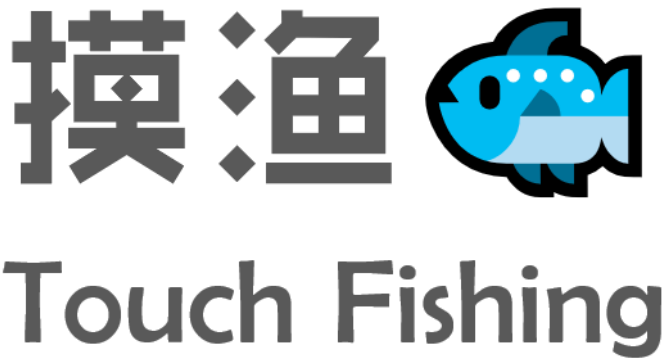
软件架构文档



2021 年 5 月

团队成员

队长　宁子淳

柯俊哲

樊伟哲

梁浩

郑继凯

指导教师：余仲星

版本：1.0

小组分工

内容	修订人	最后更改日期
1. 引言	樊伟哲	2021.5.28
2. 架构背景	樊伟哲	2021.5.28
3.1 逻辑视图	梁浩	2021.5.28
3.2 开发视图	郑继凯	2021.5.28
3.3 运行视图	柯俊哲	2021.5.28
3.4 部署视图	宁子淳	2021.5.28
4. 总体设计	樊伟哲	2021.5.28

Contents

1. 引言

1.1 目的及范围

该架构文档对实时定位系统整体结构进行描述，其中会使用多种不同的架构视图来描述系统的各个方面。它用于记录并表述已对系统的架构方面作出的重要决策。文档适用于摸渔线上购物平台，开发方为摸渔团队成员。

1.2 文档结构

文档的组织结构如下：

- 第一部分 引言
  本部分主要概述了文档内容组织结构，使读者能够对文档内容进行整体了解，并快速找到自己感兴趣的内容。同时，也向读者提供了架构交流所采用的视图信息。
- 第二部分 架构背景
  本部分主要介绍了软件架构的背景，向读者提供系统概览，建立开发的相关上下文和目标。分析架构所考虑的约束和影响，并介绍了架构中所使用的主要设计方法，包括架构评估和验证等。
- 第三、第四部分 视图及其之间的关系
  视图描述了架构元素及其之间的关系，表达了视图的关注点、一种或多种结构。
- 第五部分 需求与架构之间的映射
  描述系统功能和质量属性需求与架构之间的映射关系。
- 第六部分 附录
  提供了架构元素的索引，同时包括了术语表、缩略语表。

1.3 视图编档说明

所有的架构视图都按照标准视图模板中的同一种结构进行编档。

1.4 标识

基于 Html 的线上购物平台

摸渔

版本号：1.0.0

保密级别：普通

1.5 系统概述

根据课程要求，建立软件工程初步项目。项目中要求实现线上购物系统的基本功能，包括系统管理员、商家、顾客等相关用户的基本需求，以建立一个功能健全，可用性强，安全性高的网络购物平台。

实现环境为 html 环境，完全架构在服务器上的浏览器运行模式，同时包括 PC 端和手机端两部分，实现两个平台同时可以在线运行。

1.6 引用文件

Software Engineering 4th Edition TRANSLATED 等

2. 架构背景

2.1 系统概述

当下电子商务平台的不断发展，越来越多的人对于网上购物，直播带货等多种多样的买卖方式产生了兴趣，同时各类购物平台不断开发升级自己的购物系统，发展出个人或者企业独立的网上门店，各种各样的平台样式不断出现，但是归结到核心内容底层的开发还是比较相似的。因此本团队致力于打造自己的购物平台，深度了解购物平台底层的建设。

2.2 架构需求

2.2.1 技术环境需求

前端使用 HTML+JavaScript 开发，后端使用 Django 框架开发，使用 SQLite 作数据库。

2.2.2 功能需求

实现购物系统的基本功能，包括管理用户信息，管商品信息，管理订单信息等，基本上实现主流购物平台支持的相关功能，保证用户的最佳体验。

2.2.3 质量属性需求

系统应支持 100 人以上同时访问服务器并支持 500 人以上同时访问数据库，服务器的响应时间不应该超过 5 秒。

所有用户在保证网络连接的情况下可同时通过局域网和互联网访问系统。

系统必须保证数据的安全访问，用户需要通过用户名和密码进行身份认证，同时对数据的访问要进行授权认证。

2.3 主要设计决策及原理

为了实现用户可以互联网访问系统，程序部署在公有云上。用分阶段的生存周期计划进行严格的管理；坚持进行阶段评审；实行严格的产品控制；采用现代程序设计技术；项目的结果能够清楚的审查；开发小组的人员充分发挥各自优势特点；承认不断改进软件工程实践的必要性。

3. 视图

3.1 逻辑视图 (Logical View)

本章是对软件架构的逻辑视图的描述。主要描述系统的功能需求，即系统应该为用户服务提供的功能。

3.1.1 顶层逻辑视图

3.1.1.1 主表示

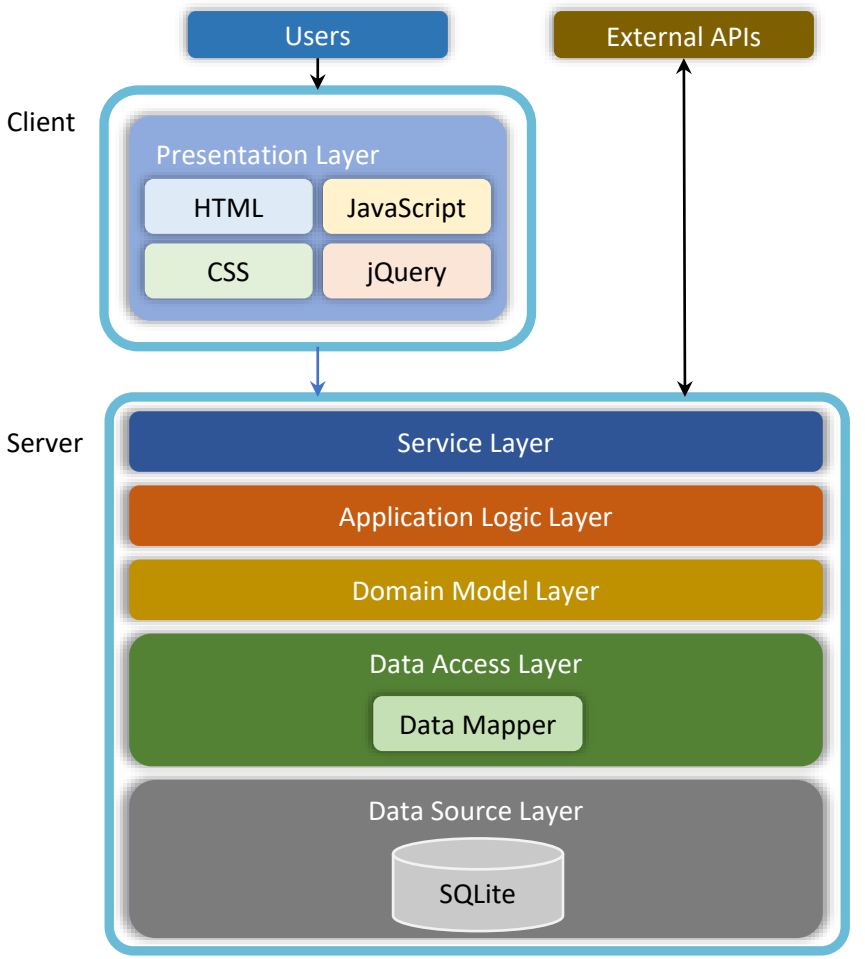


图 TouchFishing 系统的分层视图

3.1.1.2 构件目录

A. 构件及其特性

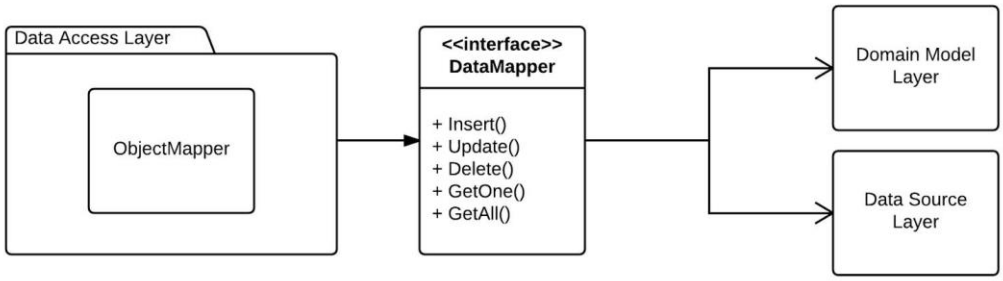
构件	*描述*
表示层	表示层将为用户提供与系统交互的图形界面。它由 HTML、CSS、JavaScript 和其他相关文件组成，这些文件将在用户选择的 Web 浏览器中运行。此层的主要目标是提供用户完成系统中任务所需的一切功能。与可用性和前端设计相关的质量属性在表示层发挥最大作用。
应用逻辑层	应用逻辑层的作用是封装系统控制器，系统控制器实现了大部分核心业务逻辑。它还提供用户界面和业务对象之间的连接。
服务层	服务层负责模块化应用程序的不同功能(服务)。某些功能（如报告）需要访问外部 API;服务层负责与外部 API 进行通信。单个服务也会使用应用逻辑层执行核心业务逻辑。 每个服务都是一个立面，作为演示层可以访问的功能的一般接口。这些接口将提供操作或修复每个服务下方代码的能力，而不会影响其他层调用该服务的方式。 系统内服务的模块化让确定潜在缺陷更为容易，可维护性更好。每个服务都可以独立测试，达到了系统范围的可测试性。
业务对象层	业务对象模型包含系统中所有的系统数据表示对象，同时也包含这些对象相关的方法
数据访问层	数据访问层包含系统中的所有数据映射器。这些映射器负责协调所有业务对象层同相应的数据库表之间的通信。这确保了数据库及模式和 SQL 接口对业务对象层的隐藏
数据源层	数据源层由所有持久性信息和外部 API 集成构成。这包括数据库，以及该数据库包含的系统的所有数据。

B. 关系及其特性

构件	*描述*
表示层和应用逻辑层	表示层表示系统视图，应用逻辑层包含控制不同角色的逻辑的控制器。在遵循 MVC 架构模式时，控制器从视图中获取信息并用它来修改或请求对象中的相关数据，防止视图直接修改对象，使视图只是显示数据的变化。
应用逻辑层到业务对象层	应用逻辑层根据业务逻辑规则修改业务对象层中封装的数据，然后将数据返回给表示层展示给用户。
业务对象层到数据访问层到数据源层	业务对象层是存储在数据层中的信息的活跃表示。当数据发生变化时，业务对象层通过数据访问层将更新推至数据层并创建持久的数据备份。数据访问层和数据源层之间的所有通信都根据 ACID 属性间接进行 SQL 查询实现。

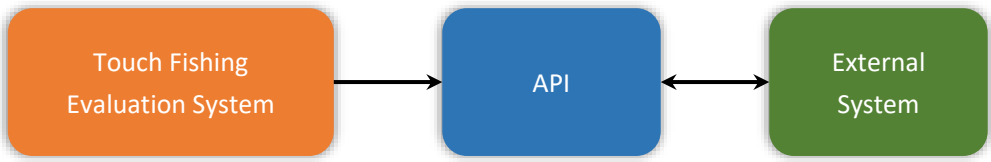
C. 元素接口　界面标识

数据映射器充当持久性层和业务对象层 (对象模型) 之间的接口。它以数据访问层的形式显示在分层试图中。它将业务对象层中的对象转换为数据库中格式化的数据。



数据映射器接口示意图

该系统还与外部 API 连接，用于表单生成等。这些 API 提供了一套使用其工具的方法，这些方法组成了其系统和我们的系统之间的接口。



外部 API 接口示意图

3.2 开发视图（实现视图，Implementation View)

开发视图由代表软件实现部分的图组成，这是定义软件内部工作流程的唯一视图，展示代码的组织和执行，主要用来描述系统的主要功能模块和各个模块之间的关系，主要被开发人员使用。

3.2.1 顶层开发者视图

3.2.1.1 主表示

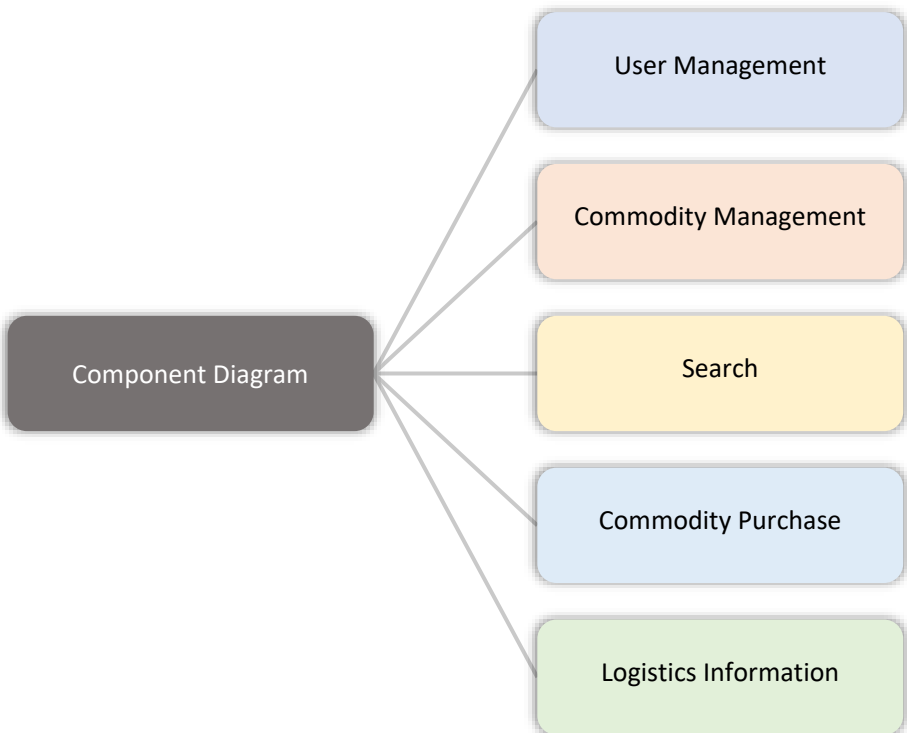


图 3.2.1 顶层开发视图

3.2.1.2 构件目录

A. 构件及其特性

构件	*描述*
用户管理	包括用户注册，用户信息查看及修改，注册成为商家等功能。
商品管理	商家用户可以在自己的商铺中完成商品添加，商品下架，商品信息修改等功能。
搜索功能	用户可以在商品页面搜索店铺和商品信息，快速找到所需的商品，商家用户也可以通过搜索功能更加方便的管理商品。
商品购买	在找到心仪的商品之后，用户可以通过商品购买功能，完成对某个商品的购买，并生成订单。
物流信息管理	订单生成之后，系统会为这个订单生成物流管理信息，商家（或物流公司）可以修改物流信息，同时用户可以随时查看物流信息。

B. 关系及其特性

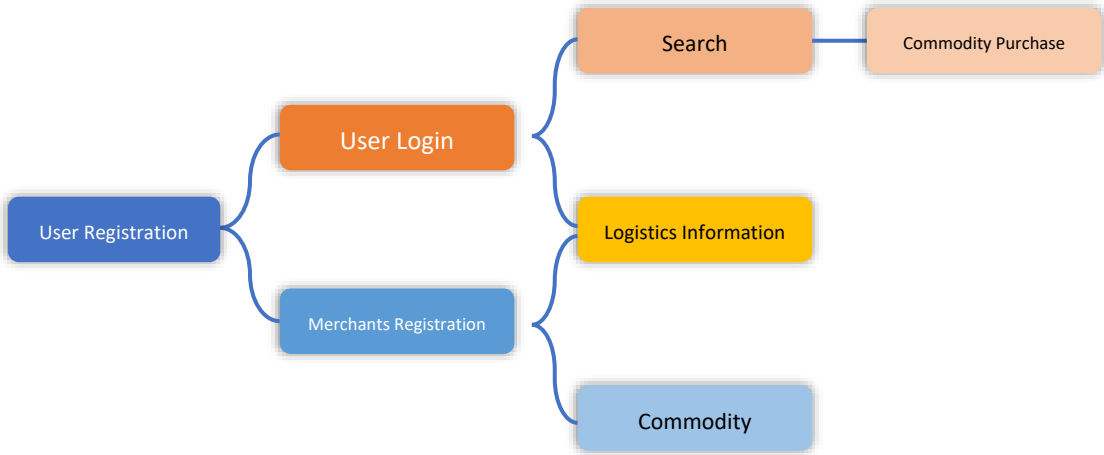
使用关系:

作为该软件的使用者，普通用户以及商家用户对不同的构件有相应的使用关系。普通用户可以使用商品浏览页面、商品搜索功能来寻找心仪的商品，通过商品购买功能购买商品，通过物流信息管理功能查看物流信息；商家用户可以通过商品管理功能实现商品添加、下架以及信息修改等。

C. 元素接口

作为实现上述功能的主要结构，后段需要与数据库连接，提供模糊搜索（关键字信息查询)、用户信息、商品信息、物流状态信息的查询与修改。

3.2.1.3 上下文图



3.2.1.4 可变性

对于一些特殊的要求，可以新增一些功能细节。例如普通用户申请成为商家的审核机制，物流公司对物流信息的管理等。

3.2.1.5 原理

实现视图通过系统输入输出关系的模型图和子系统图来描述。要考虑软件的内部需求：开发的难易程度、重用的可能性，通用性，局限性等等。开发视图的风格通常是层次结构，层次越低，通用性越好。

3.3 运行视图（过程视图，Process View)

侧重系统的运行特性，关注非功能性的需求(比如性能、可用性等)，服务于系统集成人员，方便后续性能则试。

3.3.1 顶层运行视图

3.3.1.1 主表示

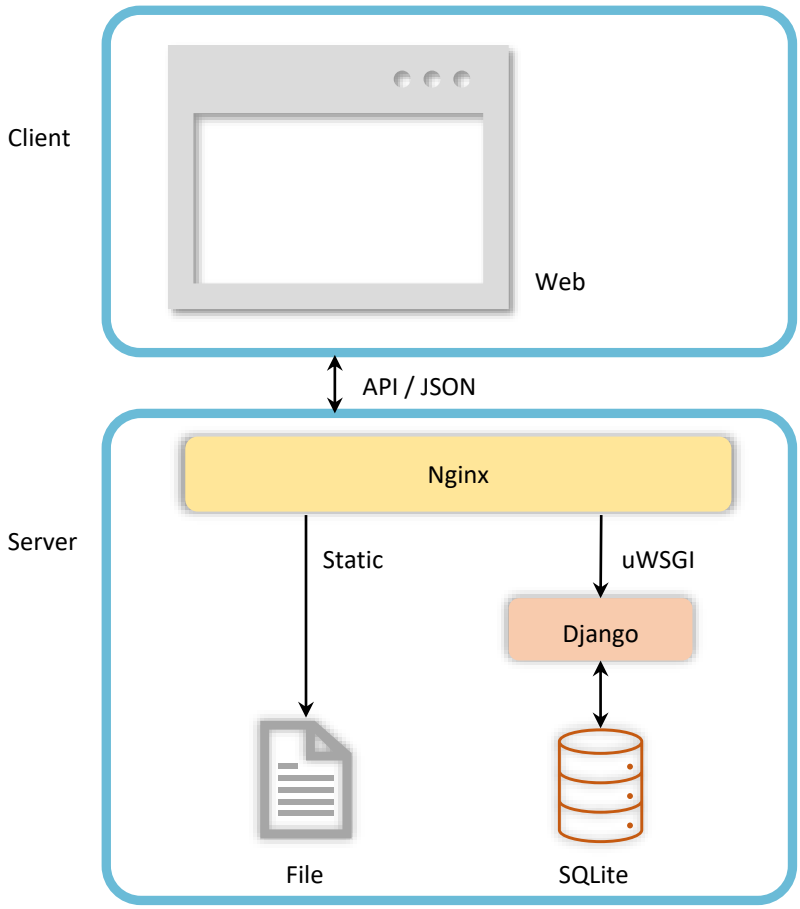


图 XXX TouchFishing 系统运行视图

3.3.1.2 构件目录

A. 构件及其特性

构件	*描述*
客户端	客户端是调用服务器服务的组件。客户端的端口可以提供他们需要的服务。在我们这个摸渔购物系统中，客户端是用于访问摸渔后台系统的 Web 浏览器。客户端使用网络服务发出 HTTP 请求。

服务器	服务器是向客户提供服务的组件。服务器具有描述其所提供的服务的端口。摸渔购物系统部署在自己测试使用的服务器上。Nginx 用于实现特定后端页面，静态调用服务器文件，用 Django 连接后端 SQLite 数据库。服务器通过 TCP/IP 应用层协议 HTTP 提供 Web 服务。
请求/回复连接器	请求和回复连接器是使用 请求/回复协议的数据连接器，客户端使用该协议在服务器上调用服务。对于摸渔购物系统，客户端和服务器 Ajax 服务进行通信。客户端使用 HTTPS 协议发出请求，从服务器请求信息，服务器也使用 HTTPS 请求进行响应回复。服务器使用 Nginx、Django 框架和 SQLite 数据库进行通信，以便将 Ajax 通信数据包映射到数据库记录。此通信以 API 和 JSON 的形式进行。

B. 关系及其特性

连接关系：通过各构件之间通过不同的通信协议互相连接。

客户端到服务器：

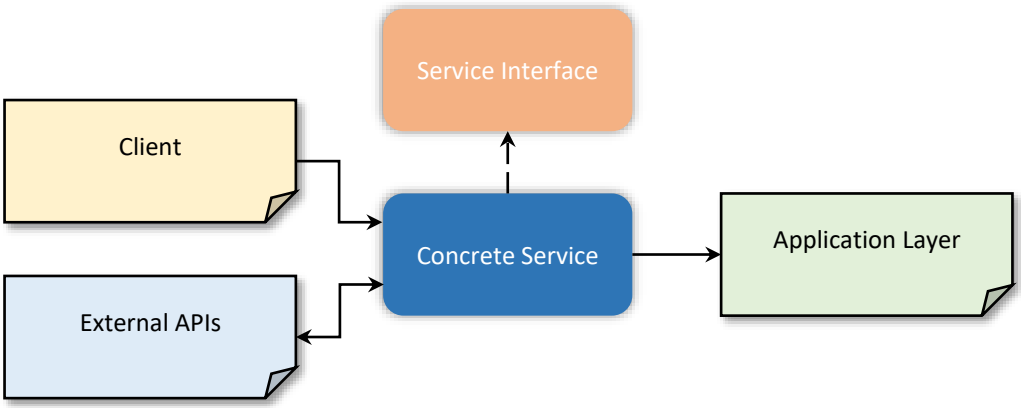
客户端和服务器使用请求响应消息模式相互通信。客户端向服务器发送请求，服务器发送响应作为回报。客户端和服务器使用 Ajax 进行通信，传输数据格式一般为 JSON，服务器给客户端开通通信接口，以便客户端和服务器进行同步更新。

要同时处理多个请求，服务器使用调度系统对来自客户端的传入请求进行优先级。服务器还限制客户端如何使用服务器的资源，以防止服务攻击。

C. 构件接口

界面标识

客户端和外部 API 都通过服务层与服务器接口。定义各种服务，以便用户界面和外部 API 可以 交换进出，而不必修改应用程序的业务逻辑。通过定义每个服务的通用界面，用户界面和外部 API 与系统的通信保持一致，并且易于修改。



D. 构件行为

由于用户与 Web 浏览器的交互，客户端将数据发送到服务器进行处理。例如，客户端可能会将评估提交给服务器的数据，然后相应地处理和存储这些数据。当 客户端从服务器请求数据时，服务器会响应要显示给用户的数据。此数据以 JSON 的形式发送，然后转换为客户端期望的格式，以便将其显示在用户界面中。

进出客户端模型的另一个数据流是从服务器到数据库，从数据库到服务器。服务器在数据库上执行操作以请求数据，然后服务器会响应请求的数据，转化为 JSON 等格式。

3.3.1.3 上下文图

见 3.3.1.1 主表示。

3.3.1.4 可变性

客户端也可以使用将 Web 封装成 APP，更加方便于移动端客户的使用，比如 PWA (Progressive Web Apps)技术，但目前技术有限，精力有限，时间有限，暂不考虑。

3.3.1.5 模型原理

客户端-服务器模型是分布式操作中的一种常见结构模型。服务器充当一个中央集权系统，可以为许多客户端提供服务。此模型非常适合摸渔购物系统，具体原因见上述模型。系统本身就搭建在服务器上，用户

使用 Web 客户端通过网络访问系统。

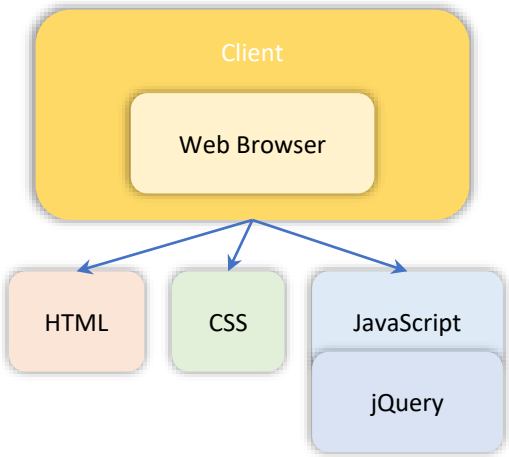
Web 客户的另一个好处是，它分离了服务器和客户端，也就是前端和后端。此外，客户端-服务器模型还允许在服务器端进行性能分析和负载平衡。

该模式也有一些缺点，比如服务器可能性能不足，无法处理大量请求，也可能发生故障。然而，现阶段还没上线，这不是该考虑的主要问题。

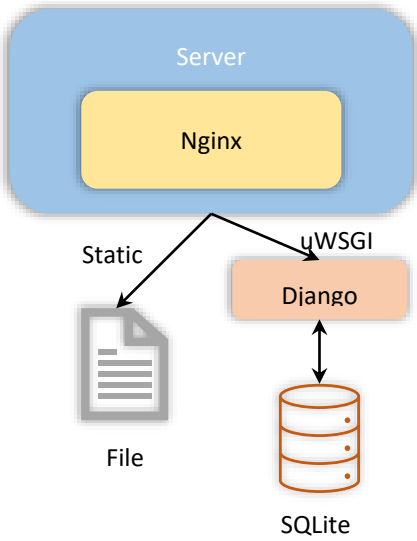
3.3.1.6 相关视图

子视图包：

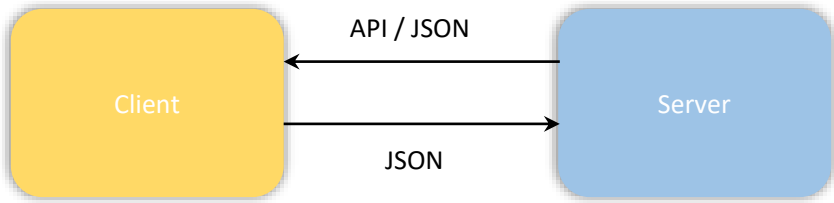
客户端：



服务器：



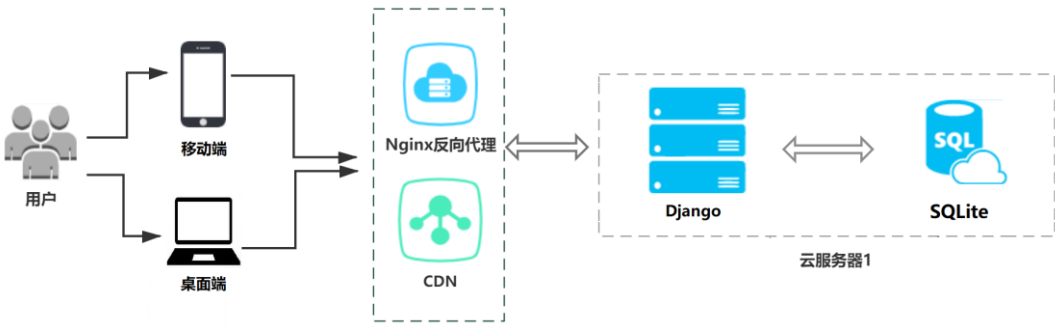
兄弟视图包：



3.4 部署视图（物理视图，Deployment View）

主要描述硬件配置，如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。

3.4.1 主表示



3.4.1.1 构件目录

A. 构件及其特性

构件	*描述*
Web 端	Web 端是用户与系统交互的可视化接口，在接口中，网页可以提供他们需要的服务。Web 端通过 HTTP 请求完成与服务器端的交互。
Nginx 服务器	Nginx 服务器接收来自 Web 客户端的 HTTP 请求，然后根据请求的类型进行分发，如果请求的是静态内容，那么直接返回相对应的静态文件。如果用户请求的是动态内容，则通过 uwsgi 接口，分发给 Django 处理。
CDN	内容分发网络，可以提升静态内容的下载速率，从而提升用户体验，降低服务器压力。
云服务器	Nginx 和 Django 程序部署的机器，为用户提供在公有网络下，软件服务的接入。
Django	开放源代码的 Web 应用框架，由 Python 写成。采用了 MTV 的框架模式。这里完成用户动态请求的处理，进行数据操作。
SQLite	轻量的数据库引擎，持续化保存程序所需要的数据。

B. 关系及其特性

Nginx 与 Django 的交互

WSGI 全称 Web Server Gateway Interface（Web 服务器网关接口），是一种描述 Web 服务器（如 nginx，uWSGI 等服务器）如何与 Web 应用程序（如用 Django、Flask 框架写的程序）通信的规范，是一种实现了比如 Python 解析的通用接口标准/标准，实现了 Python Web 程序与服务器交互的通用性，利用这个协议，Web 项目就可以轻松部署在不同的 Web Server 上了。

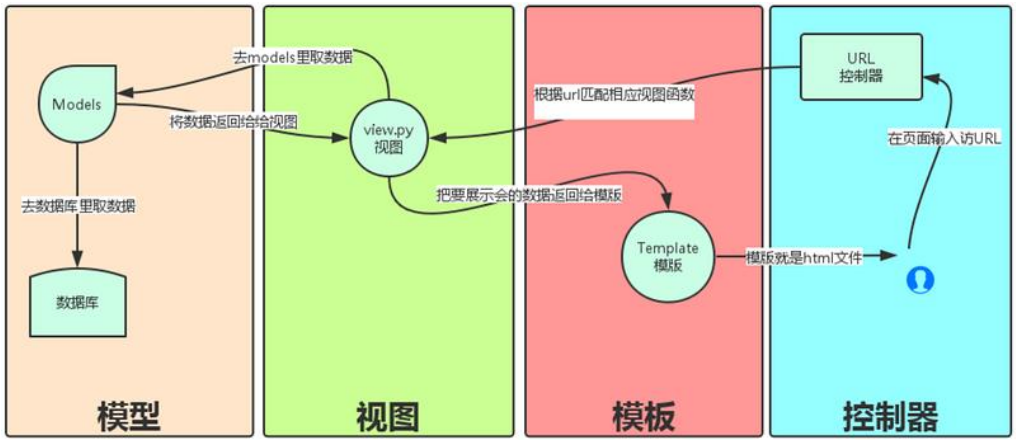
与 WSGI 一样，是 uWSGI 服务器的独占通信协议，用于定义传输信息的类型（type of information）。每一个 uwsgi packet 前 4 byte 为传输信息类型的描述，与 WSGI 协议是两种东西，据说该协议是 fcgi 协议的 10 倍快。

C. 构件行为

a. 首先客户端发送 http 请求，来获取服务器资源；
b. Nginx 作为直接对外的服务器接口，接收到客户端发过来的 http 请求，然后进行解包、分析；
c. 如果是静态资源的请求，会根据 nginx 配置的静态文件目录，返回请求的资源；
d. 如果是动态资源的请求，nginx 就会通过配置，将请求传递给 uWSGI 服务器；
e. uWSGI 将接收到的数据包进行处理，并转发给 WSGI（HTTP 协议转成 WSGI 协议）；
f. WSGI 根据请求调用 Django 项目中的某个文件或函数，进行逻辑处理，完成后 Django 将返回值交给 WSGI；
g. WSGI 将返回值打包，转发给 uWSGI 服务器（WSGI 协议转换成 HTTP 协议）；
h. uWSGI 服务器接收后转发给 Nginx 服务器，最终 Nginx 服务器将返回值返回给客户端（如浏览器）

注意：不同的组件之间传递信息涉及到数据格式和协议的转换

3.4.1.2 上下文图

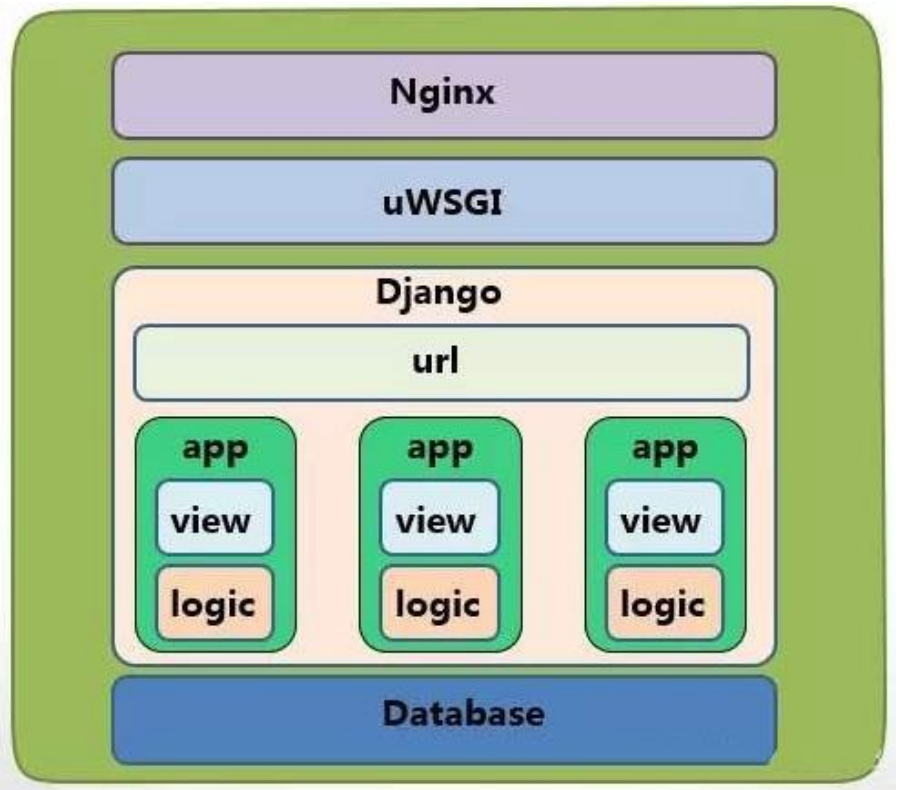


3.4.1.3 可变性

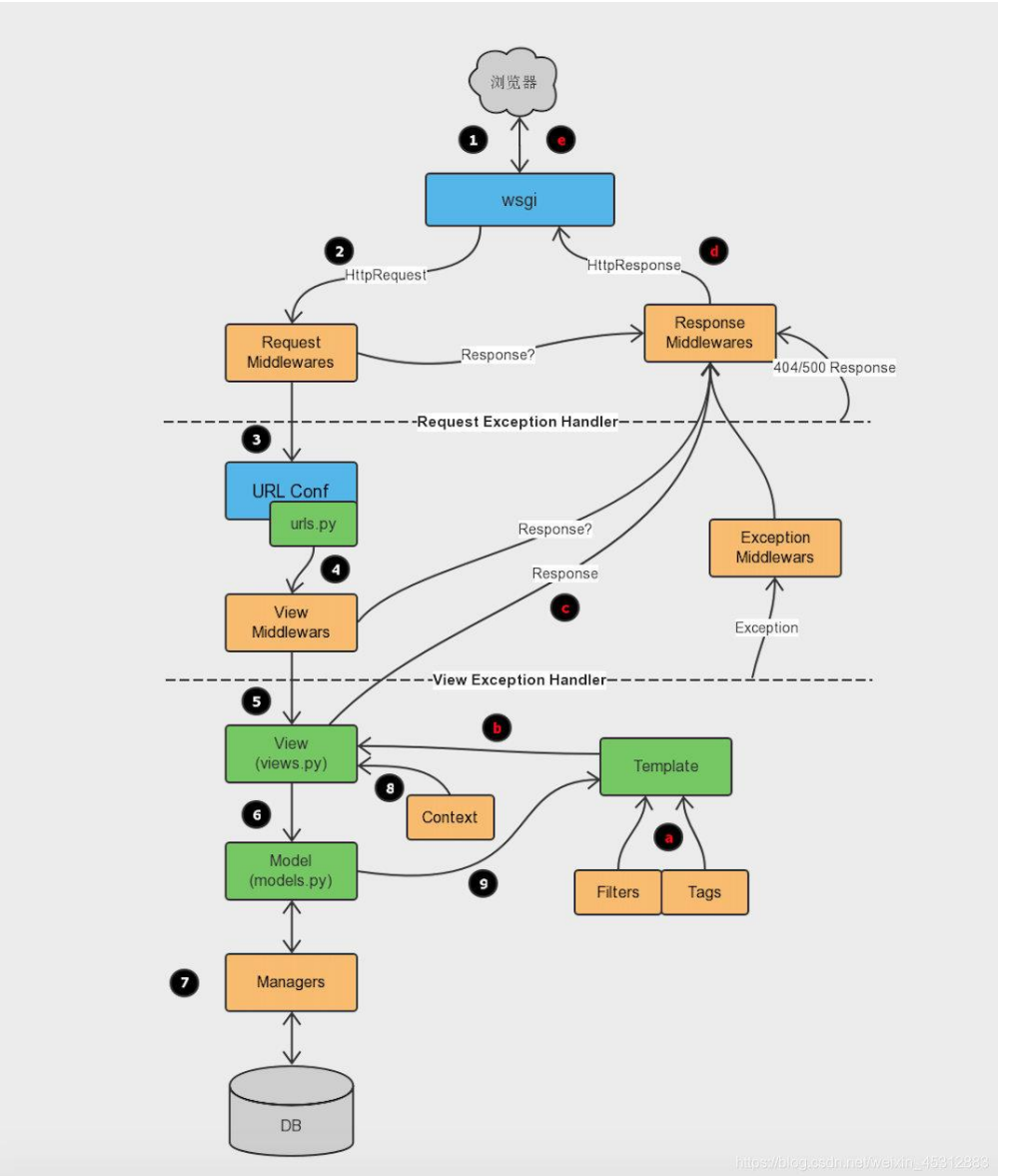
根据项目的需要，我们还可以采用不同的 Web 服务器，如 Apache2 等，也可以采用其他类型的关系型或非关系型数据库，如 MySQL、PostgreSQL、MongoDB 等。

3.4.1.4 原理

总体架构


Nginx
uWSGI
Django
url
app
view
logic
app
view
logic
app
view
logic
Database

Django 运行流程



4. 总体设计

4.1 基本功能

a) 用户登录：本用例的功能主要是用于确保用户在提供正确的验证信息之后，可以进一步使用本系统。

b) 商家：本用例的功能主要是新建，增加、删除、修改、查询店铺的信息。

c） 顾客：本用例的功能主要是查看、购买商品和查看订单信息。

d） 系统维护：本用例的功能主要是增加、禁用、修改、查询系统的用户资料。

e） 导出报表：本用例的功能主要是导出当前店铺、商品，订单信息，便于管理。

4.2 附加功能

a） 客户账号管理功能：管理人员能够列表显示用户的帐号信息、禁用或删除选定的用户帐号，在收到客户请求之后也可更改客户的账号信息。

b） 通知功能：管理人员可通过软件内消息向用户发送通知。

c） 电子邮件/短信服务：管理人员能够在客户账号信息出现异常时通过绑定的电子邮箱/手机号码向用户发送邮件/短信提醒。

5. 需求与架构之间的映射

见第三部分各视图。

6. 附录

6.1 术语表

HTML
JavaScript
CSS
jQuery
Nginx
Django
SQLite
API
uWSGI
JSON

附:

针对本组的 SAD，写出为保证项目的质量属性，采取什么策略在自己项目上增加相应的设计?

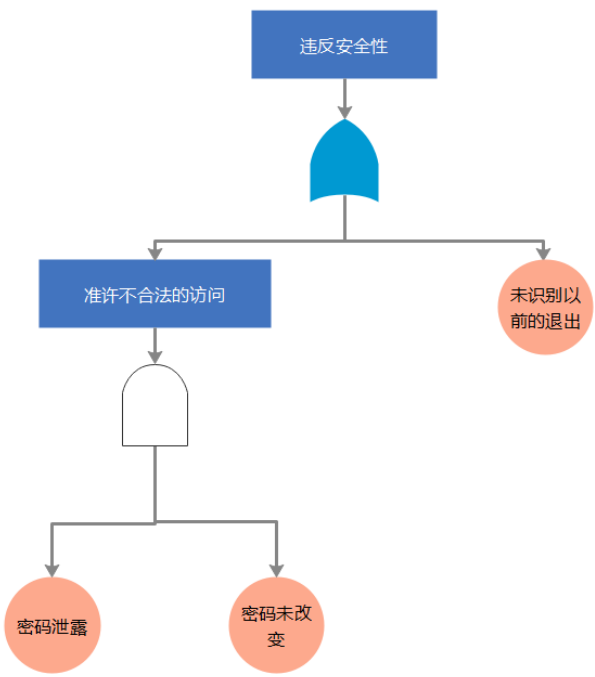
采用 Nginx 高性能 Web 服务器，相比直接使用 Python 的简单 HTTP 服务器，拥有更好的并发处理性能，能够根据 URL 进行静态内容分发及动态内容负载均衡，并且更易进行安全权限配置。

另外，本项目中不少组件是可以替换的，如可以采用不同的 Web 服务器实现不同的特性，根据实际的需求使用不同的后端数据库。

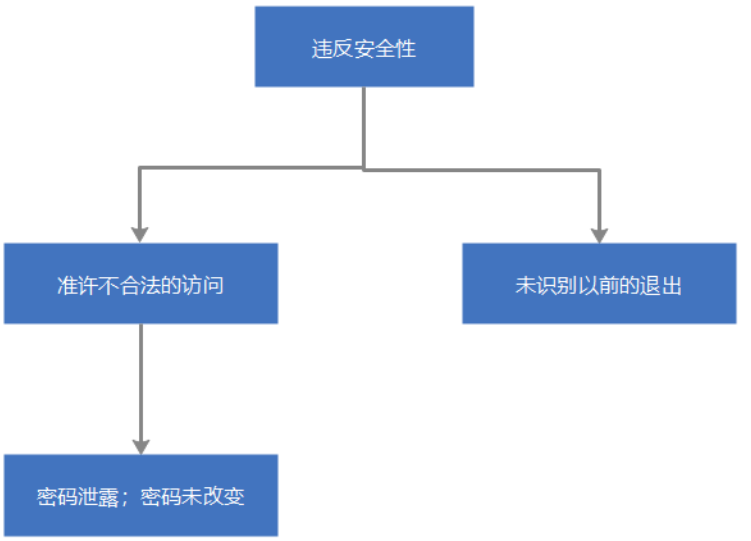
根据项目需要，后续我们可以编写一键部署脚本，将程序便捷地部署到服务器上。同时，我们可以探索应用 Docker 化及数据库分布式，从而完成服务的弹性扩展及异地容灾备份，提升服务的可用性及数据的安全性。

第五章课后习题 14，故障树转割集树练习。

故障树：



割集树：



学习、检索课本 5.17 参考文献及以下推荐的参考书或网上检索新的有关软件体系结构的资料。小组分工，每位成员选择自己关注的部分专题学习并写出学习报告（笔记）

分工：

柯俊哲：Forms and Controls

宁子淳：Model View Controller

梁浩：VisualWorks Application Model

郑继凯：Model-View-Presenter (MVP)

樊伟哲：Humble View

GUI 架构学习笔记

Definition of *Software Architecture*

网上查阅资料，搜到了这篇文档(Software Architecture Guide)，文档作者对软件架构说明了 **3** 个定义：

> - **a better view of architecture was the shared understanding that the expert developers have of the system design.**
> - **"the design decisions that need to be made early in a project", but Ralph complained about this too, saying that it was more like the decisions you wish you could get right early in a project.**
> - **His conclusion was that "Architecture is about the important stuff. Whatever that is".**

也就是说软件架构没有太明确的定义，有的人认为是专家对整个系统的设计，也有观点认为就是整个项目开始前这些设计理念的设计者认为的该有的"打开方式"，结论就是：不管是啥，就是跟重要的事情有关的东西。

我也就理解为 **something important you should decide and design before the whole project, whatever comes to your mind.**

在文档下面有个关于 GUI Architectures 的链接，虽然说 **2006** 年的可能有点老，毕竟 **User Interface** 的设计和审美随时间会不断变换，但是这篇文章不讲美观，只讲架构，还是值得深入学习的。

GUI Architectures

文章提到了几种不同的 **UI** 架构，最简单的先从表格入手。

表单/表格

结构

> **The form contains two main responsibilities:**
>
> - **Screen layout: defining the arrangement of the controls on the screen, together with their hierarchic structure with one other.**
> - **Form logic: behavior that cannot be easily programmed into the controls themselves.**

表格包含 **2** 个部分：屏幕布局和表格逻辑。

数据备份

而控件中显示了数据。然后讲到了 **SQL** 数据备份：

> **In most situations there are three copies of the data involved:**
>
> - **One copy of data lies in the database itself. This copy is the lasting record of the data, so I call it the record state. The record state is usually shared and visible to multiple people via various mechanisms.**
> - **A further copy lies inside in-memory Record Sets within the application. Most client-server environments provided tools which made this easy to do. This data was only relevant for one particular session between the application and the database, so I call it session state. Essentially this provides a temporary local version of the data that the user works on until they save, or commit it, back to the database - at which point it merges with the record state. I won't worry about the issues around coordinating record state and session state here: I did go into various techniques in [P of EAA].**
> - **The final copy lies inside the GUI components themselves. This, strictly, is the data they see on the screen, hence I call it the screen state. It is important to the UI how screen state and session state are kept synchronized.**

数据绑定

然后提到了保持前端界面状态和同步更新，以此用到的工具就是 数据绑定 **(Data Binding)**。就是说任何更新都应该立即同步。

然后就开始讲数据绑定的细节：

> **In general data binding gets tricky because if you have to avoid cycles where a change to the control, changes the record set, which updates the control, which updates the record set.... The flow of usage helps avoid these - we load from the session state to the screen when the screen is opened, after that any changes to the screen state propagate back to the session state. It's unusual for the session state to be updated directly once the screen is up. As a result data binding might not be entirely bi-directional - just confined to initial upload and then propagating changes from the controls to the session state.**

客户端服务器工具包

表格的任一字段的改变都应该被提醒，然后就提到了 **client-server toolkits** 。读完之后就是在说 **event listener**。

Diagram

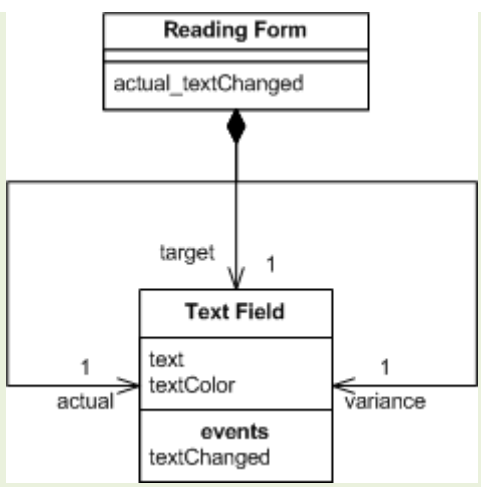


Figure 2: Class diagram for forms and controls

The text field declares an event for text changed, when the form assembles the screen during initialization it subscribes itself to that event, binding it a method on itself - here actual_textChanged.

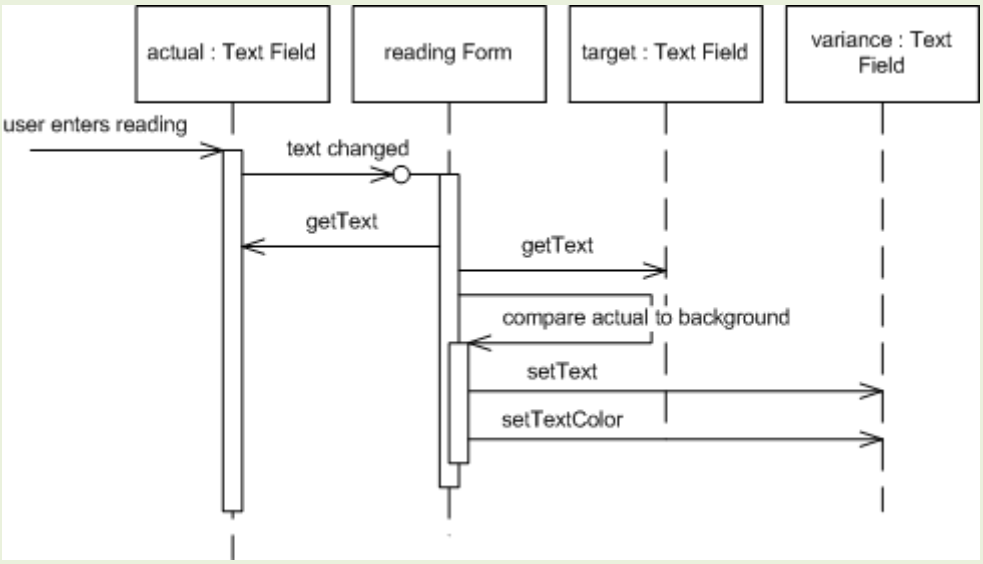


Figure 3: Sequence diagram for changing a genre with forms and controls.

When the user changes the actual value, the text field control raises its event and through the magic of framework binding the actual_textChanged is run. This method gets the text from the actual and target text fields, does the subtraction, and puts the value into the variance field. It also figures out what color the value should be displayed with and adjusts the text color appropriately.

发生事件后的各种细节，图表显示更清晰。

总结

- **Developers write application specific forms that use generic controls.**
- **The form describes the layout of controls on it.**
- **The form observes the controls and has handler methods to react to interesting events raised by the controls.**
- **Simple data edits are handled through data binding.**

> - **Complex changes are done in the form's event handling methods.**

模型视图控制器 Model View Controller (MVC)

经典 **MVC** 模式中，**M** 是指业务模型，**V** 是指用户界面，**C** 则是控制器，使用 **MVC** 的目的是将 **M** 和 **V** 的实现代码分离，从而使同一个程序可以使用不同的表现形式。**MVC** 的核心思想是分离演示。

V 即 **View** 视图是指用户看到并与之交互的界面。比如由 **html** 元素组成的网页界面，或者软件的客户端界面。**MVC** 的好处之一在于它能为应用程序处理很多不同的视图。在视图中其实没有真正的处理发生，它只是作为一种输出数据并允许用户操作的方式。

M 即 **model** 模型是指模型表示业务规则。在 **MVC** 的三个部件中，模型拥有最多的处理任务。被模型返回的数据是中立的，模型与数据格式无关，这样一个模型能为多个视图提供数据，由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。

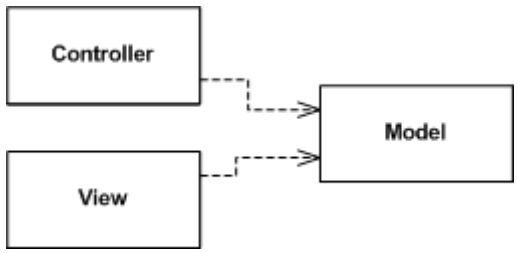
C 即 **controller** 控制器是指控制器接受用户的输入并调用模型和视图去完成用户的需求，控制器本身不输出任何东西和做任何处理。它只是接收请求并决定调用哪个模型构件去处理请求，然后再确定用哪个视图来显示返回的数据。

分离演示

MVC 中最具有影响力的框架，分离演示 **(Separated Presentation)**，明确划分了模拟我们对真实世界感知的**域对象(Domain Objects)**和我们在屏幕上看到的 **GUI** 元素的**演示对象(Presentation Objects)**。

域对象也称之为模型**(Model)**对象，完全忽视 **UI** 。

MVC 的表示部分由 **view**（视图） 和 **controller**（控制器） 两个部分组成



视图和控制器之间的基本依赖关系

流同步和观察者同步

> **These two styles I describe as patterns: Flow Synchronization and Observer Synchronization.**

这两种模式描述了在屏幕状态和会话状态之间进行同步触发的替代方式。表单和控件通过应用程序流来操作需要直接更新的各种控件。**MVC** 通过对模型进行更新，然后依靠观察者关系来更新观察模型的视图。

> **These two patterns describe alternative ways of handling the triggering of synchronization between screen state and session state. Forms and Controls do it through the flow of the application manipulating the various controls that need to be updated directly. MVC does it by making updates on the model and then relying of the observer relationship to update the views that are observing that model.**

总结

> - **Make a strong separation between presentation (view & controller) and domain (model) - Separated Presentation.**

- 将视图与控制器和模型分开。

> - **Divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should (mostly) not communicate directly but through the model.**

- 将用户图形交互组件拆分成为单一的控制器和视图。其中，控制器和视图只应通过模型来通信。

> - **Have views (and controllers) observe the model to allow multiple widgets to update without needed to communicate directly - Observer Synchronization.**

- 运行视图和控制器观测模型，从而使得组件更新无需直接通信。

视觉工作的应用模型 VisualWorks Application Model

VisualWorks 提出了一种应用模型的结构，一种类似于演示模型的结构，以此解决 **MVC** 难以应对视图逻辑和视图状态的难题。

解决问题的关键是把属性转变成对象。对于通常的带有属性的对象，我们可能会认为它有名字和地址等属性。当我们访问该对象的某属性时，如果不使用将属性看作对象的思想，我们需要使用类似 **obj.property** 的写法来获得值。若将属性变成对象，那我们可以通过属性返回一个封装实际值的对象，然后通过这个对象获取实际值。这种方法使得部件和模型间的映射变得更为简单。我们只需要告诉部件发送什么样的信息来获取相应属性，部件通过访问获得的对象来获取正确的值。使用应用模型和经典 **MVC** 的主要区别在于，我们现在在业务对象类和部件之间有一个中间类 ： 应用模型类。小部件不会直接访问业务对象。

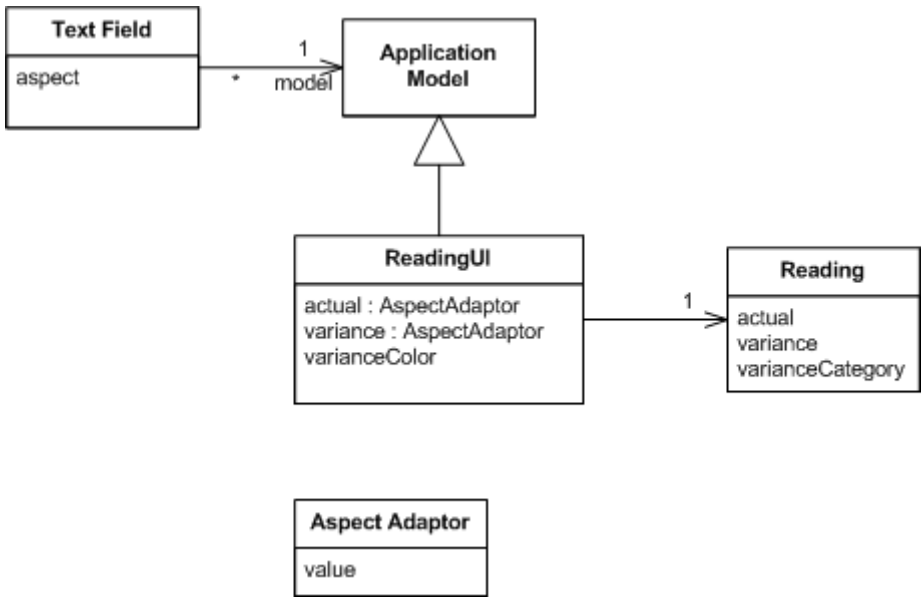


Figure 9: Class diagram for visual works application model on the running example

应用模型的特点如下：

和传统 **MVC** 一样使用展示分离和观察同步的方法

引入中间应用模型作为展示逻辑和状态的载体（展示模型的缩减版）

部件不直接访问业务对象模型，它们通过应用模型来间接访问。

使用属性对象连接不同层，并使用观察器支持粒度同步

模型视图演示者 Model-View-Presenter (MVP)

MVP 是一种架构，最早出现在 **IBM** 中，在 **1990** 年代的 **Taligent** 中更明显。最早通过 **Potel** 论文提及，这个想法被 **Dolphin Smalltalk** 的开发者进一步推广和描述。正如我们将看到的，这两种描述并不完全是相互联系的，但其基本思想已变得很流行。

在 **MVP** 模式中，**Model** 负责实现业务逻辑，保存数据，状态信息，**Model** 仅仅向 **Presenter** 提供一组服务接口，隐藏了内部实现的细节；**View** 负责与用户交互，它接收用户的操作、输入信息，与 **Presenter** 交互获得数据，展示给用户；**Presenter** 是 **View** 和 **Model** 的中间层，它接收来自 **View** 的输入，并将其传递给 **Model**，然后依据处理结果更新 **View**。下图演示了 **Model**，**View**，**Presenter** 三者的交互逻辑。

对比 **MVP** 和 **MVC**：

- 两者之间主要的区别是其实现方式和偶尔有些情况下需要同时使用 **Presenter** 和 **Controller**。
- 在 **MVP** 模式中，**View** 和 **Model** 之间是松耦合的，**Presenter** 负责将 **Model** 绑定到 **View**。通常情况下，**View** 和 **Presenter** 是一对一的关系，复杂的 **View** 可能有多个 **Presenter**。
- 在 **MPC** 模式中，**Controller** 是基于操作的，能够在 **View** 之间共享。**Controller** 负责决定显示哪个 **View**。

总结

- 在 **MVP** 模式中，接口里声明的事件和控件都是要在 **Presenter** 里要处理窗体中的信息。重要的是窗体必须实现 **IView** 接口并且必须 **New** 一个 **P**，把自身作为参数传到 **P** 里，这样在 **P** 里就可以利用多态访问窗体的成员了。并且重点是在窗体里我们可以利用委托或其他技术，把对用户输入输出、事件的响应，全部放到 **P** 里处理。因为 **P** 不知道窗体，只知道 **IView**，所以我们可以建立多个不同的窗体来对应一个 **P** 了，只要他们的业务逻辑、事件处理相同即可。
- 如果能够很好的利用 **MVP** 来编程，则窗体将变得非常简单,甚至可以让毫无经验的编码人员来负责窗体的 **UI** 设计等，十分方便。

简易视图（Humble View）

当人们谈论自测试代码时，用户界面很是一个比较突出的问题。许多人发现测试 **GUI** 介于艰难和不可能之间。这在很大程度上是因为 **UI** 紧密耦合到整个 **UI** 环境中，很难将 **UI** 分解并进行分段测试。

通过创建小部件并在测试代码中进行操作，可以取得不错的结果，但是在某些情况下，这是不可能实现的，可能会错过重要的交互，存在线程问题，测试运行的速度也会变慢。

因此，在设计 **UI** 的过程中，已经有了一个稳定的过程，使测试时遇到的对象中的行为最小化。迈克尔-弗斯在**[The Humble Dialog Box]** 将这一概念推广到**简易对象**-任何难以测试的物体都应有最低限度的行为。这样，如果我们不能将它包含在测试套件中，我们就尽量减少未被检测到的失败的可能性。

演示者不仅决定如何对用户事件作出反应，而且还处理 **UI** 小部件本身中的数据填充。因此，小部件不再具有模型的可见性，也不再需要可见性；它们形成一个**[被动视点]**被当前用户操纵。另一种方法是使用**[表示模型]**，尽管在小部件中确实需要更多的行为，但足够让小部件知道如何将自己映射到**[表示模型]**。这两种方法的关键是，通过测试演示者或测试演示模型，可以测试 **UI** 的大部分风险，而不必接触到难以测试的小部件。

> **The Humble Dialog Box paper uses a presenter, but in a much deeper way than the original MVP. Not just does the presenter decide how to react to user events, it also handles the population of data in the UI widgets themselves. As a result the widgets no longer have, nor need, visibility to the model; they form a Passive View, manipulated by the presenter.**
>
> **This isn't the only way to make the UI humble. Another approach is to use Presentation Model, although then you do need a bit more behavior in the widgets, enough for the widgets to know how to map themselves to the Presentation Model.**

带着表示模型通过让所有的实际决策都由表示模型。所有用户事件和显示逻辑被路由到表示模型，所以小部件所要做的就是将自己映射到表示模型。然后，可以测试表示模型没有任何小部件存在-唯一的风险在于小部件映射。只要这很简单，就能不去测试它。在这种情况下，屏幕并不像被动视点接近，但差别很小。

自被动视点使小部件变得比较单一，甚至没有映射，被动视点消除即使是小的风险表示模型。然而，代价是需要在测试运行过程中模拟屏幕--这是需要构建的额外机器。