас может быть шаблон страницы блога, куда передается список статей из Model через Controller. В этом шаблоне вы можете циклом пройтись по статьям и вывести их заголовки и содержимое, не прописывая каждый раз HTML вручную.

Передача данных между Controller и View происходит через возвращаемые значения функций – обработчиков запросов. Controller берет данные из Model, подготавливает их и передает в шаблон. Например, во Flask это делается с помощью функции `render_template`,

куда вы передаете имя шаблона и переменные, которые должны в нем отобразиться.

Давайте рассмотрим пример. У нас есть Controller, который получает историю чата из Model. Этот Controller передает список в шаблон `index.html`, где мы выводим запросы пользователей и ответы LLM в виде списка. В шаблоне мы не пишем логику для получения данных — мы только отображаем то, что нам передали.

Таким образом, View — это мост между данными и пользователем. Чем чище и проще ваши шаблоны, тем легче их поддерживать и изменять. В prompt-инжиниринге понимание View поможет вам оптимизировать ответы с учетом того, как будет выглядеть результат запроса к LLM.

7. Компонент Controller

Controller в паттерне MVC выступает посредником между Model и View. Это мозг приложения, который обрабатывает действия пользователя, принимает решения и координирует работу других компонентов. Когда пользователь что-то делает — нажимает кнопку, отправляет форму или переходит по ссылке — Controller первым получает этот запрос и решает, что с ним делать дальше.

В веб-разработке Controller чаще всего работает с HTTP-запросами. Например, когда вы открываете главную страницу чата, браузер отправляет GET-запрос, и Controller определяет, какие данные нужно показать, запрашивает их у Model, а затем передает в View для отображения. Если же пользователь отправляет свой вопрос через

форму ввода текста, Controller обрабатывает POST-запрос, проверяет данные и решает, отправить их в LLM или вернуть ошибку.

Библиотека Flask очень упрощает создание контроллеров. Все что нам нужно – это маршрутизация, т.е. способ определить, какой Controller должен обрабатывать тот или иной запрос. Вы создаете функции-обработчики и связываете их с конкретными адресами с помощью декоратора `@app.route`. Например, функция home возвращает начальную страницу. Функция chat обрабатывает новое сообщение и привязана к пути `/chat `.

Взаимодействие между компонентами выглядит так: Controller получает запрос, запрашивает нужные данные у компонента Model, получает их в удобном формате и передает в View для отображения. При этом Controller не должен знать, как именно данные хранятся в Model или как они будут выглядеть в View — он только управляет процессом.

Понимание работы Controller поможет вам лучше проектировать запросы, связанные с обработкой действий пользователя и управлением потоком данных. Четкое разделение логики приложения на Model, View и Controller упрощает интеграцию LLM, избавляя prompt инженера от лишних и ненужных деталей.

8. Взаимодействие компонентов

Давайте разберем, как компоненты MVC взаимодействуют между собой при обработке запроса.

При первом обращении к главной странице HTTP-запрос попадает в Controller home. Он выступает как диспетчер — анализирует запрос,

определяет, какое действие нужно выполнить, и обращается к Model за данными. В нашем случае он попросит Model прочитать историю запросов для этого пользователя и сохранит ее в переменной chat_history.

Model получает запрос от Controller и работает исключительно с данными. Она не знает ничего о том, как эти данные будут отображаться. В нашем примере Model обращается к базе данных и вернет историю запросов, отсортированную по времени их создания. Затем Model сообщит Controller об успешном выполнении операции или вернет ошибку, если что-то пошло не так.

Controller, получив ответ от Model, принимает решение о дальнейших действиях. Если есть история запросов, Controller включит эти данные в шаблон index.html и вернет View как результат выполнения функции render_template.

Для интерактивного взаимодействия нам не нужно каждый раз перезагружать страницу целиком. У нас есть код на javaScript, который позволяет отправлять запросы на сервер в асинхронном режиме.

Если пользователь вводит сообщение и нажимает на кнопку Отправить, функция javaScript из браузера отправляет запрос на Controller chat. В этом месте происходит обращение к модели llm_service. Модель может подготовить prompt и включить в него соответствующее сообщение. После получения ответа от модели llm_service, Controller передаст его в View для отображения пользователю.

View получает от Controller готовые данные в том формате, который удобен для отображения. View не содержит сложной логики — только

шаблоны и правила визуализации. В нашем случае View получит ответ модели в формате json и добавит его на HTML-страницу, которую браузер уже показывает пользователю.

Для prompt-инжинира понимание этого взаимодействия поможет увидеть полную картину: получение данных (Model), их обработка (Controller) и представление результата (View). Это сделает вашу работу в команде более эффективной, позволит говорить с разработчиками на одном языке и поможет лучше понять свою роль по разработке оптимальных промптов.

9. Расширенные возможности

В веб-разработке на Flask вы часто столкнётесь с необходимостью работы с формами. Для удобства можно использовать расширение Flask-WTF, которое помогает создавать формы, валидировать данные и защищаться от CSRF-атак. Когда пользователь отправляет форму, Controller получает данные, проверяет их через Model на соответствие бизнес-правилам, а затем либо сохраняет, либо возвращает ошибки во View для отображения. Валидация происходит как на стороне клиента (для мгновенной обратной связи), так и на сервере (для безопасности).

Аутентификация и авторизация — важные аспекты многих приложений. Здесь Model отвечает за хранение пользователей и их ролей, Controller проверяет логины и пароли, а View показывает соответствующие формы и сообщения. Для упрощения можно использовать Flask-Login, который помогает управлять сессиями пользователей. Авторизация определяет, какие действия разрешены

пользователю — эти правила обычно прописываются в Controller перед выполнением важных операций.

Статические файлы (CSS, JavaScript, изображения) во Flask по умолчанию хранятся в папке static. View обращается к ним через специальный маршрут, но важно понимать, что это не часть MVC в классическом понимании — скорее вспомогательные ресурсы для оформления. В production-окружении статические файлы лучше отдавать через CDN или веб-сервер вроде Nginx.

При работе с внешними сервисами важно предусмотреть обработку ошибок — что делать, если API недоступен или вернул некорректный ответ. Эти сценарии должны обрабатываться в Model, а Controller уже решает, показать ли пользователю сообщение об ошибке или попробовать запрос снова. View в этом случае просто отображает конечный результат или статус операции, не зная деталей взаимодействия с API.

Для сложных промптов используйте паттерн Chain of Responsibility, где обработка проходит через несколько последовательных шагов. Это особенно полезно, когда вам нужно комбинировать результаты нескольких запросов к LLM или применять каскадные уточнения. Этот подход уже реализован в библиотеке LangChain, например.

Важно помнить про кэширование. LLM-запросы могут быть дорогими и медленными, поэтому предусмотрите в Model кэш для часто используемых промптов. Но не забывайте, что кэширование должно учитывать контекст — одинаковый промпт с разными параметрами может давать разные результаты.

Эти расширенные возможности показывают, как MVC масштабируется под сложные задачи. Главное — сохранять принцип разделения ответственностей: Model работает с данными и внешними сервисами, Controller управляет логикой приложения, а View фокусируется на отображении. В prompt-инжиниринге это поможет вам создавать сложные, но хорошо организованные запросы, где чётко определено, какая часть за что отвечает.

10. Альтернативные фреймворки и архитектуры

MVC является фундаментальным паттерном, но в разных экосистемах он реализован по-своему. В Python/Flask вы работаете с классическим подходом. В Django используется MTV (Model-Template-View), что, по сути, тот же MVC, но с переименованными слоями. В JavaScript-фреймворке вроде Angular MVC эволюционировал в компонентную архитектуру. Laravel - современный PHP-фреймворк с полной поддержкой MVC, имеет встроенную систему роутинга и аутентификации. В мире java фреймворк Spring использует DispatcherServlet как центральный компонент для обработки запросов. Поддерживает различные типы представлений: JSP, HTML, XML, JSON и другие.

Язык, фреймворк и архитектура для проекта выбирается с учетом масштаба и сложности проекта. Но фундаментальные принципы остаются неизменными. Хорошо если при внедрении LLM в MVC соблюдается принцип "LLM как сервис". Вся работа с языковыми моделями должна быть вынесена в отдельный слой модели, не смешиваясь с традиционной бизнес-логикой. Например, в отдельный

модуль LLMService, который отвечает за формирование промптов, вызовы API и предварительную обработку ответов. Это позволит вам легко заменять модели или добавлять новые, не переписывая весь код.

В современных реалиях MVC адаптируется под новые требования. Если вы работаете с большим количеством динамических данных и сложной бизнес-логикой, возможно, будете иметь дело с Clean Architecture или Hexagonal Architecture, где взаимодействие с LLM будет вынесено в отдельные адаптеры. Вы можете встретить комбинацию его с микросервисной архитектурой, вынос работы с LLM в отдельный сервис. Или использование Event-Driven подхода, где изменения данных через LLM инициируют события для других компонентов системы. Главное — сохранять ясное разделение ответственностей, даже когда ваше приложение активно взаимодействует с языковыми моделями.

Для prompt-инженеров понимание этих практик позволяет создавать более профессиональные решения. Когда вы проектируете промпты с учетом архитектурных принципов, их проще интегрировать в существующие системы, тестировать и модифицировать. Старайтесь проектировать промпты как независимые модули, которые четко соответствуют своей роли, это значительно упростит их сопровождение.

11. Заключение

Сегодня мы с вами подробно разобрали один из фундаментальных архитектурных паттернов разработки программного обеспечения

– **шаблон MVC** и его применение в контексте интеграции искусственного интеллекта.

Особое внимание было уделено интеграции сервиса **LLM** в архитектуру MVC, что позволяет создавать современные интеллектуальные приложения. Мы изучили, как Flask помогает реализовать этот паттерн на практике, а также разобрали механизмы взаимодействия между компонентами системы.

Важно отметить, что понимание MVC и принципов его работы – это фундамент для разработки масштабируемых и поддерживаемых приложений. В ходе лекции мы также рассмотрели расширенные возможности архитектуры и ознакомились с альтернативными фреймворками, что дает вам представление о различных подходах к организации кода.

Помните, что выбор архитектуры – это всегда компромисс между требованиями проекта, сроками и ресурсами. Но именно MVC является проверенным временем решением.

Практическое применение полученных знаний поможет вам понимать, как работают, а в будущем и создавать более качественные и профессиональные решения. Рекомендую продолжить изучение материала, экспериментировать с различными моделями и промптами, чтобы сформировать собственное видение оптимального подхода к разработке.

Благодарю за внимание!