**软件架构文档**

<"Testland"在线评测平台>

软件架构文档

版本 <1.0>

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| <26/6/2023> | <1.0> | <初步完成软件架构设计> | <张奕涵/胡彤/全雨乐/杨菡雪> |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

软件架构文档

1. **简介**

**目的**

本文档将描述软件系统的整体结构和组织方式，以及各个组件之间的关系和交互方式，其中会使用多种不同的构架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构架方面作出的重要决策，帮助项目团队成员、开发人员以及其他相关利益相关者理解系统的设计和实现。

**本文档包含以下几个部分：**

* 用例视图
* 逻辑视图
* 进程视图
* 部署视图
* 实现视图
* 技术视图
* 数据视图
* 核心算法设计
* 质量属性的设计

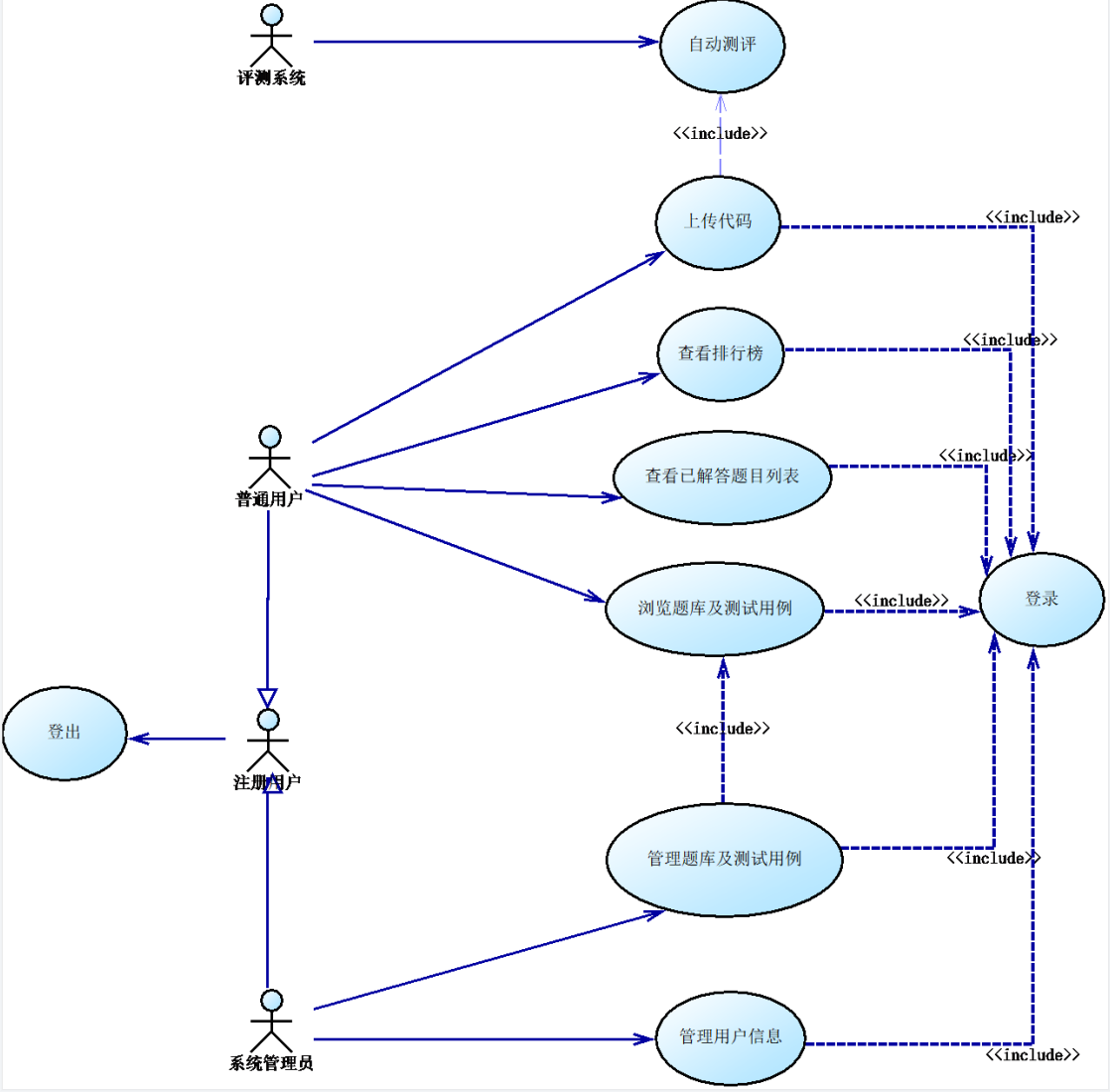
**本文档读者包括：**

* 本系统的开发人员（把握用户需求、为编码提供框架）
* 客户（检查是否满足需求）
* 课程老师和助教（检查是否符合课程要求）

**参考资料**

暂无

2. **用例视图**



**Actor：**

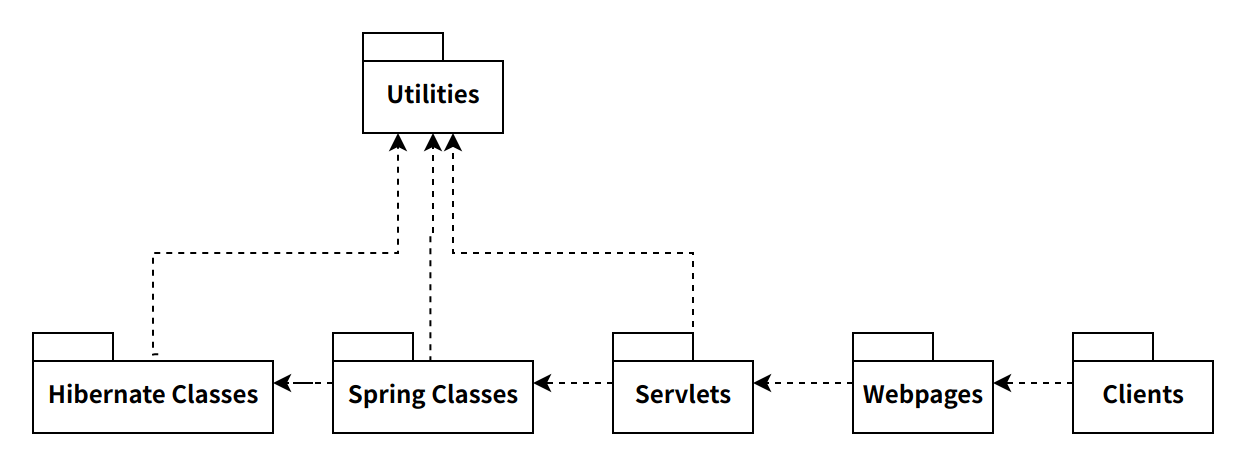
1. 注册用户：已注册用户，经登录可以使用"Testland"平台。
2. 普通用户：平台的一般用户，可以操作平台并进行普通用户权限下的操作，区别于系统管理员。
3. 系统管理员：特殊身份用户（非普通用户），可以管理平台上的内容，包括题库与测试用例。
4. 评测系统：负责自动测评的系统，会进行自动编译及基于测试用例的测评。

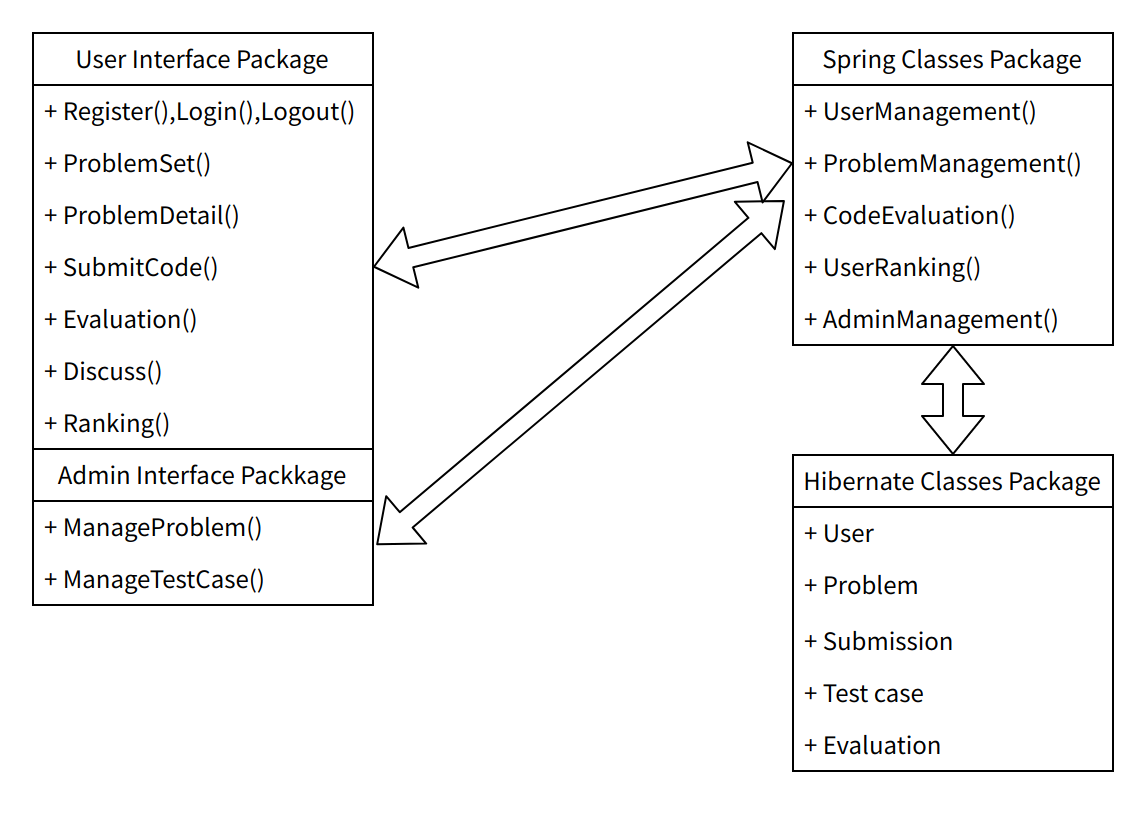
**Usecase：**

1. 登录：已注册用户进入平台进行操作的途径，用于识别身份，且区分系统管理员与普通用户模式。
2. 登出：已登录用户退出账号的操作。
3. 浏览题库：普通用户与管理员均可查看平台上的题库。
4. 管理题库及测试用例：系统管理员可以管理题目及对应测试用例，包含了浏览题库操作。
5. 上传代码：普通用户可选择不同编程语言，上传符合题目要求的代码，用于自动测评。
6. 查看已解答题目列表：普通用户可以查看自己已经解答的题目列表，包括题目名称、通过率、提交次数等。
7. 查看排行榜：普通用户可以查看平台根据用户的题目解答情况（综合考虑解题数、题目难度、一次提交成功率等）进行排名后生成的排行榜。
8. 管理用户信息：系统管理员可以禁用或解禁用户。
9. 自动测评：评测系统进行的自动编译和基于测试用例的自动测试。

3. **逻辑视图**

**概述**





代码评测平台的逻辑视图包括以下几个主要部分：用户界面、业务服务和业务对象。

用户界面包涵盖了与用户交互的各种表单，边界类实现了以下功能：

1. 用户界面包涵盖了与用户交互的各种表单
2. 业务服务包包含了控制类
3. 业务对象包包括了实体类和边界类，以表示系统中的数据结构和逻辑关系

**在构架方面具有重要意义的设计包**

**用户界面包**

1. 用户注册、登录、退出；
2. 浏览题目列表及其详情（名称、难度、标签等）；
3. 选择题目并提交代码；
4. 查看已解答题目列表及相关信息（通过率、提交次数等）；
5. 用户排行榜；
6. 系统管理员界面，用于管理题目及测试用例。

**业务服务包**

1. 用户管理（注册、登录、退出）；
2. 题目管理（查询、显示题目列表、显示题目详情）；
3. 代码评测（编译、运行测试用例）；
4. 用户排行榜计算；
5. 系统管理员对题目及测试用例的管理。

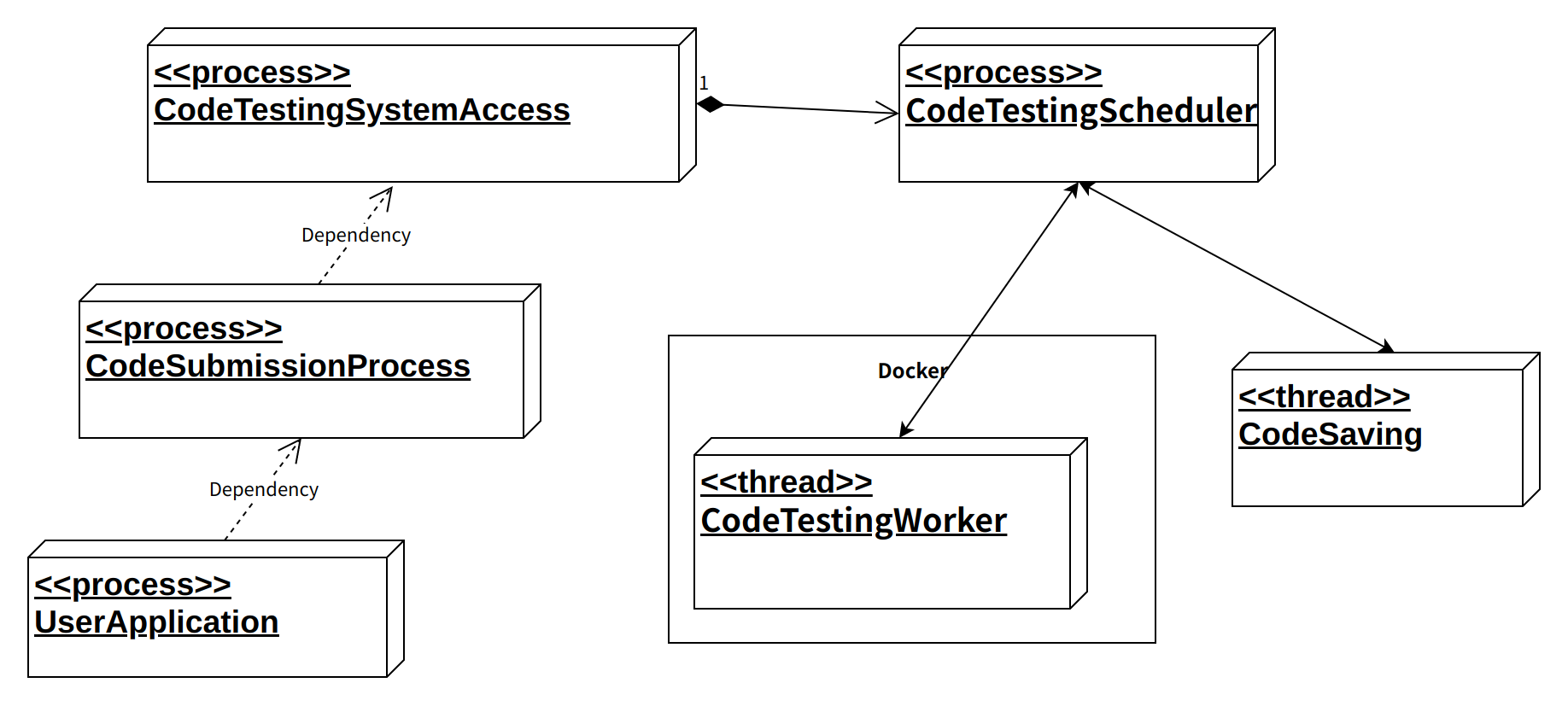
**业务对象包**

1. 用户(User)；
2. 题目(Problem)；
3. 提交记录(Submission)；
4. 测试用例(TestCase)；
5. 评测结果(EvaluationResult)。

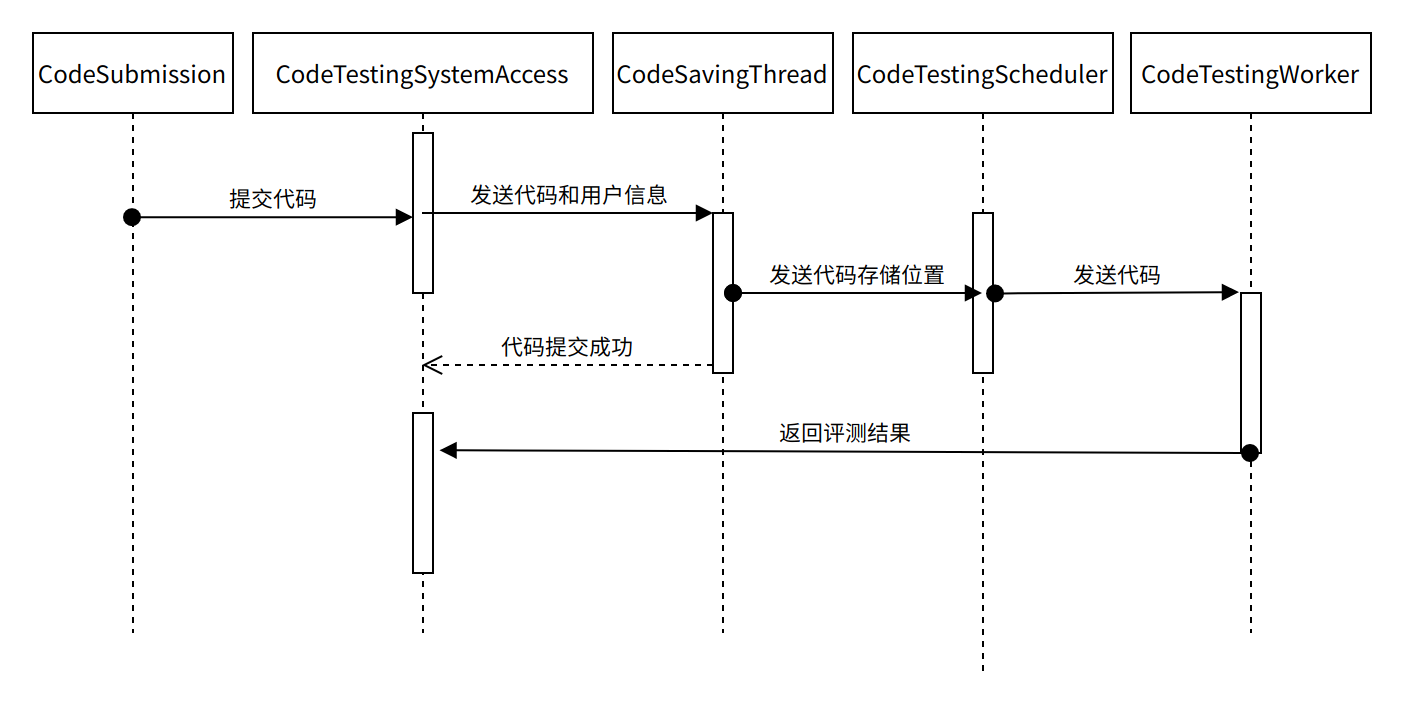
4. **进程视图**

系统采用springboot后端，其他部分进程结构较为简单，此处不再赘述，以下主要说明代码评测过程的进程设计。

**进程图**



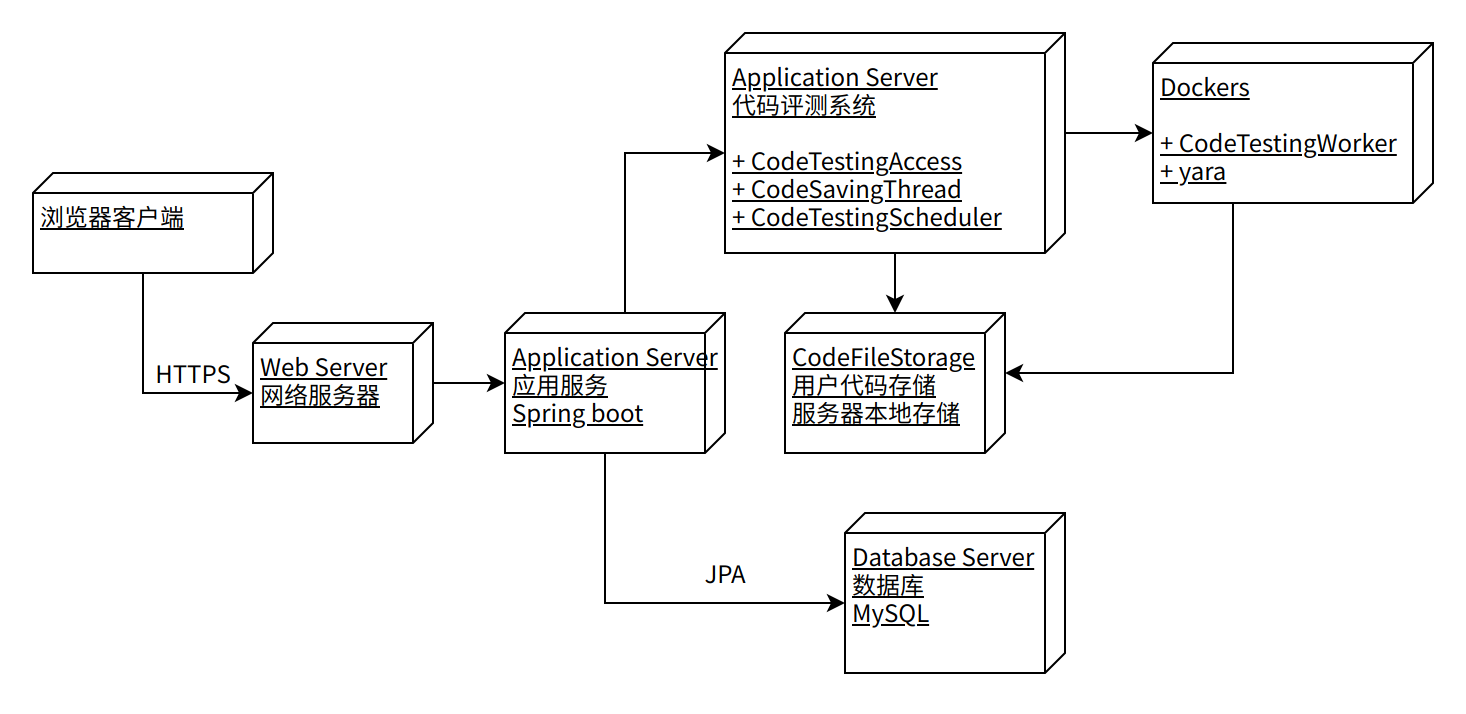
**通信图**



**说明**

1. 用户提交代码（CodeSubmissionProcess）：用户在浏览器将代码提交到系统。
2. 代码评测系统入口（CodeTestingSystemAccess）：后端处理整个代码评测逻辑的进程。
3. 保存代码（CodeSavingThread）：代码被保存到服务器的存储设备中，向评测管理进程发送包含代码存储位置和用户、题目等信息的message，并向系统入口反馈保存成功。
4. 管理代码评测（CodeTestingScheduler）：负责管理代码评测的整个过程，维护待评测队列，负责创建docker image并发送对应的待评测代码信息。
5. Docker中运行评测（CodeTestingWorker）：评测进程在Docker容器中运行，执行代码评测，并返回评测结果。

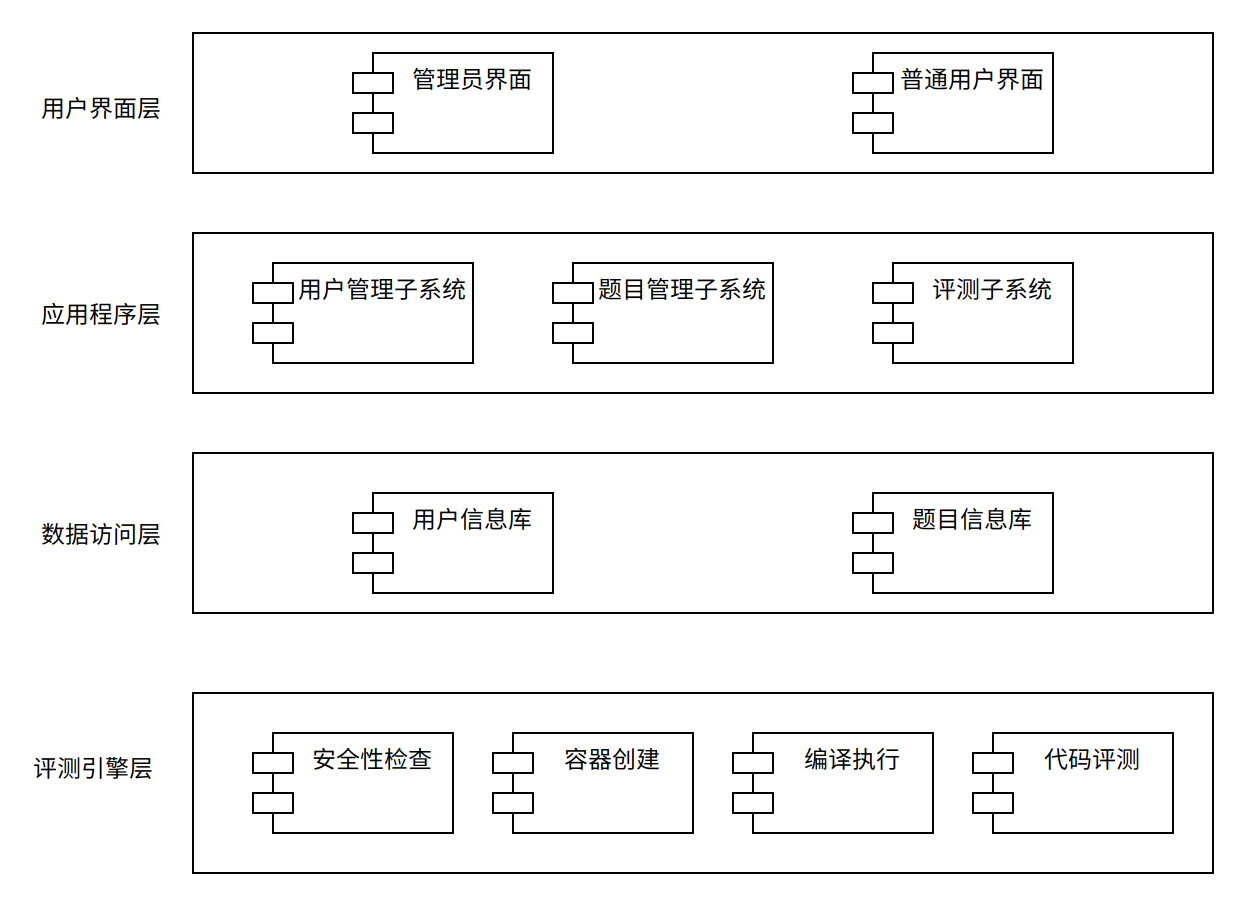
5. **部署视图**

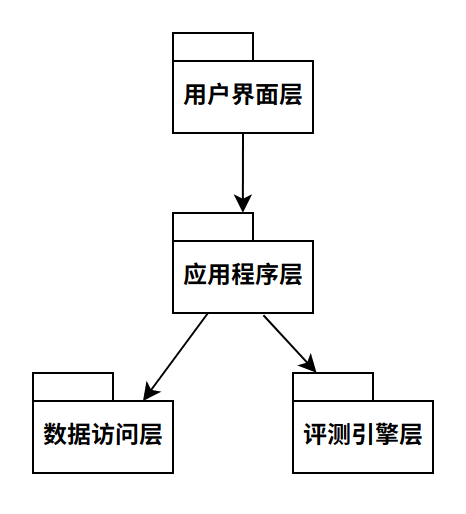


体系结构部署视图的描述了“Testland”在线代码评测平台配置的各种物理节点，以及将任务分配到物理节点的过程。以下是对每个物理节点的详细解释：

1. 浏览器客户端：用户可以通过浏览器访问代码评测平台。
2. 网络服务器：负责接收和处理客户端的HTTPS请求，并返回响应结果。
3. 应用服务器：位于Web服务器与数据库之间，负责处理业务逻辑，包括身份认证、权限控制、数据管理等。应用服务器主要采用Java EE技术栈，例如Spring、Struts、Hibernate等框架。
4. 数据库：存储和管理系统中的数据，包括用户信息、活动信息等。数据库采用关系型数据库MySQL。
5. 代码评测系统：处理用户提交代码的评测逻辑，将用户代码信息保存到用户代码存储，并管理代码评测，创建docker环境运行和测试用户代码。
6. Dockers：用于运行和测试用户代码的容器。
7. 用户代码存储：存储用户提交的所有代码。
8. yara：对用户代码进行安全检测，拦截恶意代码。

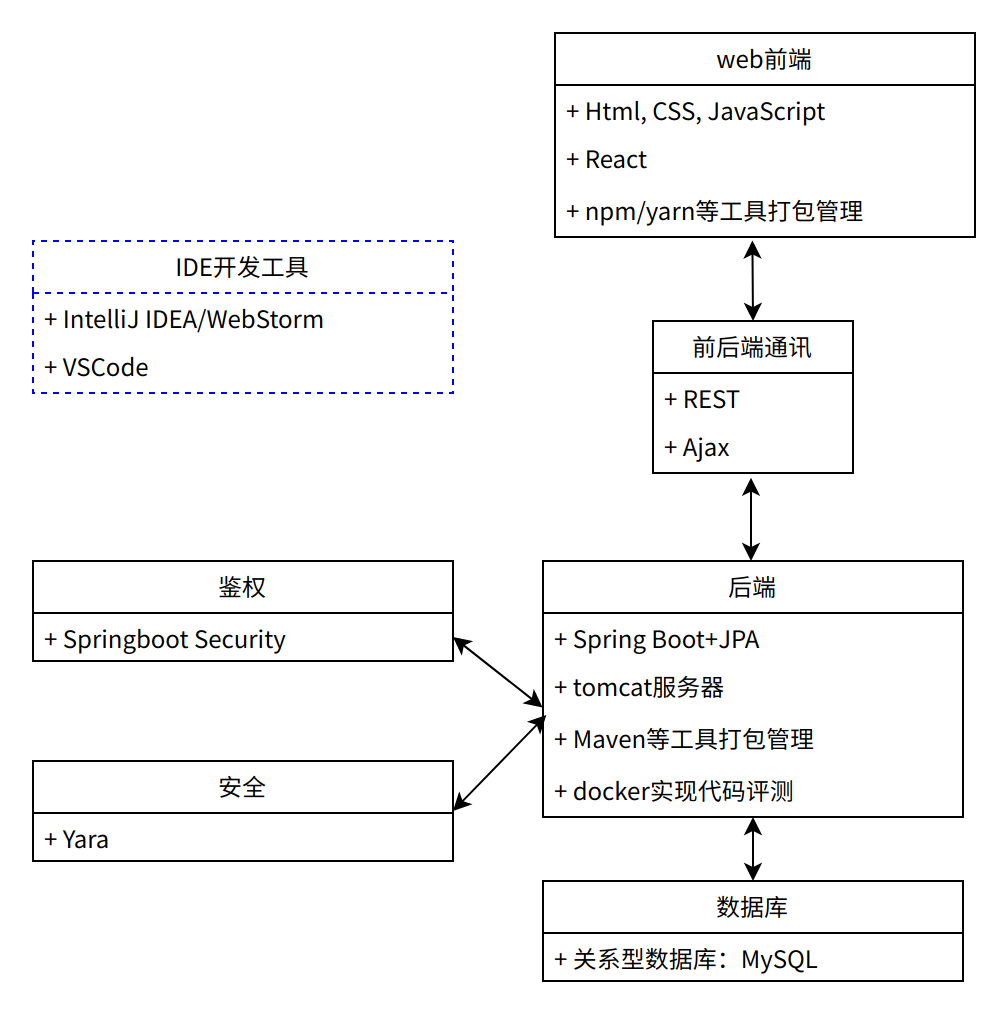
6. **实现视图**





以上是系统的实现视图，包括用户界面层、应用程序层、数据访问层、评测引擎层。其中，用户界面层负责与用户进行交互，接收用户输入和显示结果；应用程序层负责处理用户请求，管理用户和题目等信息；数据访问层负责与数据库进行交互，提供对数据的存取操作；评测引擎层是OJ系统的核心组件，负责接收和评测用户提交的代码。

7. **技术视图**



8. **核心算法设计**

**输入**

* 用户代码: 包含用户提交的待评测代码。
* 相关信息: 包括用户标识、代码标识等与用户代码相关的额外信息。
* 并行度限制: 系统后台指定允许同时运行的 Docker 容器数量。

**输出**

* 评测结果: 包括代码执行结果、错误信息、评分等评测结果数据。

**算法描述**

1. 用户提交代码后，首先进行代码保存操作，将代码保存到指定位置。
2. 返回保存成功或错误信息给用户。
3. 将用户代码存储位置、题目信息等封装成消息并发送到消息队列中。
4. 并行进程调度开始：
5. 从消息队列中获取待处理的消息。
6. 解析消息，获取用户代码和其他相关信息。
7. 创建一个新的 Docker 容器用于运行用户代码。
8. 在 Docker 容器中运行用户代码，并获取评测结果。
9. 将评测结果保存到指定位置。
10. 将评测结果在数据库中进行更新，让用户可以在浏览器端查看。
11. 标记消息为已处理。
12. 重复步骤 a 到 g，直到消息队列为空或达到预设的并行度限制。

9. **质量属性的设计**

**性能**

1. 软件架构可以通过合理的分层、模块化设计和优化算法等方式提高系统的性能。
2. 良好的架构能够减少冗余操作、提高资源利用率、优化数据流和通信，从而提升系统的响应速度和吞吐量。
3. 架构还可以支持水平和垂直扩展，使系统能够处理更高的负载和并发请求。

**可扩展性**

1. 软件架构应该支持系统的可扩展性，即能够方便地进行功能的添加、组件的替换和系统规模的扩大。
2. 良好的架构能够将系统划分为松耦合的模块，使得扩展一个功能或组件不会对其他部分造成影响。
3. 采用分布式架构和微服务架构可以进一步提高可扩展性，通过水平扩展和负载均衡来应对不断增长的用户和数据量。
4. 可扩展的架构允许根据需要增加安全措施和资源分配，提高系统对恶意攻击和安全漏洞的鲁棒性。

**可靠性**

1. 软件架构应该支持系统的可靠性，即能够持续地提供预期的功能和服务，同时对错误和故障具有容错能力。
2. 考虑故障处理和恢复机制，包括错误检测、容错设计、备份和恢复策略等。
3. 通过使用冗余和故障转移技术，可以保证系统在单个组件或节点故障时仍然可用。
4. 安全漏洞和攻击往往会导致系统故障或异常行为，因此可靠的架构设计可以减少潜在的安全风险。可靠性是保证系统的保密性和完整性的重要基础。

**易用性**

1. 我们遵循用户界面设计原则，提供直观、一致，简约的用户界面，并通过界面原型迭代来进一步验证易用性。
2. 为用户隐藏复杂性，提供简单的接口和良好的文档，以降低使用和维护系统的难度。

**可移植性**

1. 软件架构应该支持系统的可移植性，即能够轻松地在不同的平台和环境中部署和运行。
2. 我们将与平台相关的依赖项和逻辑与核心业务逻辑分离，以便在不同的操作系统、硬件或云平台上进行迁移。
3. 我们将采用标准化的接口和协议，以及遵循开放标准和互操作性原则，也有助于实现系统的可移植性。
4. 可移植性可以帮助应对潜在的威胁和攻击，因为系统可以灵活地迁移到安全性更高的平台或环境中。