# BOB MTA（Maintain Assistants）综合运维平台 详细设计说明书

**版本：** v1.0 (预留修订历史表)  
**读者对象：** 前端工程师、后端工程师、测试工程师、运维工程师、产品与实施顾问

**术语与缩写：**

* **MTA**：Maintain Assistants（平台名称，综合运维工具的简称）
* **RBAC**：Role-Based Access Control，基于角色的访问控制机制
* **i18n**：Internationalization，国际化，多语言支持
* **MinIO**：开源对象存储服务，兼容S3协议，用于非结构化文件存储
* **DDL**：Data Definition Language，数据定义语言（如SQL建表语句）
* **ERD**：Entity-Relationship Diagram，实体关系图
* **RPO**：Recovery Point Objective，恢复点目标（容忍的最大数据丢失时间窗口）
* **RTO**：Recovery Time Objective，恢复时间目标（服务中断后恢复的目标时间）
* **SLA**：Service Level Agreement，服务级别协议（对可用性/性能等的约定指标）
* **ICS**：Internet Calendar Scheduling/iCalendar（.ics日历文件格式，用于日程导入/订阅）
* **WebCal**：Web Calendar协议（一种以webcal://URL订阅在线ICS日历的方法）
* **JWT**：JSON Web Token，一种常用的无状态认证令牌格式
* **OIDC**：OpenID Connect，一种基于OAuth2的身份认证标准

**目录：**

1. 文档元信息
2. 背景与目标
3. 总体架构
4. 前端信息架构与交互
5. 领域模型与数据设计
6. 后端详细设计
7. API 设计（REST/JSON）
8. 前端组件与状态流设计
9. 多语言与资源管理
10. 部署与运维
11. 安全与合规
12. 测试与验收
13. 迁移与集成
14. 交付清单
15. 附录

## 1. 背景与目标

**平台定位：** BOB MTA（Maintain Assistants）定位为一款面向多客户运维团队的一体化工具。维护人员可以借助该平台收集和展示客户环境信息，制定和跟踪运维计划，对客户进行分组和标签分类管理，并集中管理常用的运维模板和连接方式，实现对多客户运维工作的集中支持。

**品牌与命名：** 平台品牌命名为“BOB”，产品代号“MTA”（Maintain Assistants）。“BOB”作为品牌标识，体现可靠助手形象；“MTA”则突出平台面向维护助手的定位，方便在沟通中简称引用。

**语言要求：** 平台**默认界面语言为日语**，同时内置多语言支持（至少支持中文，后续可扩展）。后端天然采用 Spring Cloud 的国际化机制集中管理多语言资源，通过资源文件或字典表统一管理词条。前后端应严格分离显示文本和代码，在界面加载时按需拉取对应语言资源。多语言切换无需重启服务，支持运行时切换。

**非功能目标：**

* **性能：** 支持一定并发量和快速响应。初期目标支持 **100+** 并发用户请求，关键页面响应时间在 <500ms 内（99%分位）。批量导入、文件下载等操作也需优化性能。
* **可用性：** 系统高可用，SLA目标可设定为 **99.9%** 年度可用时间。通过容器编排和冗余部署，减少单点故障。数据库采用主从备份，制定备份策略：RPO ≤ 5分钟，RTO ≤ 15分钟，确保故障恢复在可接受范围。
* **可维护性：** 代码结构清晰、模块边界明确，遵循良好架构和编码规范，便于团队协作和后续功能扩展。提供完整的日志、错误码和文档，方便问题排查和运维。
* **可观测性：** 内置监控指标（如接口调用次数、响应时间、错误率）和健康检查。日志采用结构化JSON输出，方便对接ELK/Opensearch等日志平台，实现对系统运行状态的实时监控。
* **安全合规：** 实现全方位的安全机制，包括严格的身份认证与RBAC鉴权、敏感操作审计和日志留存、数据脱敏展示（如有敏感信息），满足合规要求。关键数据变更均记录审计日志，包含操作时间、用户、改动前后值等细节，做到“任何变更均可追溯”。日志中包含用户ID和请求ID等信息，满足安全审计要求。
* **可部署性：** 提供Docker化部署方案，通过一份 Docker Compose 文件即可拉起全套服务，方便在不同环境快速部署。支持在容器平台（如Kubernetes）上部署，提供配置示例。
* **可扩展性：** 平台设计具备弹性扩展能力，包括：支持**自定义字段**（客户档案信息可在运行期动态扩展字段，不需修改表结构）、**自定义流程和标签**（运维流程节点类型和标签规则可扩展）、以及**多租户扩展**（预留租户隔离机制）。同时，采用松耦合架构，新增加的模块或第三方集成（如新的IM工具接口、新的远程连接方式）可通过扩展接口接入，而不影响既有功能。

综上，平台目标是在满足功能需求的同时，从架构上确保易于维护、扩展和部署，并能支撑实际运维场景对性能和安全的要求。

## 2. 总体架构

**技术栈概览：**

* 前端：采用 **React** 框架结合 Ant Design 组件库构建SPA单页应用。布局上可利用 Ant Design Pro 提供的模板/布局体系，快速搭建出成熟的管理后台界面。前端实现国际化(i18n)，建议使用 react-intl 或 react-i18next 等方案，结合后端提供的词条资源接口，实现与后端词条库的同步更新策略（见“多语言与资源管理”章节）。构建工具使用 Webpack或Vite，代码采用 TypeScript 提高可维护性。
* 后端：基于 **Java** 语言，使用 Spring Boot 搭配 Spring Cloud 体系构建微服务架构。核心框架包括：Spring MVC/WebFlux 实现REST API，Spring Security负责认证鉴权，Spring Cloud Config用于集中配置和国际化资源管理，ORM采用 MyBatis（或 MyBatis-Plus）与 PostgreSQL 数据库交互。对于非结构化数据（附件文件、文档），使用 **MinIO** 对象存储服务模拟 S3 以存储文件，并通过其Java SDK与系统集成。
* 数据库：关系型数据库 **PostgreSQL** 用于存储核心业务数据（客户信息、计划、流程、日志等）。选择PostgreSQL可利用其JSON字段支持和可靠事务性，以及丰富的扩展功能。按需要建立必要的索引和分区，保证查询性能。
* 认证与鉴权：采用 **JWT**（JSON Web Token）进行无状态认证，结合 Spring Security 实现RBAC权限控制。用户在登录成功后，后端签发包含用户身份及角色信息的JWT，前端保存该令牌并在后续请求中通过HTTP Header发送。后端通过过滤器验证JWT有效性及权限。企业环境下如需要，也可支持 **OIDC**（OpenID Connect）对接企业统一身份（例如Azure AD），但默认方案为内部JWT认证。RBAC权限模型将细化到菜单和API级别，不同角色可访问不同菜单项和接口。
* 日历集成：计划任务支持通过 **ICS/WebCal** 与外部日历（如Outlook）同步。系统提供 .ics 格式的日历文件导出以及订阅URL。Outlook等支持通过订阅网络日历自动更新计划[[1]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=Note%3A%C2%A0When%20you%20subscribe%20to%20a,take%20more%20than%2024%20hours)。需注意Outlook对订阅的刷新频率较低，更新可能**超过24小时**才同步[[1]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=Note%3A%C2%A0When%20you%20subscribe%20to%20a,take%20more%20than%2024%20hours)，因此紧急计划更改仍需人工通知。直接导入 .ics 文件则仅是一时快照，不会自动更新[[2]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=When%20you%20import%20an%20,or%20phases%20of%20the%20moon)。本平台将提供日历订阅链接用于长期动态查看计划，也支持单次导出ICS文件手动导入。
* 日志与审计：后端采用**统一的日志框架**（如Logback），输出结构化JSON日志，涵盖时间戳、日志级别、线程、请求ID等字段，便于接入 ELK/Opensearch 日志系统。重要操作还会记录**审计日志**到数据库中，字段包括操作者、时间、操作类型、涉及对象及变更内容等，满足合规要求。平台支持将日志配置为JSON格式并输出到stdout，部署时由EFK(Filebeat)+Logstash收集，或直接写入文件由外部Agent采集。审计日志可通过后台接口查询，并支持导出审计报告。
* 部署方式：采用容器化部署，全组件通过 **Docker Compose** 描述，各服务组件（前端Nginx容器、后端服务容器、PostgreSQL、MinIO等）在Compose文件中统一编排。一键启动所有模块，端口和依赖在Compose中配置妥当，便于开发测试和部署。生产环境可基于相同镜像部署至K8s等，配置基于环境变量。

**总体架构设计：** 平台采用前后端分离的分层架构，整体划分为表示层、网关层、服务层、存储层和外部集成接口等。下图展示了系统的架构概览及各组件交互关系：

flowchart LR  
 subgraph Client 前端  
 WebApp[浏览器SPA<br/>React + AntD<br/>多语言支持]  
 end  
 subgraph Server 后端  
 API[API网关服务<br/>(Spring Boot)]  
 Auth[认证鉴权模块<br/>Spring Security + JWT]  
 CustomerSvc[客户&分组服务]  
 PlanSvc[计划&流程服务]  
 TemplateSvc[模板中心服务]  
 FileSvc[文件服务]  
 AuditSvc[审计日志服务]  
 UserSvc[用户管理服务]  
 end  
 subgraph Storage 存储  
 DB[(PostgreSQL<br/>关系数据库)]  
 ObjStore[(MinIO<br/>对象存储)]  
 end  
 ExtCalendar[外部日历<br/>Outlook/Google<br/>ICS订阅]  
 ExtMail[邮件系统<br/>SMTP服务器]  
 ExtIM[IM平台<br/>Webhook/Bot]  
  
 WebApp --JWT--> API  
 API -.验证令牌 via Auth.-> Auth  
 API --路由--> CustomerSvc & PlanSvc & TemplateSvc & FileSvc & AuditSvc & UserSvc  
 CustomerSvc --CRUD--> DB  
 PlanSvc --CRUD--> DB  
 PlanSvc --生成ICS--> ExtCalendar  
 TemplateSvc --发送邮件--> ExtMail  
 TemplateSvc --发送IM消息--> ExtIM  
 FileSvc --读写--> ObjStore  
 AuditSvc --写入日志--> DB  
 API --日志输出--> AuditSvc  
 UserSvc --CRUD--> DB

*图：系统总体架构图（前端、后端各服务、存储与外部集成）*

如上所示：

* 前端SPA应用通过浏览器与后端进行通信。后端设置**API网关**（可由Spring Cloud Gateway或直接由主应用充当），统一接收客户端请求，在网关层完成JWT认证和路由转发。Spring Security在网关或各服务中保证非授权请求无法访问受保护资源。
* 服务层根据领域划分为多个微服务（初期可在一个Spring Boot应用中模块化实现，后续可拆分）：**客户服务**负责客户及分组、标签、自定义字段等管理；**计划服务**负责运维计划和流程相关逻辑；**模板服务**提供邮件、IM、远程连接等模板的管理和渲染；**文件服务**封装MinIO对象存储操作；**审计日志服务**集中处理审计日志写入和查询；**用户服务**负责用户、角色、权限管理和认证（如果不拆分，可由一个Auth模块处理认证，User模块处理用户CRUD）。各服务之间通过轻量级REST或消息队列通信（当前主要通过网关转发REST调用）。
* 存储层包括关系数据库和对象存储：业务数据主要存于PostgreSQL，各服务通过自身的DAO/Repository访问数据库（如MyBatis）。对象存储MinIO用于保存上传的文件附件，File服务通过MinIO提供的S3 SDK访问。
* 外部集成：系统通过标准协议与外部工具集成。如通过**ICS日历**订阅使外部Outlook等获取计划日程更新，通过SMTP协议与企业邮件服务器集成发送通知邮件，通过Webhook或Bot API与IM工具（如Slack、Teams或企业微信）集成发送即时通知。
* 日志与监控：各服务将操作审计信息发送给审计服务或直接写数据库审计表，同时将运行日志输出到集中日志。在生产中，可部署EFK/Opensearch收集日志，并部署Prometheus监控各服务（例如通过/actuator/prometheus指标）。

**模块划分与依赖：** 平台模块按照业务和基础能力分层，实现**高内聚低耦合**：

* **网关与认证模块（Auth/Gateway）：** 提供统一入口，处理用户登录/注销、JWT签发验证、路由转发以及权限控制。依赖用户模块获取用户和角色信息，使用JWT与各业务服务通信。
* **客户管理模块（Customer）：** 管理客户基本信息、客户分组、客户标签、自定义字段等。提供客户的增删改查API及分组、标签管理功能。依赖文件服务（例如关联客户的附件）和可能引用模板（如客户可能关联远程连接模板）。
* **标签管理模块（Tag）：** 定义标签(tag)及其样式、规则，可供客户或计划等实体关联。与客户模块、计划模块通过关联表关联，实现多对多关系。
* **自定义字段模块（Custom Field）：** 支持管理员定义新的字段（属性）以附加在客户信息上。定义表存储字段的名称、类型、适用范围，值表存储具体客户的该字段值。允许新增字段并即时在界面展示。与客户模块关联紧密。
* **运维计划模块（Plan）：** 管理维护计划任务，包含计划的时间、关联客户、参与人员等。该模块依赖流程模块来定义计划所用的流程模板，依赖标签模块以给计划打标签。
* **流程引擎模块（Process Engine）：** 定义运维流程（流程模板）及流程中的节点。负责流程的设计模式与执行模式逻辑：流程模板可以增删节点、配置顺序和阈值，执行时跟踪节点完成情况。节点可关联一个**动作(Action)**，如远程连接、发送邮件等。流程模块提供统一接口调用不同类型的动作处理器。
* **模板中心模块（Template）：** 提供各类运维动作的模板管理，包括邮件模板、IM消息模板、链接模板、远程连接模板等。模板定义包括模板内容和占位符参数。其他模块（如流程节点的动作）可以引用这些模板，根据上下文渲染出实际内容（例如填入客户名、日期等）。模板模块也负责与外部服务集成（发送邮件、调用IM webhook等）。
* **文件服务模块（File）：** 管理文件的上传、下载、存储。封装MinIO对象存储操作，包括桶(bucket)管理、预签名URL生成、元数据记录等。对上提供简洁的文件上传/下载接口，对下保证文件与业务数据(如客户、流程节点)的关联一致性。
* **审计日志模块（Audit）：** 拦截全平台的关键操作，将操作详情存储到审计日志表。可在后台界面查询过滤日志。审计模块通过AOP等手段尽可能无侵入地记录各种操作细节。
* **用户及权限模块（User/Admin）：** 管理系统的用户账户、角色和权限。提供用户注册、激活、密码重置、角色分配等功能。权限控制采用RBAC模型，预置常见角色（如管理员、运维人员等），并定义角色对应的菜单和API访问权限。

模块之间通过清晰接口交互。例如客户模块调用文件服务获取客户附件列表，流程模块调用模板服务发送通知等。各模块数据库表根据模块归属命名，必要时通过外键关联。模块拆分后，可以独立微服务形式部署，模块间通信通过REST或消息队列（当前设计均为REST API调用，通过网关聚合）。

## 3. 前端信息架构与交互

**整体布局：** 前端采用典型的管理后台布局，上方导航栏+主内容区。**顶部导航栏**包含：左侧品牌LOGO或项目名，中间是一级功能菜单，右侧是用户信息区（登录用户的名字、头像，支持下拉菜单进行个人设置、登出等；未登录时显示“登录”按钮）。导航栏固定顶部，方便随时切换功能模块。**主内容区**采用Ant Design Pro的布局方案，将各页面内容在导航栏下方渲染，支持路由切换和面包屑导航（如适用）。整个UI支持响应式，在常规桌面分辨率下正常显示。

**菜单项与权限控制：** 顶部菜单根据用户角色动态展示：

* “客户一览” – 对应客户管理页面，**所有登录用户**可见（运维人员需要查阅客户信息）。
* “运维计划” – 对应计划和流程管理页面，**所有登录用户**可见（运维人员需要查看和执行计划）。
* “用户管理” – 对应用户/权限管理页面，仅**管理员角色**可见（普通运维无权管理用户）。

菜单项可以扩展，例如将来增加“模板中心”“审计日志”等菜单，可以通过权限配置只给管理员或特定角色显示。前端可从后端获取当前用户角色拥有的菜单列表，动态生成导航。未授权访问的路由应有重定向或403提示。

### 客户一览页面

该页面采用**左右分栏**布局：左侧是客户列表导航，右侧是所选客户的详细信息。

* **左侧客户列表：** 提供多种视图方式以便快速查找客户：
* **分组视图：** 按客户分组（例如按“地区”分组，或按“客户所属行业/类型”分组）。分组类别可配置，默认支持按客户所属的“客户分组”分类（由后台数据定义，如“大学”“企业”“政府”等）或者按地理区域（如“北海道”“関東”等）分类。
* **索引视图：** 提供按拼音/假名的首字母索引浏览（对中文客户名按拼音首字母A-Z，对日文客户名按假名あ-わ排序）。点击字母/假名索引，列表跳转到相应首字母的客户。
* **无分组视图：** 即简单列表模式，所有客户按照编码或名称排序展示。用户可切换排序字段（客户编码、名称等）。

列表上方提供**搜索框**，支持按客户名称关键字搜索，支持拼音/假名模糊匹配（如输入“北”会匹配“北海道大学”等）。搜索支持实时过滤列表。列表支持**折叠/展开**分组节点，以及根据**标签**进行过滤（如只显示标记了特定标签的客户）。标签过滤UI可用多选下拉或图标按钮形式，用户选中某标签后，仅显示具有该标签的客户。

列表每项显示概要信息：如客户名称（主要显示项），可附带次信息如客户编码、所属分组或地区、已使用产品列表等摘要。根据需要可在列表项右侧显示该客户的标签标识（彩色小圆点或图标）。当客户较多时，列表支持分页或虚拟滚动加载。

* **右侧客户详情：** 展示所选客户的详细档案信息。为了清晰起见，详情按**信息类别分块**显示，支持自定义分组和字段。默认可参考附件提供的“客户远程接続情報”CSV的字段，将其划分为若干板块，例如：
* **基本信息：** 包含客户编码、名称、简称、所属分组、地区、使用产品等基础字段。
* **远程连接信息：** 包含连接形态（如VPN/専用線/アナログ等）、使用的连接工具（如GlobalProtect、LAPLINK等）、接続制限（如是否限制终端数量）、特殊运用说明、顾客别情報参照（例如是否需查看特定资料，“要”表示需要）、特記事项（特殊备注）等。该部分对应CSV中的“リモート接続情報”栏目及相关字段。
* **连接地址/联系方式：** 包含远程连接的地址（IP）或电话，可能还有VPN服务器地址、账号等（根据CSV“接続先IPアドレス/電話”字段）。
* **其他附加信息：** 任何不属于以上分类的其他信息，如最新更新信息（CSV中的“更新情報”历史）、维护备注等。

每个板块清晰标明标题，板块内按照字段列出“字段名称：值”。对于**自定义字段**，如果管理员新增了新的字段且属于某分类，该字段应自动出现在对应分类板块中（如果事先设定了板块，否则归入“其他信息”）。平台支持**自定义板块和字段**：管理员可以通过配置将若干字段归为一组显示，并命名板块；未配置的字段默认平铺。

**标签展示：** 客户详情页明显位置展示该客户的标签列表，每个标签以特定颜色和图标标识。标签可能代表客户属性（例如“重点客户”、“需定期巡检”等）。颜色和图标在标签定义时指定，并可带有**条件规则**：例如可以定义一个标签“需更新”自动应用于那些“更新情報”字段超过一年未更新的客户。这要求系统在保存客户信息或定时任务时，根据条件为客户打/去除标签（条件规则可简单配置或后期扩展）。初期实现可为标签配置手动或简单自动规则。

**远程连接入口：** 在远程连接信息板块，需提供实际操作的入口。例如对于VPN或远程桌面连接的信息，如果可能集成跳转，则提供一个“连接”按钮/链接。当用户点击时，平台可根据预先配置的**远程连接模板**启动连接流程。例如： - 若模板是RDP远程桌面，点击后触发浏览器下载RDP文件并提示用远程桌面客户端打开； - 若是SSH，点击可能复制SSH地址到剪贴板或通过特定URL协议尝试打开ssh客户端； - 对于需要第三方工具（如LAPLINK）的情况，可提供相应说明或调用外部程序的接口（受浏览器限制，Web无法直接运行本地程序，可能需要用户手动操作）。

平台在当前版本会**预留远程连接执行接口**：即使暂不能完全从Web直接发起远程连接，也提供入口按钮和相应后端API，后续可以与客户端代理或本地应用集成来完成实际连接执行。

### 运维计划页面

运维计划页面提供对所有维护计划的查看和管理，支持**多种视图切换**：

* **按客户视图：** 按客户组织显示计划。用户先选择一个客户，然后看到该客户下的所有计划列表。列表中每条计划显示计划名称、计划开始/结束时间、状态（设计中/已发布/执行中/已完成）、主要参与人等。可以通过下拉选择其他客户或在列表中搜索计划名称。适合聚焦单一客户的所有计划。
* **日历视图：** 以时间维度查看所有计划。提供多种粒度：
* **日视图（日程表）：** 当天24小时按时间轴列出各计划，通常以小时刻度时间轴形式展示。适合查看当天具体工单的时间安排。
* **周视图（周历）：** 显示一周7天的日历，每天作为一列，计划块显示在各自日期的列中（类似常见工作日历周视图）。适合查看本周的工作安排。
* **月视图（月历）：** 显示一个月的日历网格，每个日期格子内列出当天的计划简要。适合宏观查看当月任务分布。
* **年视图（年历）：** 以月份为单位，展示全年12个月，每月可能以简略形式显示计划数量或重要计划标记。适合长期规划查看。

用户可通过Tab或下拉切换上述视图。日历视图借助Ant Design的 Calendar 组件或 FullCalendar 等实现交互。可以提供左右切换按钮翻阅前后时间范围。

**计划卡片内容：** 在无论哪种视图下，计划都会以卡片或条目形式呈现，展示核心信息：计划标题/名称、计划时间（开始~结束或开始+持续时长）、参与人员（主要负责人或执行人列表）、流程进度摘要、标签等。若计划还在设计阶段，可能标记为“设计中”。如果计划已发布执行，显示当前进展（如“进行中，第2/5步”）。计划卡上应有一个按钮/链接切换**“设计模式/执行模式”**视图，方便用户查看或编辑流程细节。对于尚未发布的计划，进入设计模式（可编辑流程）；对于已发布的计划，进入执行模式（记录实际执行情况）。

**流程执行模式 UX：** 当用户打开某个具体计划的详情（包括流程）时，根据计划状态呈现两种模式：

* **设计模式：** 用于计划编排和流程设计。界面上流程节点以**可编辑**状态展示，可以增删节点、调整顺序、编辑每个节点属性（名称、负责人、预计时长、动作类型等）。设计模式下，所有节点一律均等样式（无强调当前节点概念，因为未开始执行），可拖拽调整顺序。支持将某些节点缩进表示子流程/子节点（树结构）。未发布前，流程节点可以自由编辑。
* **执行模式：** 当计划发布开始执行后，流程切换为执行模式UI。执行模式下，前端对流程步骤进行视觉强调：**当前待执行的节点放大突出显示**，使用高亮边框或放大卡片表示；**已完成的节点**标记为完成状态，例如以绿色底色并加上对勾标记，文字变灰或划线表示完成；**后续未开始的节点**显示为半透明或缩小状态以弱化。整个流程可能采用水平进度条或竖向时间轴的方式展现，当前节点居中或居前显示，上下滚动区域露出前后节点但淡化。已完成节点不仅样式变化，也可能锁定不可再编辑，仅展示记录的数据（实际完成时间、执行人等）。当前节点则提供操作入口（开始/完成按钮，填写备注等）。

这种设计保证执行时用户聚焦当前步骤，同时可以回顾已完成和预览后续步骤。完成一个节点后，前端自动更新UI，将下一个节点设为当前，更新样式。用户还可手动选择查看任意节点详情（包括日志或附件）。执行模式的界面设计需**直观易用**，例如可以使用一个进度条组件或定制的流程图组件。**示例：**当前节点背景高亮蓝色，字体加粗；完成节点背景灰绿并显示✔符号；未开始节点背景浅灰，无突出。确保界面在执行模式与设计模式间切换时有明显提示，避免混淆（可在界面上方以标签或颜色提示当前是“执行视图”还是“设计视图”）。

### 用户管理页面

用户管理页面供管理员管理平台用户账号，支持用户自行注册流程。主要功能和界面要点：

* **用户列表：** 管理员可在此看到所有用户账户的列表，包括用户名、姓名（可选）、邮箱、角色、状态（激活/未激活/锁定）等。列表支持按角色或状态过滤，按用户名/邮箱搜索。
* **创建用户：** 管理员可以手动创建新用户账号。点击“新建用户”打开表单，填写用户名、邮箱、默认角色等信息。提交后，新用户记录创建，状态为“未激活”，系统发送一封激活邮件到用户邮箱，其中包含激活链接。管理员也可选择“立即激活”选项直接将用户设为激活状态并手动发送初始密码给用户（此情况下需为用户设定初始密码）。
* **自助注册：** 平台支持用户自注册，但为防止无关人员注册，需要预先在系统录入其邮箱或发放邀请码。实现方案：管理员在系统中预创建一个“待激活用户”记录，填入邮箱（和可选姓名、分配角色），状态标记为“待注册”。系统发送邀请邮件给该邮箱，包含注册链接。当用户点击注册链接时，前端打开注册页面，自动带入邮箱，让用户设置用户名和密码，提交后完成注册并自动激活。也可以允许用户在登录页点击“注册账号”，输入邮箱，如果该邮箱在预留列表，则发送验证码或激活邮件完成流程；不在预留列表则提示无法注册。
* **激活流程：** 无论管理员创建还是自助注册，激活逻辑相似：系统生成一个唯一激活token，存数据库并通过邮件发送链接（链接包含token）。当用户点击链接，前端调用后台激活API（带上token），后台验证token有效后将对应用户状态置为激活，可以登录。未激活用户尝试登录应被拒绝。激活链接应设置有效期（如24小时），超期需管理员重新发送。管理员界面提供“重发激活邮件”功能。
* **密码重置/修改：** 用户登录后可在个人设置中修改密码（需提供旧密码验证）。未登录情况下提供“忘记密码”功能：用户输入注册邮箱，后台验证存在后生成重置token发送邮件。用户通过邮件链接进入设置新密码页面（类似激活流程，token有效期短比如30分钟）。管理员也可在用户列表中对某个用户执行“重置密码”操作，此操作可让系统生成随机密码并邮件通知，或发重置链接。
* **角色与权限分配：** 用户详情或编辑页面，管理员可以为用户指定角色（单选或多选）。界面上应加载角色列表供选择。角色变更提交后即时生效（前端可能需提示用户重新登录以更新权限）。权限（如果细化到单独权限点）通常通过角色管理，但若实现了更细粒度权限，也可提供在用户详情中直观查看其权限集。

此外，用户管理页面应显示**多语言**支持情况，例如邮件内容有日文和中文模板。激活邮件和重置邮件需支持根据用户偏好或系统默认语言发送对应语言版本。管理员在创建用户时可选择语言（或系统默认日文）。

**前端多语言策略：** 前端整体使用日语为默认语言，支持中文切换。所有UI文本（菜单名、按钮标签、提示消息等）不得硬编码，而是使用国际化资源键。采用 react-intl 或 react-i18next 等库实现文本渲染，根据当前语言选择对应的文案。多语言资源的管理由后端集中提供：后端维护一个**词条字典**，前端在初始化时调用API获取默认语言（日语）的全部词条JSON，并缓存在客户端。用户切换语言时，再调用API获取目标语言资源（或预先加载中/日两个包）。**资源同步策略：** 考虑到词条可能随版本更新增加或修改，前端应设计获取资源接口时支持**版本控制**：例如后端提供每种语言一个版本号或哈希，前端存储上次获取的版本，通过 ETag/If-None-Match 机制避免重复加载。如果后端检测词条有更新则返回新资源，否则返回304无需更新。这样实现灰度发布词条：在后台更新词条库后，前端用户刷新或定时轮询词条接口即可拿到最新翻译。

前端与后端的词条**命名规范**需统一，如采用“模块.页面.元素”的key格式，避免冲突和歧义。例如客户列表页面标题使用key customer.list.title，日文对应“顧客一覧”，中文对应“客户列表”。后端返回的校验错误消息也应有相应key并提供多语言文本。前后端需约定好各功能模块使用的词条前缀和含义，维护一份词条清单以供翻译和开发参考。

**组件与状态管理：** 前端大量使用 Ant Design 提供的现有组件构建界面，例如：表格组件展示列表，Form组件处理表单，Modal对话框用于弹出编辑框，Tree或Menu组件用于左侧分组列表，Calendar组件用于日历视图，Steps组件或自定义Progress用于流程节点显示，等等。全局状态管理方面，可根据复杂度选型：若模块间共享状态较多，可使用 **Redux + Redux Toolkit** 规范化管理，结合 React Context 提供局部状态；或者使用更轻量的 **Zustand** 等简化方案。考虑到涉及用户登录信息、多语言当前语言、缓存的词条、选中客户等跨页面状态，使用Redux是稳妥选择。使用Redux Toolkit简化样板代码，并约定各slice按模块划分（如 authSlice, customerSlice, planSlice 等）。

异步数据请求使用 **Axios** 或基于fetch封装的HTTP库，统一封装请求方法和拦截器，处理JWT注入头、全局错误处理（如401跳转登录）。也可采用 **React Query** 来管理服务器状态，自动处理缓存和刷新。React Query对列表数据和详情数据的缓存更新很方便，但需注意与Redux配合避免重复。方案选择上，可以**Redux负责全局少量关键状态**（如当前用户信息、词条库），而**React Query负责服务器数据**（如客户列表、计划列表），让组件直接通过hooks获取数据和更新。这种组合可减少手写Redux代码，提高数据获取的实时性。

组件设计遵循**单一职责**、解耦复用的原则。如客户详情中的每个信息板块可拆为独立组件（BasicInfoPanel, ConnectionInfoPanel等），以属性传入数据或配置；标签列表也封装为 TagList 组件；自定义字段渲染器根据字段类型渲染文本、链接等。全局提供错误边界或统一的ErrorHandler组件，捕获渲染异常并给予用户友好提示。

## 4. 领域模型与数据设计

平台的数据模型围绕客户信息、运维计划、流程、模板等核心概念设计，采用**第三范式**规范建模，同时为了满足灵活性，引入自定义字段、标签等扩展机制。以下给出核心实体的ERD（实体关系图），并说明主要字段和关系。

erDiagram  
 TENANT ||--o{ CUSTOMER : "包含"  
 CUSTOMER }o--o{ CUSTOMER\_GROUP : "分类"  
 CUSTOMER ||--o{ CUSTOMER\_CUSTOM\_FIELD\_VAL : "有"  
 CUSTOMER\_CUSTOM\_FIELD\_VAL }o--|| CUSTOMER\_CUSTOM\_FIELD\_DEF : "定义"  
 CUSTOMER ||--o{ CUSTOMER\_TAG : "标记"  
 CUSTOMER\_TAG }o--|| TAG\_DEF : "标签定义"  
 CUSTOMER ||--o{ PLAN : "参与"  
 USER ||--o{ PLAN\_PARTICIPANT : "参与计划"  
 PLAN ||--o{ PROCESS\_NODE : "包含节点"  
 PLAN ||--o{ PLAN\_TAG : "标记"  
 PLAN\_TAG }o--|| TAG\_DEF  
 PROCESS\_NODE ||--o{ PROCESS\_NODE : "子节点"  
 PROCESS\_NODE ||--o{ NODE\_INSTANCE : "执行实例"  
 PLAN ||--o{ PROCESS\_INSTANCE : "执行过程"  
 PROCESS\_INSTANCE ||--o{ NODE\_INSTANCE : "包含节点执行"  
 USER ||--o{ NODE\_INSTANCE : "执行人"  
 TEMPLATE\_EMAIL ||--o{ EMAIL\_HISTORY : "发送记录"  
 TEMPLATE\_IM ||--o{ IM\_HISTORY : "发送记录"  
 TEMPLATE\_REMOTE ||--o{ REMOTE\_HISTORY : "执行记录"  
 USER ||--o{ AUDIT\_LOG : "操作"  
 USER ||--o{ FILE\_OBJECT : "上传文件"  
 FILE\_OBJECT ||--o{ FILE\_OBJECT\_VERSION : "版本"  
 USER ||--o{ CUSTOMER : "属于租户"  
 USER ||--o{ USER\_ROLE : "拥有"  
 ROLE ||--o{ USER\_ROLE : "赋予"  
 ROLE ||--o{ ROLE\_PERMISSION : "赋权"  
 PERMISSION ||--o{ ROLE\_PERMISSION : "附于角色"

*图：核心实体ERD简图（主外键关系）*

上图描述了主要实体及关系（去除部分中间实体以简化阅读）。下面对关键实体及字段进行说明：

* **Tenant（租户）**：支持多租户的场景，一个租户代表使用平台的一个组织/公司。字段：id，name等。如单一部署给内部使用可无视租户。客户、用户等记录需有tenant\_id与租户关联，用于数据隔离。
* **Customer（客户）**：客户基本信息表，表示受运维支持的客户单位。主要字段：id (PK), code (客户编码，唯一), name (名称), kana/pinyin (名称的读音，用于索引排序), group\_id (所属分组，可为空), region (地区), remarks (备注) 等。根据CSV示例，客户表也可包含一些固定字段如connection\_type(接続形態), connection\_tool, connection\_limit, special\_operation(特殊運用), info\_required(顧客別情報参照要否), note(特記事項), ip\_address 等。但是为了**灵活性**，这些更适合作为**自定义字段**而非固定列（详见下文自定义字段模型）。因此Customer表除了通用字段外，其它客户特有属性通过CUSTOMER\_CUSTOM\_FIELD存储，更易扩展。Audit: Customer应有created\_at,updated\_at,created\_by,updated\_by等审计字段。
* **Customer\_Group（客户分组）**：枚举客户的分组类别，如行业或客户类型。字段：id, group\_name, parent\_id(支持分组层级，可选), sort\_order等。客户表的group\_id外键关联到此表。管理员可CRUD分组，用于左侧列表的分组视图。
* **Customer\_Custom\_Field\_Def（客户自定义字段定义）**：用于支持动态扩展客户属性。字段：id, field\_key(字段标识/英文), field\_name(字段名称/显示名，多语言可扩展), field\_type(数据类型，如TEXT, NUMBER, BOOLEAN, DATE等), field\_group(字段所属分组，如“远程连接信息”板块), tenant\_id(字段作用域，可选：为空表示全局字段，不区分租户)。可以由管理员在UI上新增一条记录，即在系统中增加一个新的客户字段。可以考虑增加is\_active(是否启用)，is\_required(是否必填)等属性。
* **Customer\_Custom\_Field\_Val（客户自定义字段值）**：存储特定客户的某个自定义字段的值。字段：id, customer\_id(外键关联Customer), field\_def\_id(外键关联字段定义表), value(通用值字段，存字符串；对于不同类型字段，可在应用层做类型转换，或额外存numeric\_value, date\_value等但为简化设计，这里用字符串统一存储并约束长度)。需要唯一约束 (customer\_id, field\_def\_id) 防止重复。同一字段的值可更新，updated\_at记录最近修改时间。**约束：** 如果字段定义停用或删除，应该同时处理相关值记录（可选策略是逻辑删除）。

*说明：通过上述两表，可以支持任意数量的动态字段扩展。例如CSV里的“接続形態”可以在字段定义表中创建一条记录(key=connection\_type, 名称=接続形態)，然后每个客户的Val表里插入对应值（VPN/専用線等）。新增字段不涉及表结构变更，只需插入定义表数据，前端根据定义自动呈现。同时若考虑档案版本化（版本管理），可以在Val表中增加version或设立历史表，每次修改保存旧值，以备查询历史档案。但鉴于所有更改已在审计日志中记录，可选择不额外保留历史版本，或仅对关键字段做履历。*

* **Tag\_Def（标签定义）**：定义系统中可用的标签。字段：id, tag\_name(标签名称), color(颜色码，例如Hex或预设颜色名), icon(图标名，例如使用AntD图标库标识一个形象,可选), scope(适用范围，例如 CUSTOMER, PLAN, BOTH), apply\_rule(可选，应用规则表达式，用于自动打标签; 简单起见可用JSON或脚本描述，比如 {"field":"last\_update","op":">","value":"365d"} 表示最后更新超过365天自动加此标签)。tenant\_id(标签可租户隔离)。标签定义由管理员维护，可手动指定颜色和图标。
* **Customer\_Tag（客户标签关联）**：表示某客户被赋予了某标签。字段：customer\_id, tag\_id，联合主键。标签可以手动关联（管理员在客户详情中添加/移除标签），也可通过规则自动关联（由后台任务或保存时判断满足Tag\_Def.apply\_rule则插入记录）。需支持移除标签（删除记录）。
* **Plan（运维计划）**：维护计划表，每条记录表示一个具体的运维任务安排。主要字段：id, title(计划标题), customer\_id(关联的客户ID，表明此计划针对哪个客户。若计划涉及多个客户，可扩展一张关联表Plan\_Customer，但本版假定一个计划对应一个客户), description(计划描述), start\_time, end\_time(计划预定的时间范围，或单次任务只用start\_time+duration), status(状态枚举：DESIGN/未发布, SCHEDULED/已发布未执行, IN\_PROGRESS/执行中, COMPLETED/完成, CANCELED/取消), process\_id(关联流程模板，详见下面Process定义), timezone(时区，默认为Asia/Tokyo，可留空表示系统默认), repeat\_rule(重复规则，如ICS RRULE，可暂留空日后扩展), created\_by, created\_at, updated\_by, updated\_at等。

除此之外，还有**参与人**信息。可有字段owner\_id(负责人用户ID)以及通过关联表记录多个参与执行的人员：  
- **Plan\_Participant（计划参与人）**：字段：plan\_id, user\_id, role(角色：执行者/审核者等，可选)，表示某用户参与该计划。至少需要一个负责人（owner）。在界面上，计划卡片会显示参与者列表。

计划表需要在**发布**时锁定关联的流程定义（如复制流程模板为执行版本），或通过状态标识在流程表区分设计/执行。具体流程设计参见Process相关。

* **Process\_Def/Node（流程定义及节点）**：这里采用两张表来描述运维流程模板，或者也可合并到Plan内。考虑到流程可能在多个计划中复用，最好拆出独立实体表示“流程模板”。不过根据需求每个计划的流程可能都定制，因此流程定义可视为计划的子表。两种方案：
* **Plan本身就是一个流程容器**，直接在节点表按plan\_id区分。即不显式有Process\_Def表，Process\_Node表的plan\_id引用Plan，所有节点隶属于某计划。这样每个计划都自带流程，不存在独立模板复用。
* **独立Process\_Def模板**，Plan引用一个Process\_Def，这允许多个计划使用同一个流程模板（例如标准操作流程）。Plan发布时可复制模板节点成为实例。

鉴于需求未明确流程需要复用，且更强调设计/执行模式差异，倾向使用**第一种**简化方案：以Plan ID标识一棵流程节点树，用Process\_Node表存储。若将来需要模板库，可将某计划流程保存为模板再套用。

* **Process\_Node（流程节点）**：表示流程中的一个步骤节点。字段：id, plan\_id, parent\_id(父节点ID，支持子流程结构，如某节点下有一组子任务), name(节点名称，比如“备份数据库”), assignee\_id(指派的责任人用户ID，可以为空表示未定或多人参与), expected\_duration(预计时长，分钟数或小时数), action\_type(动作类型，枚举，例如 NONE(无自动动作), REMOTE(远程连接), EMAIL(发送邮件), IM(发送IM通知), LINK(打开链接), FILE(打开文件夹) 等), action\_ref\_id(引用 Template 的ID，如action\_type是EMAIL则ref\_id指向某邮件模板ID; 如果action\_type是REMOTE则指向Remote模板ID; 没有则为空), threshold(完成阈值，仅对有子节点的节点有效，数值0-100表示百分比，或特定值如100意味着子节点全部完成才算完成。可扩展支持任意条件表达式，但当前用百分比简化。单个节点无子节点时阈值忽略。), order(同一父节点下的顺序序号), remark(节点说明备注，比如操作指引) 等。
* **节点动作类型扩展：** 系统定义上述有限几种类型，通过代码枚举或字典表限制。为了将来扩展更多类型，如执行脚本等，可以设计一个接口，让每种动作类型实现处理逻辑。例如定义接口 ActionHandler，根据action\_type调用对应实现类。这部分在后端设计章节详述。数据库约束上，可对action\_type用CHECK限制合法值或用枚举域表。如：
* ALTER TABLE process\_node ADD CONSTRAINT ck\_action\_type   
   CHECK (action\_type IN ('NONE','REMOTE','EMAIL','IM','LINK','FILE'));
* 这样在数据库层保证数据有效性。
* **Process\_Instance & Node\_Instance（流程执行实例）**：为记录流程实际执行情况，引入实例表。这个概念在单次执行的计划中可能直接记录在Plan和Process\_Node表里，但区分实例可以支持**流程多次执行**或者**流程执行与定义分离**。这里设计：当计划开始执行时，创建一条Process\_Instance（可能对应Plan一对一），然后每个节点对应创建Node\_Instance逐个更新状态。
* **Process\_Instance**：字段：id, plan\_id(对应哪个计划), start\_time(实际开始执行时间), end\_time(实际完成时间, 未完成为空), status(进行中/完成), percent\_complete(完成百分比，计算子节点完成占比), 等。一个Plan通常对应一个Process\_Instance（除非允许执行多次或重试概念，那可能多条，但本设计假定一次）。
* **Node\_Instance**：字段：id, process\_instance\_id, node\_def\_id(关联哪一个流程定义节点Process\_Node), start\_time, end\_time, status(PENDING, IN\_PROGRESS, DONE, SKIPPED等), assignee\_id(实际执行人，可能继承自定义的assignee，可在执行时修改), actual\_duration(实际耗时，由end - start计算或手动填写), result(执行结果概要，例如“成功”/“失败”或备注), log(操作日志文本，如执行中记录的文字)、file\_id(如该节点上传了附件，关联File\_Object的ID，若有多个附件可有关联表或JSON存IDs)等。

Node\_Instance在执行过程中可多次更新，例如用户点击“开始”则记录start\_time和状态，在点击“完成”时记录end\_time和状态=DONE，并可填写结果和日志。对于无需人工标记的子流程节点，系统可根据threshold自动判断完成。

* **Template\_\* 系列（模板中心）：** 根据不同类型分别设计表结构：
* **Template\_Email**：邮件模板表。字段：id, name(模板名称，如“系统升级通知”), subject(邮件主题模板，可以含占位符，如“【{{customer\_name}}】系统升级通知”), body(邮件正文模板，支持简单占位变量替换或Markdown/HTML格式), to\_list(默认收件人列表，逗号分隔邮箱或引用用户角色，如ROLE\_SUPPORT表示所有维护人员), cc\_list(抄送人列表，可选), enabled(是否启用), tenant\_id(可选), created\_by 等。
* **Template\_IM**：IM消息模板表。字段类似：id, name, message(消息内容模板), endpoint(Webhook URL或Bot标识等，用于发送消息的目标), enabled, 等。不同IM平台可能需要不同字段，比如Slack webhook只需URL，Teams也类似；如果是Line或微信企业号可能需要API凭证。这里简化以Webhook为主。
* **Template\_Link**：链接/网址模板表。字段：id, name, url(网址，可包含{{}}占位符如客户ID，用于构成客户监控页面链接), description(说明，例如“跳转到客户监控系统”), enabled等。
* **Template\_Remote**：远程连接模板表。字段：id, name(名称，如“RDP远程桌面模板”), protocol(协议类型，如 RDP/SSH/VPN 等), address(地址或域名，可含变量), credential(凭据信息，例如用户名/密码或证书路径的引用; 由于安全考虑密码等敏感信息应加密存储，或使用Vault集成，这里存引用即可), launch\_params(启动参数，如RDP文件的额外配置，或SSH命令选项等), enabled等。

模板中心的表可能还包括通用字段如last\_used、use\_count统计使用等，可选。各模板表独立是为了灵活存各自特有字段。另一方案是统一Template表加类型区分，但字段多样不便管理，故分别建表。

模板与运维流程节点动作相关联（Process\_Node.action\_ref\_id引用template）。执行流程时，根据模板定义实际执行：  
- 若Email节点，后台取Template\_Email渲染邮件内容并通过SMTP发送，保存一条**Email\_History**记录：字段：id, template\_id, plan\_id, node\_instance\_id, sent\_time, to, subject, status(成功/失败), error\_msg等。  
- 若IM节点，类似**IM\_History**记录发送。  
- 若Link节点，前端直接打开URL，不一定需要历史记录，或可记录用户点击了链接的审计。  
- 若Remote节点，因无法直接由浏览器执行，可在点击时后台记录**Remote\_History**：记录何时由谁点击了该远程连接（可视为操作审计的一部分），并提供必要信息供后续线下执行。

* **File\_Object（文件对象元数据）**：该表记录上传到MinIO的文件信息。字段：id, object\_key(对象存储中的键，例如生成的UUID或路径), file\_name(原始文件名), content\_type, size(文件大小字节), bucket(所在MinIO桶名称), uploaded\_by(上传用户ID), uploaded\_at, tenant\_id(可选), biz\_type(业务类型，如标识此文件属于“客户附件”或“节点附件”之类，可帮助检索和清理), biz\_id(业务对象ID，例如customer\_id或node\_instance\_id，用于明确关联)。

为支持文件版本，可以设计**File\_Object\_Version**表：记录版本号、File\_Object\_id、object\_key、上传时间等，每次更新文件（如覆盖上传）生成新记录，并保留旧版本引用。但在运维场景下文件更多是文档上传，很少更新覆盖，也可以简单在File\_Object表加version字段，更新时保留旧object\_key在MinIO上但标记不活跃。这里提供版本表设计：  
- **File\_Object\_Version**：id, file\_object\_id, version\_no(从1开始递增), object\_key, file\_name, content\_type, size, uploaded\_at, uploaded\_by. 当前有效版本可以存在File\_Object主表里，也可以每次插入version后更新File\_Object.object\_key指向最新。实现视需要决定。

**ACL(访问控制)**：MinIO自身支持Bucket Policy和预签名URL控制访问。File\_Object表可增加is\_public(是否公有)或更细粒度权限字段。但出于安全，默认所有文件私有，仅允许通过授权请求或已签名URL访问，无需额外ACL字段。

**完整性约束：** 文件与业务的关联通过biz\_type和biz\_id，或者通过专门关联表。例如可以有 Customer\_Attachment 表记录customer\_id和file\_id，Plan\_Attachment表记录plan\_id和file\_id等。为简化，本表直接记录biz\_id，但这样一个文件只能属于一个业务对象。如果有一文件关联多个对象，需采用中间表。当前考虑的文件应用场景：客户附件(如网络拓扑图PDF)、节点执行附件(如截图、日志文件)。这些都一对一或一对多关联（一个节点可有多个文件）。多个文件可以对一个节点，所以Node\_Instance可能需要 Node\_File 表记录 node\_instance\_id 与 file\_id 关联，多文件时使用此表。类似Customer也可有Customer\_File映射。如果用File\_Object.biz\_id存，会丢失一对多能力。因此，更合理：**专门的关联表**：  
- Customer\_File(customer\_id, file\_id)  
- Node\_File(node\_instance\_id, file\_id)  
- (Plan本身如果有附件也可Plan\_File)

这样一个File\_Object可以被多个业务关联引用，但一般不会，一个文件重复引用没意义，还是多处用应各上传各的。所以unique constraint: file可以只属于一个映射。

* **User（用户）**：用户账号表。字段：id, username(登录名，唯一), email, password\_hash(密码哈希，存储加密后的密码), salt(如使用单向哈希需要盐), name(用户姓名), is\_active(是否激活), is\_locked(是否被锁定), preferred\_lang(偏好语言，如"ja"/"zh"), tenant\_id(所属租户), created\_at, last\_login\_at等。  
  **密码存储：** 使用安全哈希算法如BCrypt，密码要求一定复杂度，具体在安全章节说明。  
  **初始账户：** 系统安装时会有一个默认管理员（如用户名admin/初始密码，通过环境变量或安装时指定），存此表。
* **Role（角色）**：系统角色表。字段：id, role\_name(如ADMIN, USER, GUEST等英文标识), role\_desc(描述，如“管理员”), tenant\_id(如果支持租户自定义角色，否则全局)。预置几个固定角色在初始化SQL中插入。
* **User\_Role（用户角色关联）**：字段：user\_id, role\_id, 复合主键。一个用户可属于多个角色（RBAC灵活性，但可以限制每用户一个主要角色也可）。当用户有多个角色时，其权限为并集。
* **Permission（权限）**：系统权限点表。字段：id, perm\_key(权限键，例如MENU\_CUSTOMER\_VIEW, API\_PLAN\_CREATE, USER\_EDIT等), perm\_desc(描述，可以多语言资源标识菜单名或操作名)。Permission可以表示菜单级或功能级权限。可预先配置好并导入。
* **Role\_Permission（角色-权限关联）**：字段：role\_id, perm\_id，复合PK。通过该表赋予角色具体的权限点。前端菜单权限控制可基于perm\_key，后端接口访问也基于perm\_key注解。RBAC模型由以上四表实现。
* **Audit\_Log（审计日志）**：用于记录系统中重要数据变更操作的表。字段设计需**记录操作的各维度信息**：
* id (PK),
* timestamp (操作时间),
* user\_id (操作人用户ID，未登录操作可为空或记录系统进程ID),
* user\_name(操作人用户名，冗余存储方便查询),
* entity\_type (操作对象的实体类型，例如 "Customer", "Plan", "ProcessNode", "User" 等),
* entity\_id (操作对象的ID，如customer\_id具体值),
* action (操作类型，如 "CREATE", "UPDATE", "DELETE", "LOGIN", "ASSIGN\_ROLE" 等，可定义枚举),
* detail (操作详情描述，JSON或文本，记录字段级的变化情况),
* old\_data (变更前数据快照，JSON存储主要字段旧值, 对于新增则为空),
* new\_data (变更后数据快照，JSON存储主要字段新值, 对于删除则为空),
* request\_id (请求ID，用于串联一次请求的多条日志，trace追踪),
* ip\_address (操作者IP),
* user\_agent (操作者客户端信息)。

其中detail或old/new\_data可考虑只存改变的字段，而不是整记录，以减少冗余。但为确保追溯完整，可在必要时存整对象。出于性能考虑，可将old/new\_data定义为JSONB字段(在PostgreSQL中)方便查询特定字段变化【或将变化的每个字段拆行存Audit\_Detail表: 字段名, old, new】。设计上倾向简化，用Audit\_Log一表JSON记录。在具体实现审计时，可能通过程序拦截或数据库触发器获取old/new值。

审计日志表应有索引：如(entity\_type, entity\_id)索引，user\_id索引，时间索引等，以支持按对象或用户筛选查询。考虑数据量增长，需定期归档老审计数据。

上述实体表遵循**命名规范**：全部使用蛇形小写命名（snake\_case），表名力求简洁但意义明确。若需要区分模块，可加前缀如 cust\_ 表示客户模块，plan\_表示计划模块，但由于库中表有限，可不强制前缀，通过上下文理解。主键字段一般用 id 或 <table>\_id（例如customer\_id）一致性。外键命名 <ref\_table>\_id，如Plan表的customer\_id。多对多中间表命名一般 <table1>\_<table2>或具业务意义名如user\_role。

**数据库DDL示例（PostgreSQL）：** 根据上述模型，提供部分核心表的建表SQL片段，供直接执行：

-- Tenant (optional multi-tenant)  
CREATE TABLE IF NOT EXISTS tenant (  
 id SERIAL PRIMARY KEY,  
 name VARCHAR(100) NOT NULL UNIQUE,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW()  
);  
  
-- Customer (客户表)  
CREATE TABLE IF NOT EXISTS customer (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 code VARCHAR(50) NOT NULL UNIQUE,  
 name VARCHAR(200) NOT NULL,  
 kana VARCHAR(200), -- 名称拼音/假名  
 group\_id INT REFERENCES customer\_group(id), -- 分组（可为空）  
 region VARCHAR(100),  
 remarks TEXT,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT,  
 updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 updated\_by INT  
);  
CREATE INDEX idx\_customer\_name ON customer(name);  
CREATE INDEX idx\_customer\_group ON customer(group\_id);  
CREATE INDEX idx\_customer\_tenant ON customer(tenant\_id);  
  
-- Customer Group (客户分组表)  
CREATE TABLE IF NOT EXISTS customer\_group (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 group\_name VARCHAR(100) NOT NULL,  
 parent\_id INT REFERENCES customer\_group(id) ON DELETE SET NULL,  
 sort\_order INT DEFAULT 0  
);  
CREATE UNIQUE INDEX uidx\_group\_name\_tenant ON customer\_group(tenant\_id, group\_name);  
  
-- Tag Definition (标签定义表)  
CREATE TABLE IF NOT EXISTS tag\_def (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 tag\_name VARCHAR(50) NOT NULL,  
 color VARCHAR(20) NOT NULL, -- 颜色，如"#FF0000"或预设名称  
 icon VARCHAR(50), -- 图标标识  
 scope VARCHAR(20) NOT NULL DEFAULT 'CUSTOMER', -- 适用范围: CUSTOMER/PLAN/BOTH  
 apply\_rule TEXT, -- JSON或表达式描述自动应用规则（可为空）  
 UNIQUE(tenant\_id, tag\_name)  
);  
  
-- Customer-Tag mapping (客户标签关联表)  
CREATE TABLE IF NOT EXISTS customer\_tag (  
 customer\_id INT REFERENCES customer(id) ON DELETE CASCADE,  
 tag\_id INT REFERENCES tag\_def(id) ON DELETE CASCADE,  
 PRIMARY KEY(customer\_id, tag\_id)  
);  
CREATE INDEX idx\_customer\_tag\_tag ON customer\_tag(tag\_id);  
  
-- Custom Field Definition (自定义字段定义表)  
CREATE TABLE IF NOT EXISTS customer\_custom\_field\_def (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 field\_key VARCHAR(50) NOT NULL,  
 field\_name VARCHAR(100) NOT NULL,  
 field\_type VARCHAR(20) NOT NULL DEFAULT 'TEXT', -- 数据类型: TEXT/NUMBER/DATE/BOOL 等  
 field\_group VARCHAR(100), -- 字段分组显示，如"远程连接信息"  
 is\_required BOOLEAN NOT NULL DEFAULT FALSE,  
 is\_active BOOLEAN NOT NULL DEFAULT TRUE,  
 sort\_order INT DEFAULT 0,  
 UNIQUE(tenant\_id, field\_key)  
);  
  
-- Custom Field Value (自定义字段值表)  
CREATE TABLE IF NOT EXISTS customer\_custom\_field\_val (  
 id SERIAL PRIMARY KEY,  
 customer\_id INT NOT NULL REFERENCES customer(id) ON DELETE CASCADE,  
 field\_def\_id INT NOT NULL REFERENCES customer\_custom\_field\_def(id) ON DELETE CASCADE,  
 value TEXT,  
 updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 updated\_by INT,  
 UNIQUE(customer\_id, field\_def\_id)  
);  
CREATE INDEX idx\_field\_val\_field ON customer\_custom\_field\_val(field\_def\_id);  
  
-- Plan (运维计划表)  
CREATE TABLE IF NOT EXISTS plan (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 customer\_id INT REFERENCES customer(id) ON DELETE SET NULL, -- 可为NULL表示跨客户计划  
 title VARCHAR(200) NOT NULL,  
 description TEXT,  
 start\_time TIMESTAMP, -- 计划开始时间  
 end\_time TIMESTAMP, -- 计划结束时间或截止  
 timezone VARCHAR(50) DEFAULT 'Asia/Tokyo',  
 status VARCHAR(20) NOT NULL DEFAULT 'DESIGN', -- DESIGN/SCHEDULED/IN\_PROGRESS/COMPLETED/CANCELED  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT,  
 updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 updated\_by INT  
);  
CREATE INDEX idx\_plan\_customer ON plan(customer\_id);  
CREATE INDEX idx\_plan\_status ON plan(status);  
CREATE INDEX idx\_plan\_start\_time ON plan(start\_time);  
  
-- Plan Participant (计划参与人表)  
CREATE TABLE IF NOT EXISTS plan\_participant (  
 plan\_id INT REFERENCES plan(id) ON DELETE CASCADE,  
 user\_id INT REFERENCES "user"(id) ON DELETE CASCADE,  
 role VARCHAR(50), -- 角色/职责，如"OWNER"/"MEMBER"  
 PRIMARY KEY(plan\_id, user\_id)  
);  
  
-- Plan Tag (计划标签关联表)  
CREATE TABLE IF NOT EXISTS plan\_tag (  
 plan\_id INT REFERENCES plan(id) ON DELETE CASCADE,  
 tag\_id INT REFERENCES tag\_def(id) ON DELETE CASCADE,  
 PRIMARY KEY(plan\_id, tag\_id)  
);  
  
-- Process Node (流程节点表)  
CREATE TABLE IF NOT EXISTS process\_node (  
 id SERIAL PRIMARY KEY,  
 plan\_id INT REFERENCES plan(id) ON DELETE CASCADE,  
 parent\_id INT REFERENCES process\_node(id) ON DELETE CASCADE,  
 name VARCHAR(200) NOT NULL,  
 assignee\_id INT REFERENCES "user"(id),  
 expected\_duration INT, -- 预计时长(分钟)  
 action\_type VARCHAR(20) NOT NULL DEFAULT 'NONE',  
 action\_ref\_id INT, -- 引用相应模板ID（如template\_email.id等，视action\_type而定）  
 threshold INT DEFAULT 100, -- 完成阈值 (%)  
 order INT NOT NULL DEFAULT 0, -- 顺序  
 remark TEXT  
);  
-- 限制action\_type在预定义范围  
ALTER TABLE process\_node ADD CONSTRAINT ck\_process\_node\_action   
 CHECK (action\_type IN ('NONE','REMOTE','EMAIL','IM','LINK','FILE'));  
CREATE INDEX idx\_node\_plan ON process\_node(plan\_id);  
CREATE INDEX idx\_node\_parent ON process\_node(parent\_id);  
  
-- Process Instance (流程执行实例表)  
CREATE TABLE IF NOT EXISTS process\_instance (  
 id SERIAL PRIMARY KEY,  
 plan\_id INT REFERENCES plan(id) ON DELETE CASCADE,  
 start\_time TIMESTAMP,  
 end\_time TIMESTAMP,  
 status VARCHAR(20) DEFAULT 'IN\_PROGRESS', -- IN\_PROGRESS/COMPLETED/ABORTED  
 percent\_complete INT DEFAULT 0  
);  
CREATE INDEX idx\_procinst\_plan ON process\_instance(plan\_id);  
  
-- Node Instance (流程节点执行表)  
CREATE TABLE IF NOT EXISTS node\_instance (  
 id SERIAL PRIMARY KEY,  
 process\_instance\_id INT REFERENCES process\_instance(id) ON DELETE CASCADE,  
 node\_def\_id INT REFERENCES process\_node(id) ON DELETE CASCADE,  
 assignee\_id INT REFERENCES "user"(id),  
 start\_time TIMESTAMP,  
 end\_time TIMESTAMP,  
 status VARCHAR(20) DEFAULT 'PENDING', -- PENDING/IN\_PROGRESS/DONE/SKIPPED  
 result VARCHAR(100), -- 简要结果  
 log\_text TEXT, -- 执行日志文字  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT,  
 updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 updated\_by INT  
);  
CREATE INDEX idx\_nodeinst\_proc ON node\_instance(process\_instance\_id);  
CREATE INDEX idx\_nodeinst\_node ON node\_instance(node\_def\_id);  
  
-- Template Email (邮件模板表)  
CREATE TABLE IF NOT EXISTS template\_email (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 name VARCHAR(100) NOT NULL,  
 subject VARCHAR(200) NOT NULL,  
 body TEXT NOT NULL,  
 to\_list TEXT, -- 逗号分隔收件人邮件或用户标识，如"<user:123>,<email:xx@yy>"  
 cc\_list TEXT,  
 enabled BOOLEAN NOT NULL DEFAULT TRUE,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT  
);  
-- 模板名称在租户内唯一  
CREATE UNIQUE INDEX uidx\_tmpl\_email\_name ON template\_email(tenant\_id, name);  
  
-- Template IM (IM消息模板表)  
CREATE TABLE IF NOT EXISTS template\_im (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT,  
 name VARCHAR(100) NOT NULL,  
 message TEXT NOT NULL,  
 endpoint TEXT, -- Webhook URL 或 Bot标识  
 enabled BOOLEAN NOT NULL DEFAULT TRUE,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT  
);  
CREATE UNIQUE INDEX uidx\_tmpl\_im\_name ON template\_im(tenant\_id, name);  
  
-- Template Link (链接模板表)  
CREATE TABLE IF NOT EXISTS template\_link (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT,  
 name VARCHAR(100) NOT NULL,  
 url TEXT NOT NULL,  
 description TEXT,  
 enabled BOOLEAN NOT NULL DEFAULT TRUE,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT  
);  
CREATE UNIQUE INDEX uidx\_tmpl\_link\_name ON template\_link(tenant\_id, name);  
  
-- Template Remote (远程连接模板表)  
CREATE TABLE IF NOT EXISTS template\_remote (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT,  
 name VARCHAR(100) NOT NULL,  
 protocol VARCHAR(20) NOT NULL, -- 协议类型：RDP/SSH/VPN等  
 address VARCHAR(200) NOT NULL, -- 地址或主机  
 credential TEXT, -- 凭据（加密存储或引用）  
 launch\_params TEXT, -- 启动参数/配置  
 enabled BOOLEAN NOT NULL DEFAULT TRUE,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT  
);  
CREATE UNIQUE INDEX uidx\_tmpl\_remote\_name ON template\_remote(tenant\_id, name);  
  
-- File Object (文件对象表)  
CREATE TABLE IF NOT EXISTS file\_object (  
 id SERIAL PRIMARY KEY,  
 bucket VARCHAR(50) NOT NULL,  
 object\_key VARCHAR(200) NOT NULL UNIQUE, -- 存储对象键（UUID路径等）  
 file\_name VARCHAR(200) NOT NULL,  
 content\_type VARCHAR(100),  
 size BIGINT,  
 biz\_type VARCHAR(50), -- 业务类型，如 'CUSTOMER\_ATTACHMENT','NODE\_ATTACHMENT'  
 biz\_id INT, -- 业务对象ID，根据biz\_type解释  
 uploaded\_by INT,  
 uploaded\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 version INT DEFAULT 1 -- 当前版本号  
);  
CREATE INDEX idx\_file\_biz ON file\_object(biz\_type, biz\_id);  
  
-- User (用户表)  
CREATE TABLE IF NOT EXISTS "user" (  
 id SERIAL PRIMARY KEY,  
 tenant\_id INT REFERENCES tenant(id) ON DELETE CASCADE,  
 username VARCHAR(50) NOT NULL UNIQUE,  
 email VARCHAR(100) NOT NULL UNIQUE,  
 password\_hash VARCHAR(100) NOT NULL,  
 salt VARCHAR(50),  
 name VARCHAR(100),  
 preferred\_lang VARCHAR(10) DEFAULT 'ja',  
 is\_active BOOLEAN NOT NULL DEFAULT FALSE,  
 is\_locked BOOLEAN NOT NULL DEFAULT FALSE,  
 last\_login\_at TIMESTAMP,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 created\_by INT  
);  
  
-- Role (角色表)  
CREATE TABLE IF NOT EXISTS role (  
 id SERIAL PRIMARY KEY,  
 role\_name VARCHAR(50) NOT NULL UNIQUE,  
 role\_desc VARCHAR(100)  
);  
-- Insert basic roles  
INSERT INTO role(role\_name, role\_desc) VALUES   
 ('ADMIN','管理员'),('USER','普通用户')   
ON CONFLICT DO NOTHING;  
  
-- User\_Role (用户角色关联)  
CREATE TABLE IF NOT EXISTS user\_role (  
 user\_id INT REFERENCES "user"(id) ON DELETE CASCADE,  
 role\_id INT REFERENCES role(id) ON DELETE CASCADE,  
 PRIMARY KEY(user\_id, role\_id)  
);  
  
-- Permission (权限点表)  
CREATE TABLE IF NOT EXISTS permission (  
 id SERIAL PRIMARY KEY,  
 perm\_key VARCHAR(100) NOT NULL UNIQUE,  
 perm\_desc VARCHAR(200)  
);  
-- Role\_Permission (角色权限关联)  
CREATE TABLE IF NOT EXISTS role\_permission (  
 role\_id INT REFERENCES role(id) ON DELETE CASCADE,  
 perm\_id INT REFERENCES permission(id) ON DELETE CASCADE,  
 PRIMARY KEY(role\_id, perm\_id)  
);  
  
-- Audit Log (审计日志表)  
CREATE TABLE IF NOT EXISTS audit\_log (  
 id BIGSERIAL PRIMARY KEY,  
 timestamp TIMESTAMP NOT NULL DEFAULT NOW(),  
 user\_id INT,  
 user\_name VARCHAR(100),  
 entity\_type VARCHAR(50),  
 entity\_id VARCHAR(50),  
 action VARCHAR(20) NOT NULL,  
 detail TEXT, -- 可存人读描述  
 old\_data JSONB,  
 new\_data JSONB,  
 request\_id VARCHAR(100),  
 ip\_address VARCHAR(50),  
 user\_agent TEXT  
);  
CREATE INDEX idx\_audit\_entity ON audit\_log(entity\_type, entity\_id);  
CREATE INDEX idx\_audit\_user ON audit\_log(user\_id);  
CREATE INDEX idx\_audit\_time ON audit\_log(timestamp);

以上DDL涵盖了主要实体，创建了必要索引和约束。某些字段如 created\_by, updated\_by 可加FOREIGN KEY引用user(id)以保证数据完整性，但也可出于灵活和性能考虑不加（因为删除用户可能不想连带删大量业务数据）。审计日志的旧新数据使用JSONB类型，方便查询变化细节，例如可以利用PostgreSQL JSON查询语法查找特定字段的变动。

**关键字段和约束说明：**

* **自定义字段模型：** 通过 customer\_custom\_field\_def 和 customer\_custom\_field\_val 实现。可扩展到其他实体类型（例如Plan也可以有plan\_custom\_field\_xxx表），当前需求主要对客户信息扩展。类型字段field\_type可约束值格式，如为NUMBER时在保存val时尝试转换为数字验证。
* **标签颜色/图标规范：** 颜色存储为HEX字符串（#RRGGBB）或采用AntD预设色名（如“magenta”、“volcano”）。图标可存Ant Design图标组件名或自定义icon路径，在前端渲染对应的小图标。
* **流程阈值（threshold）：** 对于有子节点的节点，threshold=100表示要求所有子节点完成该节点才算完成；如果设置为小于100，如50，则子节点完成达到50%即认为父节点完成，可以进入后续流程。这类似于并行任务的“通过条件”，但本系统节点还是顺序执行，只是可以提前跳过剩余子任务。所以threshold<100的场景意味着一些子任务可选，达到阈值即可跳转下一个节点。实现上，当父节点检查子节点完成数量/比例达到阈值，就将未完成的子节点标记为SKIPPED并标记父节点完成。
* **节点动作类型枚举及扩展机制：** 当前类型包括REMOTE, EMAIL, IM, LINK, FILE几种。如需扩展新的动作类型（例如SCRIPT脚本执行），可：在代码中新增对应ActionHandler实现类，在数据库允许新的action\_type值（update CHECK或domain表），界面提供配置入口。这种插件式扩展要求设计ActionHandler接口统一方法，如 execute(NodeInstance ni)，不同类型有不同实现。框架可通过Spring Bean动态发现新实现或配置文件加载，以便后续扩展无需改动核心流程逻辑。
* **DDL命名规范：** 整体采用小写下划线。主键一般采用序列（SERIAL），PostgreSQL序列名自动生成。表、约束、索引命名控制在30字符左右以防超长。可以通过在DDL中定义模式search\_path或特定schema，不过当前全部放public schema。
* **索引和唯一性：** 所有外键字段都建索引以优化关联查询；对经常按时间、状态筛选的列也建索引如plan.start\_time；unique保证不重复的有customer.code/email/username等。
* **审计触发器雏形：** 在DDL中未具体给出，但可设想对关键表的UPDATE/DELETE操作写触发器函数，将old和new记录插入audit\_log。例如对customer表的AFTER UPDATE触发：比较每个字段，将变化组装JSON记录。考虑工作量和灵活性，也可以选择**不用数据库触发器**，而在应用层通过AOP统一处理审计，这样更容易拿到user\_id、request\_id等。详见后端设计中审计部分。

至此，数据模型设计支持**运行期扩展字段**（通过Custom Field）、**分类与标签**（通过Group和Tag）、**流程与计划**（Plan/Node/Instance模型），以及**模板**和**文件存储**等需求。模型充分考虑了一对多/多对多关系，满足功能要求且具备扩展性。

## 5. 后端详细设计

后端基于Spring Boot进行分层设计，并考虑将业务模块解耦为微服务。**服务划分**可采取**渐进式**策略：即初期以单体应用实现全部模块功能，但在内部按照未来微服务拆分的边界组织代码包和模块，预留拆分接口。一旦需要水平扩展或独立部署某模块，可将其代码抽取成独立服务。按照第2章模块划分，本项目可考虑以下服务（或模块）：

* **Auth Service（认证服务）**：负责用户登录认证、JWT令牌签发验证、权限校验等。也包括用户注册、密码管理等用户相关接口（可与User服务合并）。如果拆分独立服务，可使用Spring Security OAuth/OIDC实现单点登录。当前可内嵌在API网关或User服务中。
* **User Service（用户管理服务）**：提供用户和角色、权限的CRUD接口，供管理员管理用户，以及用户自助注册/激活等流程。Auth服务在验证JWT或登录时会调用User服务查用户信息。
* **Customer Service（客户管理服务）**：管理客户及分组、标签、自定义字段等所有客户相关功能。对外提供REST接口，例如获取客户列表、编辑客户详情、分组管理、标签管理、自定义字段定义管理等。
* **Plan Service（计划服务）**：管理运维计划及其流程设计/执行。Plan服务内部会处理Plan、Process\_Node、Process\_Instance、Node\_Instance等逻辑，包括发布计划、开始执行、更新节点状态等。
* **Template Service（模板中心服务）**：管理所有类型模板，并提供模板渲染和执行接口。例如根据模板发送邮件/IM、提供远程连接参数等。外部集成（SMTP发送、Webhook调用等）也由该服务承担，使之成为系统与外部服务交互的网关。
* **File Service（文件服务）**：封装MinIO的文件存储操作，提供文件上传（生成预签URL）、下载（生成访问URL或代理下载）接口，以及文件元数据维护。作为独立服务有助于隔离文件传输流量和权限控制。
* **Audit Service（审计日志服务）**：集中处理审计日志写入和查询。如果使用异步队列（如Kafka）收集日志，Audit服务可消费日志事件写数据库。初期也可不独立服务，而是在各服务中直接写audit\_log表，通过统一AOP组件实现。查询审计日志可以归到User/Admin服务权限下处理。

上述服务可通过Spring Cloud Netflix或Spring Cloud Alibaba体系进行注册发现和网关路由。如果在单体应用实现，则按照功能划分不同的Spring @Component/@Service包。**集中配置**方面，引入Spring Cloud Config Server将配置（如DB连接、第三方服务凭据、多语言资源等）集中管理是可选项。对于国际化词条资源，可以利用Config Server存储properties/yaml并在各服务启动时拉取，也可以将词条存DB由某服务提供API。下面详细说明**国际化资源集中管理**。

### 国际化（i18n）设计

后端作为多语言资源的单一来源，使用Spring自带的MessageSource机制或自定义实现，以便在服务端和前端共享翻译。Spring Boot通常通过 messages\_xx.properties 文件来提供国际化文本[[3]](https://phrase.com/blog/posts/database-stored-messages-for-i18n-in-spring-boot/#:~:text=The%20most%20common%20approach%20to,is%20not%20an%20optimal%20approach)。为了让终端用户或实施人员可以**在运行时**添加新语言或修改文案，我们设计**数据库驱动的国际化词条存储**[[4]](https://phrase.com/blog/posts/database-stored-messages-for-i18n-in-spring-boot/#:~:text=Hence%2C%20we%20are%20going%20to,existing%20localized%20messages%20at%20runtime)。方案如下：

* **集中词条库：** 建立一个词条表（如i18n\_message），字段：id, lang(语言, 如 'ja','zh'), code(资源键), text(文本)。初始填入系统所有UI和消息的中日文对照。也可以扩展字段如module分类，但通过code前缀区分模块亦可。
* **后端加载：** 在Auth/User服务或Config服务实现一个自定义的 MessageSource bean，即 Spring的消息源接口。应用启动时，从数据库加载所有词条到内存（可缓存），并实现 getMessage(code, locale) 时查询对应语言。如果未找到，返回默认语言(日文)文本或code本身。
* **增量更新：** 对于新词条或更新的翻译，可通过管理界面或直接DB操作插入/更新 i18n\_message 表。服务端应提供刷新机制：简单可在更新后清空缓存或者记录版本号。更优可每条记录有 timestamp，前端请求时提供上次更新时间，服务端返回此后有变动的条目。实现上，可在 i18n\_message 表上维护 last\_modified 列，并在API /api/v1/i18n?since=<timestamp> 中返回所有大于since的记录，供前端更新。
* **前端获取：** 前端APP启动时，调用如 GET /api/v1/i18n?lang=ja 拉取日文所有词条（或如果已缓存则带If-Modified-Since）。后端返回 JSON 列表或键值对集合。前端存储在内存或localStorage。切换语言时再调对应语言接口。
* **缓存及版本：** 在后端MessageSource实现中，可将数据缓存在内存Map。当有更新操作时，通过Spring的RefreshScope或者主动调用刷新Bean方法清除缓存。前端通过HTTP缓存控制(ETag)减少不必要的拉取。如配置Cache-Control让浏览器缓存一段时间，但考虑到需要及时更新，用ETag基于版本号比较合适。版本号可在服务端维护一个全局i18n版本，每次更改+1。前端存该版本，下一次请求附带If-None-Match: "<ver>"头，服务端对比，无变化则返回304。

**资源命名规范：** 约定code采用点分命名如 customer.list.title。domain一般对应模块或页面，例如customer模块下list页面的title。动词类消息（如错误、提示）可按功能前缀：如error.login.failed，info.customer.save\_success等。这让开发人员易于定位。文案中尽量避免包含格式变量，如果需要变量，使用消息格式化功能（Spring MessageFormat 或自行处理模板标记）。例如邮件模板内容中如插入客户名，则模板文本可用占位符 {0}, 在getMessage时传参数数组。对于前端直接使用的UI词条，通常无需复杂格式，直接替换或字符串模板在前端完成更灵活。

**默认与新语言接入：** 默认提供日语(ja)和中文(zh)两套资源文件/数据（中文用于实施/调试）。当需要增加新语言，比如英文en，流程为：在i18n\_message插入一批lang='en'的翻译记录（可从日文翻译得到），或者提供Excel给翻译后批量导入DB。系统UI提供语言切换选项（可自动根据浏览器Accept-Language或用户设置选择默认语言）。

**示例词条：** 以下给出一个词条JSON示例对照：

{  
 "customer.list.title": "顧客一覧",  
 "customer.list.search": "検索...",  
 "customer.detail.basic\_info": "基本情報",  
 "customer.detail.remote\_info": "リモート接続情報",  
 "plan.status.design": "設計中",  
 "plan.status.in\_progress": "実行中",  
 "plan.calendar.view.day": "日ビュー",  
 "plan.calendar.view.week": "週ビュー",  
 "user.login.username": "ユーザー名",  
 "user.login.password": "パスワード",  
 "user.login.submit": "ログイン",  
 "message.save.success": "保存しました。"  
}

(日语默认) 对应中文资源：

{  
 "customer.list.title": "客户列表",  
 "customer.list.search": "搜索…",  
 "customer.detail.basic\_info": "基本信息",  
 "customer.detail.remote\_info": "远程连接信息",  
 "plan.status.design": "设计中",  
 "plan.status.in\_progress": "执行中",  
 "plan.calendar.view.day": "日视图",  
 "plan.calendar.view.week": "周视图",  
 "user.login.username": "用户名",  
 "user.login.password": "密码",  
 "user.login.submit": "登录",  
 "message.save.success": "保存成功。"  
}

新功能上线时，开发需添加相应词条到词条库（至少日文），中文可后补。**词条发布流程**：如有一新页面，需要10条新文案，开发在设计文档或Excel列出词条key和日文文本，交由翻译提供中文文本，录入数据库。版本上线时在SQL或配置中包含这些新词条。对于已部署系统，也可以通过后台管理提供的“词条管理”界面在线添加/修改词条，再触发前端刷新。以确保研发、测试按照最新词条进行UI检查。

### MyBatis持久层规范

后端与数据库交互采用 MyBatis 框架。设计上，将各模块的数据库操作接口划分到相应的Mapper，实现DAO层功能。规范如下：

* **包结构：** 建议按模块分包，例如 com.bobmta.customer.mapper 存放客户模块的Mapper接口和对应XML；com.bobmta.plan.mapper 放计划流程相关；com.bobmta.user.mapper 等。领域模型类(Entity/Domain对象)可在com.bobmta.customer.entity等包定义。也可使用MyBatis Generator或MyBatis-Plus减少样板代码。
* **命名约定：** Mapper接口以表或领域名命名，方法名表达动作。例如 CustomerMapper，方法 selectById, insertCustomer, updateCustomer 等。SQL较复杂时使用XML文件定义，简单CRUD可用注解。
* **XML映射：** 将SQL写在Mapper XML中，与Mapper接口名对应。配置驼峰自动映射(column下划线-> property驼峰)。任何输入值使用#{}参数占位符防SQL注入。
* **分页查询：** 对于列表接口需要分页，采用两种方式：
* 使用PageHelper插件，在调用Mapper方法前启动分页模式，例如：
* PageHelper.startPage(pageNo, pageSize);  
  List<Customer> list = customerMapper.queryByName(name);  
  PageInfo<Customer> pageInfo = new PageInfo<>(list);
* 这样插件会自动修改SQL加limit并返回total等。
* 手工在SQL加 LIMIT #{limit} OFFSET #{offset}，并提供COUNT查询获取total。对于数据量大表，可以采用游标或搜索深分页优化，但暂不考虑。
* **动态查询：** 通过MyBatis <where> 和 <if> 元素实现动态条件。比如客户列表搜索，可以写：
* <select id="selectCustomers" resultType="Customer">  
   SELECT \* FROM customer   
   <where>  
   <if test="groupId != null"> AND group\_id = #{groupId} </if>  
   <if test="nameKeyword != null"> AND name ILIKE CONCAT('%', #{nameKeyword}, '%') </if>  
   <if test="tagId != null">  
   AND id IN (SELECT customer\_id FROM customer\_tag WHERE tag\_id = #{tagId})  
   </if>  
   </where>  
   ORDER BY name  
  </select>
* 这样根据传入参数拼接条件SQL。MyBatis能处理null参数自动跳过。
* **审计字段自动填充：** 表中通用的 created\_at/updated\_at 和 created\_by/updated\_by 字段，在INSERT或UPDATE时应自动设置。方案：
* 使用MyBatis Interceptor插件，在执行SQL前拦截判断并补全这些字段。对于INSERT语句，如果SQL未赋值 created\_at，则附加NOW()默认可以满足，也可以在数据库默认值中已定义。对于updated\_at，可在UPDATE前自动set为NOW()。
* 或者简单采用MyBatis-Plus的MetaObjectHandler机制，可以统一指定填充策略。
* 另外可以通过数据库trigger在更新时set new.updated\_at = now()，但应用层处理更直观。
* **乐观锁/悲观锁：** 为防止并发修改冲突，可在部分表使用乐观锁策略。例如为重要表（customer, plan等）增加 version 字段，每次UPDATE where version = oldVersion，update成功说明没人抢占更新，否则失败提示用户刷新。MyBatis支持在XML里条件带上 AND version = #{version} 以及设置 version = version + 1。悲观锁则在需要强一致场景采用：MyBatis执行SELECT ... FOR UPDATE获取行锁。使用时确保在事务范围内。尽可能减少锁定粒度。
* **事务管理：** Spring Boot自动配置MyBatis与Spring事务集成，可用 @Transactional 注解在服务层方法上管理事务。跨服务的操作暂假定在单库内完成，如拆微服务则需要分布式事务或柔性事务支持。当前模型基本上一致性要求高的操作都在单服务内（如更新客户及其字段、计划及节点），使用本地事务即可。

### Spring Cloud配置与集中资源

如前述，我们可以使用Spring Cloud Config Server集中管理配置和资源，包括i18n词条。若采用Config Server方案：在Config仓库中维护各服务的application.yaml，还可以维护一份 messages\_ja.properties, messages\_zh.properties 提供词条。各服务在启动时从Config Server获取对应文件。这样词条修改只需更新配置仓库并发送refresh。缺点是改动频繁时重启或刷新较重，不如数据库灵活。因此**推荐**：**配置中心主要管理敏感配置**（如数据库URL、MinIO密钥、SMTP配置等），而**国际化词条使用数据库加缓存方案**。

具体配置项举例：  
- 数据库连接URL、用户名密码  
- MinIO服务的URL, AccessKey/SecretKey  
- SMTP服务器地址, 端口, 账号, 密码  
- 安全配置如 JWT签名私钥/公钥路径 或对称密钥  
- 第三方IM webhook基础URL等  
- ElasticSearch/Opensearch地址（如果日志直写ES的话）  
- ...

Config Server可存这些在不同profile的yml里，例如 application-prod.yml 等。服务启动时由 Spring Cloud Config 自动注入。开发测试环境也可直接用本地 application.yml 跳过Config Server复杂度。

### 流程引擎详细设计

运维流程引擎是平台核心特色。设计思路是轻量的可配置流程，而非完整BPM引擎。**特点**：顺序执行为主，支持简单的子流程和完成条件。下面从设计和执行两个模式解释：

* **流程设计模式：**
* **新增/编辑流程节点：** 用户通过前端UI对某计划的流程进行设计，包括添加节点、删除节点、设置父子层级、调整先后顺序、编辑节点属性。后端提供对应API，如 POST /plans/{id}/nodes 添加节点，DELETE /plans/{id}/nodes/{nodeId} 删除节点，PUT /plans/{id}/nodes/{nodeId} 更新节点属性，以及可能 PATCH /plans/{id}/nodes/reorder 批量调整顺序。实现上，可简化为前端将最终设计的节点树序列化提交（比如数组表示节点及层级），后端接收后删旧节点重建；或者逐步操作。考虑需要保留节点ID以供后续实例引用，推荐**逐步编辑**：每次操作更新数据库对应记录。顺序调整可以通过更新 order 字段或 parent\_id 来变换层级和顺序。
* **子流程处理：** parent\_id字段非空表示子节点。允许多级嵌套。UI上体现为缩进结构。需注意**阈值**：在设计时，如果一个节点下子节点并行执行，可设置threshold < 100，使之成为“或”关系子任务。例如有4个子任务，threshold=75%，则执行任意3个就算满足继续。UI应提示这一条件。
* **保存/发布：** 设计模式下修改直接保存到Process\_Node表，Plan状态仍为DESIGN。只有当用户选择“发布计划”（或到达计划开始时间自动发布）时，Plan状态转为SCHEDULED/IN\_PROGRESS开始执行。发布操作可以在Plan表做标记，也可能复制当前节点定义快照以防后续改动。简单起见，不深拷贝，直接用Process\_Node记录作为执行依据，并冻结对其再编辑（即Plan状态非DESIGN时，后端拒绝变更节点定义API）。
* **流程执行模式：**
* **执行启动：** 当Plan进入执行状态（IN\_PROGRESS），需创建一条Process\_Instance记录（如果在Plan.publish时就创建也可以）。Process\_Instance关联Plan并初始化start\_time。当Plan实际开始执行时填入start\_time（可能与Plan.start\_time计划值不同）。
* **节点状态流转：** 执行中，系统跟踪当前节点。可以按顺序逐个激活节点：一种做法是**显式当前节点**：在Plan表或Process\_Instance记录一个current\_node\_id字段。但由于可以通过Node\_Instance状态推算当前节点（第一个PENDING的节点即下一个要执行），也可以不额外存。每次用户点击“开始节点”按钮，调用API如 POST /plans/{id}/nodes/{nodeId}/start，后端检查该节点前序是否完成且状态是PENDING，然后：
  + 写对应Node\_Instance的start\_time=now, status->IN\_PROGRESS, assignee=当前用户（或预先指派人）。
  + 返回成功后前端UI高亮该节点为当前执行中。
* 用户完成操作后点击“完成节点”按钮，对应 API POST /plans/{id}/nodes/{nodeId}/complete，请求体可附带实际结果和日志、上传的文件ID等。后端处理：  
  - 将 Node\_Instance 的end\_time=now, status->DONE, result/log填入；  
  - 检查该节点是否有父节点，以及父节点的threshold：如果父节点存在且child全部完成或达阈值，则将父节点对应Node\_Instance标记DONE（若该父节点本身没有手动执行过程，它的Node\_Instance可能在创建时就存在未开始，或可以动态创建，当第一个子执行时创建父的Node\_Instance status=PENDING）。为了简化实现，可以对所有节点预先创建Node\_Instance记录树，父节点的状态开始为PENDING，子完成时update父状态。  
  - 同时检查Plan的整体完成度：如果所有节点DONE或被跳过，则设置Process\_Instance.end\_time=now, status->COMPLETED；Plan.status->COMPLETED。
* **并行任务处理：** 实际上我们的引擎仍是顺序驱动，但对于有子节点的节点，可以**并行执行**子节点（没有严格同步锁，只要子任务Ready都可由不同人同时做）。阈值条件允许部分子任务未做也算完成。实现时，可允许多个Node\_Instance同时IN\_PROGRESS。当阈值达到时，可以将剩余未完成子节点标记SKIPPED并把父节点标Done。这里有复杂度，但如果需要可以实现：当任一子节点完成触发父节点阈值检查，若达到条件：  
  - 把父Node\_Instance.end\_time设为now, status=DONE；  
  - 自动将其他尚未DONE的子节点 Node\_Instance.status -> SKIPPED（表示这些任务被跳过未执行）。  
  - 之后父节点后续节点可以执行。
* 前端应在某节点满足阈值后，标识未做的子任务为灰色或跳过状态。
* **节点执行的日志和附件：** 用户在执行过程中，可以填写操作日志（文本）或截图/文件上传。前端应提供在当前节点界面内一个日志输入框和上传文件控件。日志可多次记录，可在完成时一并提交，也可通过单独接口实时保存（例如 PATCH /node\_instance/{id} with log append）。文件上传通过File Service预签名，得到file\_id后，调用 POST /plans/{id}/nodes/{nodeId}/files 附加文件。后台在Node\_File表插入记录，并把file\_id记在Node\_Instance.log\_text或关联里。审计日志应记录用户上传了什么文件。
* **执行过程中的调整：** 有时执行时可能需更改责任人或调整顺序（比如某步失败需要插入补救步骤）。由于执行模式默认冻结流程，不建议中途修改定义。但可提供**应急措施**：如管理员可暂停计划，将Plan状态退回DESIGN，对剩余未执行部分调整，然后再恢复执行。实现上复杂，这里暂不展开，只提示可行性。
* **动作执行：** 在执行模式，当一个节点具备action\_type时，系统可辅助执行：
  + **REMOTE**: 点击节点的“启动远程连接”按钮时，后台可以根据节点的action\_ref\_id找到Template\_Remote记录，检查protocol，如果是RDP，则生成一个临时的.rdp文件内容（包含地址、用户名等），返回给前端触发浏览器下载；如果是SSH，可构造 ssh://user@host 的URL尝试调用本地ssh协议（需要客户端注册该URL scheme）；VPN之类可能无统一协议，可能仅提示连接地址和凭据，让用户手动使用VPN客户端。因为浏览器直接执行远程连接有限制，设计上**预留**Remote动作的接口供将来对接。例如，将来开发一个本地Agent，可以通过WebSocket/本地服务来控制建立连接。当前版本则**仅记录**点击行为和提供信息，不真正自动连。后台Remote\_History记录操作者和时间。
  + **EMAIL**: 执行节点若是EMAIL类型，则可以在状态变为IN\_PROGRESS时由系统自动发送邮件。实际执行可在用户点击“开始”或“完成”时触发一次发送。如定义在完成节点时发报告，那么在complete接口处理时，通过TemplateService取模板渲染（填充变量，比如客户名=Plan.customer.name，计划名=Plan.title等），然后通过配置的SMTP服务器发送邮件【可能用JavaMail或Spring MailSender】。结果（成功或异常）反馈给前端，异常时可重试。无论成功与否，Audit日志记录“发送邮件: 模板X to Y成功/失败”。同时插入Email\_History记录。
  + **IM**: 类似EMAIL，用HTTP请求发送消息。若配置的是简单Webhook URL，直接发POST请求携带message文本。IM不保证送达（可不处理响应或简单记录HTTP状态）。
  + **LINK**: 此动作只需前端打开链接（可能带一些Plan或Customer参数）。因此当用户点击“打开链接”按钮，后台审计记录一下，直接由前端用window.open(template.url替换变量)跳转。
  + **FILE**: 这种类型表示“打开文件夹/文件”。Web浏览器无法访问本地文件系统或网络共享直接。可能方案：提供文件路径显示，用户手动去访问；或者使用特定协议 file://（但浏览器通常不允许file://访问非本站文件）。因此这个动作更多是提示或集成点。如果后续开发客户端组件，则可通过调用客户端打开路径。当前仅作为提醒，或要求用户下载某文件。
* 为保持架构完整，每种Action处理可封装在后端 TemplateService 中。例如 TemplateService有方法sendEmail(templateId, context), sendIM(templateId, context), getLink(templateId, context) 等。Remote的执行可有 placeholder接口 doRemote(templateId, context) 目前只是记录log。通过这种封装，新增类型比如SCRIPT时，只需添加对应handler方法、数据库记录和前端按钮，无须修改流程主逻辑。

### 运维计划与日历同步

运维计划模块除流程外，还负责提供计划的时间维度信息。针对日历集成功能，需要实现：

* **ICS生成**: 提供生成iCalendar (.ics)文件的功能，可针对单个计划或一组计划生成。使用库如 iCal4j 或自己拼字符串。基本思路：
* ICS文件是文本，包含事件属性：UID, DTSTART, DTEND, SUMMARY, DESCRIPTION 等。
* 单次导出单个计划：当用户点击“导出ICS”按钮，对应后端GET /plans/{id}/ics，返回Content-Type: text/calendar 响应。文件内容包括该计划的摘要(title)、开始结束时间、描述(可含流程信息或链接回本系统)等。
* 订阅链接：提供一个URL如 /calendar/{userId}/subscription.ics，当Outlook/Google以webcal方式订阅它时，每次它被请求都动态生成包含用户可见的所有未来计划事件的ICS feed。这样无需每个计划逐个订阅。实现上，可以有CalendarController，GET方法读取数据库中未完成的计划（或一段时间范围内）生成ICS。UID可用计划ID结合租户/用户，确保唯一。
* **Outlook集成注意**: Outlook订阅网络日历刷新频率较低[[1]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=Note%3A%C2%A0When%20you%20subscribe%20to%20a,take%20more%20than%2024%20hours)，用户在第一次订阅可看到当前事件，但后续更新不会马上反映，有时一天以上才更新[[1]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=Note%3A%C2%A0When%20you%20subscribe%20to%20a,take%20more%20than%2024%20hours)。需要在使用手册中说明这一限制。如果要求较高实时性，可考虑Exchange日历实时推送或Graph API集成，但那超出本项目范围。
* **重复任务**: 需求提到可留扩展，不实现复杂重复。这可记录Plan.repeat\_rule，如 “FREQ=YEARLY;COUNT=5” 等ICS RRULE。如果前端提供UI选择重复模式（每天/每周/每月），可以在Plan创建时自动生成多条计划记录（分开存），或者存一条Plan带repeat\_rule，在生成ICS或显示时去展开。简化设计：暂不做自动重复生成，由用户手动复制计划。repeat\_rule字段保留供显示或提醒用途。
* **时区处理**: 所有时间在数据库可以存UTC或本地。在日本部署，可考虑存 Asia/Tokyo 本地时间，但更规范是UTC存储、附带timezone字段。ICS中需要正确的TZID。如果Plan有timezone字段，就用它；没有则默认为服务器时区。UI显示方面，应考虑用户所在时区。如果所有用户都在JP，可不考虑切换。如果要国际化，将来可按用户偏好时区转换。
* **冲突检测**: 当创建或发布计划时，检查是否冲突：例如同一客户在同一时间已经有计划，或者同一参与人员已经在别的计划中执行。这种冲突可以在PlanService里通过查询时间重叠来判断，若有则返回警告或错误。实现：SQL检查存在 WHERE customer\_id = X AND status IN (SCHEDULED,IN\_PROGRESS) AND timeslot overlap. PostgreSQL支持 daterange 用&&判定区间重叠。不过会复杂。简单可判断 start\_time 在别的计划 start~end 范围内 或 vice versa。冲突处理策略：允许但提示，或直接不允许。具体产品决策，此处可实现为产生一个警告列表。

### 文件存储与MinIO集成

File服务使用MinIO作为后端存储。设计要点：

* **Bucket策略：** MinIO需要创建一个或多个bucket（桶）来存放文件。可以按功能或租户分桶。两种策略：
* **单一Bucket**：创建一个如bobmta的bucket，所有文件对象放其中，通过object\_key区分路径，例如 customer/<customerId>/<filename>。管理方便，但权限难细分，如果想对不同租户隔离，可以将租户id作为key前缀（MinIO的policy可以基于前缀限制用户）。
* **多Bucket**：按租户或文件类型建多个桶，如每个租户一个bucket(tenant-<id>), 或一个customer-files桶和plan-files桶等。MinIO允许对bucket设置独立策略。

考虑实现简单，**采用单Bucket**方案，命名比如mta-files。对每个文件，object\_key可以这样生成：<tenantId>/<bizType>/<uuid>.<ext>，如1/customer\_attachment/550e8400-e29b-41d4-a716-446655440000.pdf。也可以不分层，直接uuid命名但加业务元数据在DB。加前缀有利于肉眼辨识和管理。

* **上传流程：** 为避免文件大流量经过应用服务器，采用**前端直传**MinIO的模式：
* 前端向File服务请求上传授权。调用POST /files/upload-url（需登录），请求体提供期望文件名、大小、业务类型等。
* 后端FileService生成一个新的File\_Object记录（或暂存，未真正插入，或者准备好插入），主要生成一个唯一object\_key。可用UUID，或者如果需要可读，可以包含文件名，但为安全避免特殊字符和重复，推荐UUID加扩展名。
* 调用MinIO SDK的 generatePresignedUrl(bucket, objectKey, expiry) 方法，得到一个临时可PUT的URL。【MinIO Java SDK有PresignedObjectUrl，或底层签名算法。】Expiry可以设短一点，比如 10分钟。
* 将此URL和所需HTTP方法(PUT)返回给前端，附带需要的HTTP header (如Content-Type)等信息。如果需要表单上传方式，也可返回policy (base64)等。但简单起见，用PUT直传即可。
* 前端获得URL后，通过fetch或Axios直传文件内容至该URL，MinIO验证签名允许上传。大文件也可以支持断点续传(那需分片签名，不在此细化)。
* 上传完成后，MinIO存储文件。前端应再通知应用服务器确认上传成功（可调用一个完成接口）——不过MinIO不直接告诉应用上传结果，所以稳妥是前端在PUT完得到HTTP 200响应后，再调用POST /files 提交 file metadata（或者在最初拿URL时后台就已经存了File\_Object记录为pending，现在需要确认）。

其实可以在upload-url接口中就把File\_Object insert为pending状态，并返回ID。前端上传后再调用 /files/{id}/complete 确认，这时后台更新File\_Object记录状态生效。如果前端放弃上传或失败未调用complete，可有定时清理pending。为简化，可以省略complete步骤，让前端拿到URL后直接用MinIO返回的HTTP状态决定。如果上传成功，就认为完成并UI提示完成；File\_Object记录可以先行插入状态=uploaded或者通过MinIO事件通知（MinIO支持Webhook通知上传事件）来确认。但搭Webhook较复杂。

此外，小文件可直接通过应用服务器POST上传文件表单，但那会走应用流量，除非部署有需要否则不推荐。

* **下载访问：** 默认文件私有，不开放匿名下载。因此提供**受控下载**：
* 浏览器访问文件需要有临时授权。可通过FileService提供GET /files/{id}/download-url，后台校验当前用户有权限访问该文件（比如匹配租户或业务关联），然后调用MinIO SDK生成一个GET的presigned URL（有效期例如 5分钟），返回给前端。前端再打开这个URL去下载文件内容。这使得文件内容还是由MinIO直传，应用仅签名授权。
* 或者为了不暴露URL，应用服务器也可直接读取MinIO对象流后通过自己的Response返回。但这样占用服务器带宽，不如presigned。签名URL暴露的也只是MinIO地址和有限时token，问题不大。
* MinIO默认服务在端口9000(或配置)，可以通过nginx反代或就直接用。前端拿到的URL可能是 http://minio:9000/mta-files/<object>，需确保网络可达。生产环境MinIO应配置域名和SSL。
* **文件引用完整性：** 在删除业务对象（如客户、计划节点）时，如果存在关联文件，应考虑一起删除或至少提示管理员清理。简单方式：在删除客户时，查询Customer\_File表有无记录，如有则同时删除MinIO上的对象和File\_Object记录（或设置File\_Object关联customerId ON DELETE CASCADE让数据库自动删除元数据，但MinIO内容需手动删除via API）。MinIO可以通过API删除object。最好由FileService提供方法deleteFile(fileId)执行：删除DB记录并调用MinIO删除物理文件。
* 另一侧，当用户要求删除一个文件（如误上传），UI提供删除按钮，调DELETE /files/{id}，后台检查关联是否允许删除（比如节点执行记录中的文件可能不允许无痕删除，可标记已删除或保留审计），再删除MinIO和DB。
* 如果File\_Object有biz\_id关联，最好由业务服务在删除关联对象时调用FileService清理，不要靠外键强删MinIO。
* **容量与备份：** MinIO存储的文件数据需要备份策略，尤其当使用本地FS存储MinIO数据。推荐挂载卷并定期备份（后面部署章节详述）。

### 审计与日志实现

**操作审计**是本平台重要非功能需求。实现方案：

* **AOP拦截**：使用Spring AOP在Service层或Controller层拦截标记了@Auditable注解的方法。对于CRUD方法，可以在执行后获取输入和输出对象，比较变化字段，然后写audit\_log。
* 例如：在CustomerService.updateCustomer(customerDto) 后，AOP切面拿到返回结果或通过customerDto.id从DB查询新值，与之前oldValue比较。Old可以在调用前从DB读一次。然后构造 Audit\_Log entry: user\_id从SecurityContext, entity\_type="Customer", entity\_id=customer.id, action="UPDATE", old\_data=json(oldValue), new\_data=json(newValue), detail=字符串描述如"名称 从 'A大学' 改为 'B大学'"。
* 对Delete, Create类似处理。Create无old数据，Delete无new数据。
* 一些敏感操作可以自定义detail，比如“用户 重置了 X 的密码”（不记录密码但记录操作）。
* **集中AuditService**：也可以不在各服务中直接写数据库，而是将审计事件封装成消息发给AuditService。如使用Spring Cloud Stream/Kafka，可发送一个AuditMessage（包含上述字段）。AuditService从队列消费并插入数据库。这种异步方式对性能友好，不阻塞主流程。但实现稍复杂，且本项目规模可接受同步写。可以做一个简单异步：各服务将Audit\_Log插入操作提交到一个线程池执行，避免阻塞用户请求过久。
* **日志结构**：Audit\_Log表的old\_data/new\_data JSON可以存储全部字段，也可以只存改变的。考虑查询需要，可以把主要标识字段列出来，如customer name在detail写明，或even duplicate in audit\_log columns if needed for filter. 但这样做需要升级表结构，当字段很多不实用。所以我们提供查询界面时，通过组合条件实现，例如按user筛选或按entity筛选即可，具体字段变动通过detail查看。
* **日志记录范围**：要求“无论变更什么，都记录细节”。所以几乎所有写操作都应记录，包括：配置修改（新建/修改自定义字段定义、模板创建修改）、执行流程的操作节点完成（包括上传文件这些）。很多细小操作如用户登录也应该记（一般登录记录在audit\_log里action=LOGIN entity\_type=User entity\_id=userid detail=ip等）。对于文件下载，视安全需要可记录。至少上传和删除文件要记录。
* **日志输出**：除了审计表，系统也会在应用日志文件中打印重要事件。为了不重复维护，两者区别：**审计日志**重在业务数据变更，**系统日志**涵盖调试、错误。使用Logback配置JSON格式输出的日志，日志字段中包含traceId、user等，可与审计记录的request\_id关联上（通过MDC）。
* 采用SLF4J MDC: 在每次请求开始(通过过滤器)生成一个UUID作为traceId，放入MDC，并获取当前用户username放入MDC。Logback配置%X{traceId} %X{user}. 这样每条日志行都会带这些上下文。
* 通过Spring Interceptor或Filter，将traceId也传递给响应header，方便客户端debug。
* 在任何异常或特殊事件时，日志级别Warn/Error打印堆栈，易排查。
* **ELK集成**：JSON日志可以被Filebeat收集至Logstash->ES。若要求，可定义索引名称如 bobmta-logs，字段包括timestamp, level, logger, message, traceId, user等等。OpenSearch类似。Audit日志如果也希望在ELK中分析，可以同步输出一份到日志，或者索性以日志形式存储而不在DB。但由于审计要求强一致性和结构化查询，用DB存是必要的。ELK分析审计数据可以通过配置Logstash JDBC input 或直接查询PG，不展开。
* **安全日志**：诸如登录失败、权限拒绝等，也应记录。Spring Security可配置事件监听，或手动在登录验证失败catch中记录audit: action=LOGIN\_FAIL, detail=用户名IP。

### 安全设计

安全部分涵盖认证、授权、防护和敏感数据处理：

* **认证 (Authentication)**：采用JWT方案：
* **登录**：用户提供用户名密码到 /auth/login （POST），AuthController验证：利用UserService查找用户，对比密码哈希（BCrypt matches）。如验证通过且用户激活，则生成JWT令牌。JWT通常包含claims：userId, username, roles, (tenantId)，签名使用服务器的私钥或对称密钥（配置在环境）。使用HS256对称算法简单，RS256非对称可选（便于将来单点登录公钥验证）。签发的JWT通过返回JSON或Cookie给前端。这里建议放入响应体，由前端存LocalStorage或Cookie（HttpOnly）。
* **JWT验证**：前端每次调用API在HTTP Header加 Authorization: Bearer <token>。后端网关Filter会解析该token验证签名和有效期，成功则将用户信息存入Spring Security Context，供后续权限判断。失败则返回401。JWT有效期可设比如2小时，支持refresh token机制：可以发一个长寿命的refresh token用于换新，但增加复杂度。简单可在过期后要求重新登录。
* **登出**：JWT是无状态，不存于服务器，所以“登出”仅是前端删除令牌或将其加入黑名单。在不考虑黑名单的情况下，只需前端清除本地token即可。为了保险，可以维护一个短期黑名单(token的jti)存服务器以防撤销（如用户密码修改应使旧token失效）。也可不做。
* **鉴权 (Authorization)**：RBAC模型通过Spring Security的@PreAuthorize或URL权限配置实现：
* Spring Security配置类里定义哪些URL需要认证，哪些允许匿名访问（如注册、激活接口）。对需要特定角色的接口，可用注解或在配置中 .antMatchers("/api/v1/admin/\*\*").hasRole("ADMIN") 等。菜单数据在后端返回时也应根据用户角色过滤。
* 方法级权限：对于敏感操作例如删除用户，可以加 @PreAuthorize("hasAuthority('USER\_DELETE')") 限制。权限点和角色对应关系存在DB，但Spring Security默认不会从DB实时获取，可以在登录时将用户权限加载进UserDetails里（比如查询role\_permission表获得permKey列表存入JWT或UserDetails GrantedAuthority）。JWT一般只存角色名以缩小大小，若需要精细权限控制，可以：
  + 将角色->权限逻辑写死在代码(不灵活)
  + 或每次鉴权时查询DB验证（性能差不推荐）
  + 或将权限也放入JWT claims（如果角色变化需要重新发token）。  
    简化起见，只基于角色名鉴权，大致够用。菜单等可以前端根据角色名控制显示。
* **输入校验**：对用户输入的数据，无论是API参数还是文件，都要校验和清理：
* **后端参数校验**：DTO类使用JSR 303注解，如 @NotNull, @Size, @Pattern 等，并在Controller方法上加 @Valid。Spring会自动返回400错误，对应的国际化错误消息可配置。
* **特殊字符处理**：对字符串防止SQL注入（MyBatis参数化已处理）、XSS存储攻击：对于存储到DB的文本，需要在输出给前端时做HTML转义或在前端使用安全渲染。如客户备注里有<script>标签，如果前端直接innerHTML显示会XSS。但React默认会转义危险字符，所以直接输出变量是安全的，除非使用 dangerouslySetInnerHTML。还需防范的是富文本字段(如邮件模板body如果允许HTML，需要存储时清理无效标签，在发送时控制)。
* **CSRF/CORS**：前后端分离，若前端域名不同需允许CORS。在Spring Security配置开启CORS with proper config (origins, methods, headers)，前端请求附带JWT即可。CSRF主要针对Cookies场景，因本项目JWT存在前端不是浏览器自动发送，不存在跨站自动利用的问题，可在Security中disable CSRF for JWT usage。
* **防重放**：JWT若泄露可被重放直到过期。Mitigation: 使用HTTPS避免窃听；Optionally, embed iat(issued at) and short TTL, or track an jti(token id) to detect reuse. 也可要求敏感操作提供一次性token(例如交易场景)，但维护复杂。当前场景下不特别实现防重放，依靠短有效期和HTTPS。
* **暴力破解防护**：登录接口可限制尝试次数：比如连续5次失败则锁账号或弹出验证码验证。实现：在AuthService中对登录失败计数（存在内存Map或在User表加字段fail\_count, last\_fail\_time），超限则将User.is\_locked置true或要求稍后再试。也可以对IP限制频率（用Bucket4j等限流库实现或在网关层Nginx配置limit\_req）。
* **数据权限**：按客户/组/标签/区域控制可见性。这涉及**ABAC**(基于属性的访问控制)。例如：某运维只能看负责的几个客户的数据。这可通过在User表加字段，如 assigned\_group\_ids (或一张User\_Group表) 来配置权限域。在查询客户列表API时，后端依据当前用户属性加过滤条件（如 user.assigned\_region only allowed）返回限定数据。实现上可能复杂，如以标签权限为例：给用户授予一个标签权限，那么只返回那些客户具有该标签的。需要设计一个结构Mapping user->tag or user->customer. 这里没有详细要求实现，只要提及数据权限控制点：
* 当用户不是管理员时，只能看到自己负责或所属组下的客户。
* 这样，在CustomerService查询时，可检查SecurityContext中用户角色，如果是ROLE\_USER，进一步筛选customer.group\_id in (user.allowed\_groups) 或 customer.id in (user.allowed\_customers)。
* allowed\_groups可通过角色来决定，比如“北海道团队”角色只能看region=北海道的客户，那么配置role权限里包含 region filter。这个需要硬编码或配置映射，不在此详细设计。
* 简单实现：User加字段 customer\_filter\_type（如 'ALL','GROUP','TAG') 以及对应值列表。登录时存入JWT。Service层解析并在SQL加where。
* 由于当前未具体要求实现哪种维度权限，我们只在此指出：系统设计预留这种扩展，未来可通过配置用户与客户、组、标签的关联，限制数据访问范围。例如在User界面可以将用户指派为某几个客户的负责人，则系统即可按此过滤。
* **敏感字段安全**：平台存储一些敏感数据如远程连接的密码/密钥等（Template\_Remote.credential）。必须确保：
* 数据库存储加密：可使用对称加密算法（如AES-256）将敏感信息加密后存储，密钥保存在应用配置不公开。这样DB泄露也难直接获取密码。实现可在TemplateService保存时调用一个EncryptionUtil.encrypt()，读取时解密。
* 使用者权限：只有特定权限的用户才能查看或使用这些信息。例如普通运维可能看不到明文密码，只能点击“连接”由系统使用密码。UI上可将密码字段mask。如果需要查看密码，要求提升权限或二次验证。
* 传输时也要加密：JWT可能包含用户信息但不会有密码。敏感数据如密码绝不发到前端，前端只获得临时可用的数据（比如Remote模板通过生成RDP文件包含密码，但可以在RDP文件内加密码吗？RDP文件可以包含，属风险，但如果电脑受控还好）。或者要求用户现用自己的凭据接入，但那复杂化。
* 远程连接凭据涉及客户机密，建议仅管理员可录入更新，运维人员执行时应用自动使用但不暴露。例如未来Agent实现可在服务端解密密码后建立通道。当前如果只是手工，用不着密码因为VPN之类人工输入。综合而言，**当前实现可以存但不显示**。
* **文件安全检查**:
* **文件类型限制**：允许上传的文件类型应有限制清单。例如允许图片（png/jpg）、pdf、txt、csv等常用格式，不允许可执行文件（exe, bat）以免存储恶意文件。具体实现：FileService根据文件扩展名或MIME类型判断，如果不在允许列表则拒绝上传（返回415 Unsupported type之类）。允许列表可配置为 environment变量，如 ALLOWED\_FILE\_TYPES=jpg,png,pdf,docx,xlsx,txt.
* **病毒扫描**: 上传的文件有可能含病毒。可集成杀毒软件扫描接口。常用方案：部署ClamAV服务，FileService在文件上传后调用ClamAV扫描该对象（MinIO可以通过文件网关或FileService先落地文件临时存一下再扫描再存MinIO）。比较复杂。当前设计可**预留接口**：FileService.upload完成后调用 VirusScanService.scan(objectKey)，目前没有实现只是打日志“假定通过”。以后有需要可启用实际扫描服务。
* **下载安全**：提供下载链接的时候，如扩展名敏感也没关系因为浏览器处理。关键是确保只有授权人能下载（通过签名URL有效期短控制）和文件内容未经篡改（MinIO自带Etag可以做校验，用不到，我们信任MinIO存储完整性）。
* **最小权限原则**：在配置各组件时要注意：
* 数据库用户权限：应用连接数据库的用户应只授予必要权限（CRUD自己库表即可，不用超级权限）。
* MinIO凭据：AccessKey/Secret控制整个Bucket，如果有多租户可以考虑MinIO多用户及策略限定访问自己前缀。但实现复杂度高，可先用单账户，注意保护Secret不泄露。
* 第三方集成的凭据（SMTP密码、Webhook密钥）都要安全存储（配置中心加密或者环境变量提供，别硬编码）。
* 日志中对敏感数据脱敏：如某些日志别打印密码、Token等。如果启用了SQL日志或调试日志，注意过滤。

## 6. API 设计（REST/JSON）

后端对外通过统一的RESTful风格API提供服务。基本约定：

* **路径规范：** 以 /api/v1/ 为统一前缀，版本号v1便于未来升级兼容。各资源使用复数命名，如 /customers, /plans. 子资源用嵌套，如 /customers/{id}/tags.
* **HTTP方法约定：** GET用于查询，POST用于创建和特殊动作，PUT用于整体更新（幂等），PATCH用于部分更新，DELETE用于删除操作。
* **请求和响应格式：** 全部采用JSON格式报文，编码UTF-8。客户端请求Body使用 Content-Type: application/json。文件上传除外使用multipart/form-data或预签URL方式。返回结果包含业务数据或错误信息。
* **统一返回结构：** 为便于前端处理，建议所有成功响应包装为统一格式，例如：
* { "code": 0, "message": "OK", "data": {...} }
* 其中 code=0 表示成功，非0表示各种错误类型（类似错误码）。data为具体返回数据（对象或列表或其他），message一般成功可为OK或留空，错误时包含简要描述。或者也可直接返回数据对象，用HTTP状态区分成功/错误。但统一格式利于前端统一拦截处理错误。这里采用**统一包装**方案。
* **错误码规范：** 约定错误码为数字，分段表示类型：如 400系参数错误，401未认证，403未授权，404未找到，409业务冲突，500服务器错误等。可以将HTTP状态和内部错误码对应，如code=4000表示参数校验不通过，4010表示未登录，4031表示权限不足等等。后端维护一张错误码字典（参见附录），同时返回时message可由后端根据Accept-Language给出当前语言的错误信息。比如用户登录失败，返回 {code: 1001, message: "ユーザー名またはパスワードが違います"} (日文)，中文环境则返回中文提示。这样前端可以直接显示message，无需二次翻译。
* **分页约定：** 列表查询支持分页参数，一般通过查询参数 ?page=1&size=20（第1页，每页20条）。返回时，在data中可包含分页信息，如：
* { "code":0, "data": { "list": [...], "total": 95, "page":1, "size":20 } }
* 也可以将分页信息置于HTTP响应header如 X-Total-Count，但前后端约定放body结构即可。
* **幂等性：** 除GET天然幂等外，对于POST创建等，需要防重复提交机制。可采用**Idempotency-Key**：前端在需要场景生成UUID作为请求头传递，服务器保存处理过的key在一定时间内（比如写入Redis或内存），若重复收到则直接返回相同结果或拒绝。典型用于支付、工单创建。平台可选实现。例如创建计划接口，前端若检测到重复提交可以重用key。此实现较复杂，当前可以写在文档中以备扩展，不强制实现。

以下按照领域列举主要API接口，以 **表格** 方式或列表说明接口路径、方法、功能、权限、请求和响应示例：

**客户与分组标签自定义字段 API**

* **获取客户列表**  
  **GET** /api/v1/customers  
  **功能**：查询客户列表，可选分页和过滤参数（group, tag, keyword）。  
  **权限**：登录用户 (ROLE\_USER以上)。  
  **请求示例**：GET /api/v1/customers?groupId=5&tagId=2&page=1&size=50&keyword=大学  
  **响应示例**：
* {  
   "code": 0,  
   "data": {  
   "list": [  
   { "id": 101, "code": "0100", "name": "北海道大学", "group": "大学", "region": "北海道", "tags": ["VPN", "重要"] },  
   { "id": 102, "code": "0120", "name": "北見工大", "group": "大学", "region": "北海道", "tags": ["アナログ"] }  
   ],  
   "page": 1,  
   "size": 50,  
   "total": 2  
   }  
  }
* 列表元素可简要带部分字段，具体需要哪些字段可根据前端需要调整（可能不需要全部详情以节省流量）。
* **获取客户详情**  
  **GET** /api/v1/customers/{customerId}  
  **功能**：获取单个客户的详细信息，包括自定义字段值、标签、远程连接信息等。  
  **权限**：ROLE\_USER 且对该客户有查看权限。  
  **响应示例**：
* {  
   "code": 0,  
   "data": {  
   "id": 101,  
   "code": "0100",  
   "name": "北海道大学",  
   "group": { "id": 5, "name": "国立大学" },  
   "region": "北海道",  
   "tags": [ { "id": 1, "name": "VPN", "color": "#1890ff" }, { "id": 3, "name": "重要", "color": "#f50" } ],  
   "fields": {  
   "接続形態": "VPN",  
   "接続ツール": "GlobalProtect",  
   "接続制限": null,  
   "特殊運用": null,  
   "顧客別情報参照": "要",  
   "特記事項": "GlobalProtectで接続, ...",  
   "接続先IPアドレス/電話": "133.50.8.10 / 0157-22-2704"  
   },  
   "last\_update": "2018-02-15",  
   "remote\_connections": [  
   { "type": "VPN", "tool": "GlobalProtect", "address": "133.50.8.10", "note": "V6バージョンアップに伴いDBサーバ変更" }  
   ]  
   }  
  }
* 其中fields列出所有自定义字段(含CSV中那些)，remote\_connections如一个客户有多个地址/方式可以罗列；或者直接在fields里表达亦可。  
  注：字段名此处直接用了日文，因为前端默认日文环境可直接显示，也可返回key和让前端翻译。视实现决定。
* **创建新客户**  
  **POST** /api/v1/customers  
  **功能**：创建一个客户档案。请求Body提供基本信息和可选自定义字段值。  
  **权限**：管理员 (ROLE\_ADMIN) 或有客户创建权限的用户。  
  **请求示例**：
* {  
   "code": "0130",  
   "name": "新潟大学",  
   "groupId": 5,  
   "region": "新潟県",  
   "fields": {  
   "接続形態": "VPN",  
   "接続ツール": "LAPLINK",  
   "顧客別情報参照": "不要"  
   },  
   "tags": [1, 5] // 标签ID列表  
  }
* **响应示例**：
* { "code": 0, "data": { "id": 120, "message": "Customer created" } }
* 返回新建客户ID等。注意tags和fields如果有未知字段键要报错或忽略。
* **更新客户信息**  
  **PUT** /api/v1/customers/{id}  
  **功能**：更新客户基本信息以及自定义字段、标签。PUT通常要求客户端发送全量信息（PATCH则可部分）。  
  **权限**：ROLE\_ADMIN 或有此客户编辑权限的用户。  
  **请求示例**：
* {  
   "name": "北海道大学",  
   "region": "北海道",  
   "fields": {  
   "接続形態": "VPN",  
   "特記事項": "2023年更新: 新VPN機器導入"  
   },  
   "tags": [1, 3]  
  }
* **响应**：
* { "code": 0, "message": "Customer updated successfully" }
* 如果更新包含自定义字段，后台需更新或插入customer\_custom\_field\_val表相应记录；标签更新则调整customer\_tag关联。
* **删除客户**  
  **DELETE** /api/v1/customers/{id}  
  **功能**：删除客户记录及其附属信息。通常应考虑逻辑删除或需要级联删除关联数据。这里提供物理删除接口。  
  **权限**：ROLE\_ADMIN。  
  **响应**：{ "code": 0, "message": "Deleted" }  
  若客户有关联计划等，不允许删除，应返回错误码409冲突。
* **获取客户分组列表**  
  **GET** /api/v1/customer-groups  
  **权限**：ROLE\_USER  
  **响应**：包含所有分组及父子关系，用于前端构建树。  
  **示例**：
* { "code":0, "data": [ { "id":5, "name":"国立大学", "parentId":null }, { "id":6, "name":"大学", "parentId":null } ] }
* **新建客户分组**  
  **POST** /api/v1/customer-groups （管理员权限）  
  请求：{ "name": "私立大学", "parentId": null }  
  响应：{ "code":0, "data": {"id":7} }
* **编辑客户分组**  
  **PUT** /api/v1/customer-groups/{id} （管理员）  
  请求：{ "name": "私立大学", "parentId": 6 }  
  响应：{ "code":0 }
* **删除客户分组**  
  **DELETE** /api/v1/customer-groups/{id} （管理员）
* 若该分组下有客户，需先处理（可阻止并返回错误409）。
* **获取标签定义列表**  
  **GET** /api/v1/tags  
  **权限**：ROLE\_USER  
  **响应**：{ "code":0, "data": [ { "id":1, "name":"VPN", "color":"#1890ff", "scope":"CUSTOMER" }, ... ] }
* **新建标签**  
  **POST** /api/v1/tags （管理员）  
  请求：{ "name": "重要", "color": "#f50", "icon": "StarFilled", "scope": "BOTH" }  
  响应：{ "code":0, "data": {"id": 9} }
* **更新标签**  
  **PUT** /api/v1/tags/{id} （管理员）  
  **删除标签**  
  **DELETE** /api/v1/tags/{id} （管理员，需考虑已有引用）
* **给客户打标签**  
  **POST** /api/v1/customers/{id}/tags  
  请求：{ "tagId": 3 }  
  响应：{ "code":0 }
* 此接口也可以批量如传数组。或通过更新customer时附带tags字段已实现，无需单独。
* **移除客户标签**  
  **DELETE** /api/v1/customers/{cid}/tags/{tid}  
  **权限**：ROLE\_ADMIN 或客户负责人
* **获取自定义字段定义列表**  
  **GET** /api/v1/custom-fields （管理员权限，因为普通用户可能不需要看到定义但无妨只读）  
  响应：{ "code":0, "data": [ { "id":11, "fieldKey":"connection\_type", "fieldName":"接続形態", "fieldType":"TEXT", "fieldGroup":"リモート接続情報", "required":false }, ... ] }
* **新增自定义字段定义**  
  **POST** /api/v1/custom-fields （管理员）  
  请求：{ "fieldName": "新項目", "fieldKey": "new\_item", "fieldType": "TEXT", "fieldGroup": "その他", "required": false }  
  响应：{ "code":0, "data": { "id": 15 } }
* 后端需确保fieldKey唯一，若未提供则可根据Name自动生成拼音或下划线key。
* **更新自定义字段定义**  
  **PUT** /api/v1/custom-fields/{id} （管理员）
* 允许修改名称、分组、是否必填等。不建议改fieldKey因影响已存数据。
* **删除自定义字段定义**  
  **DELETE** /api/v1/custom-fields/{id} （管理员）
* 如已有值，可能需要先处理值，可禁止删除或采用逻辑删除（is\_active=false）。

**计划与流程定义/执行 API**

* **获取计划列表**  
  **GET** /api/v1/plans  
  **功能**：按筛选条件查询计划列表。支持按客户、日期范围等过滤。  
  **参数**：customerId (选), fromDate, toDate, status 等。也可根据视图类型返回不同结构。如 calendar视图想要按天分组，可加参数 mode=calendar-month 等，或者客户端自行按日期组织。  
  **响应示例**（简单列表模式）：
* {  
   "code":0,  
   "data": {  
   "list": [  
   { "id":501, "title":"定期バックアップ", "customer":{"id":101,"name":"北海道大学"}, "startTime":"2025-10-01T10:00:00", "endTime":"2025-10-01T12:00:00", "status":"COMPLETED", "owner":{"id":1,"name":"admin"}, "progress":"5/5" },  
   { "id":502, "title":"システム更新", "customer":{"id":102,"name":"北見工大"}, "startTime":"2025-10-05T09:00:00", "endTime":"2025-10-05T18:00:00", "status":"SCHEDULED", "owner":{"id":2,"name":"山田"}, "progress":"設計中" }  
   ],  
   "total": 2  
   }  
  }
* 可根据不同需求调整字段，比如progress这里用字符串，可能也可以给百分比等。
* **获取某客户的计划列表**  
  **GET** /api/v1/customers/{cid}/plans  
  功能类似上，可作为简便过滤。
* **获取计划详情（含流程定义）**  
  **GET** /api/v1/plans/{planId}  
  **功能**：获取计划详细信息，包括嵌套的流程节点定义结构，和执行情况（如果有）。  
  **响应示例**：
* {  
   "code":0,  
   "data": {  
   "id": 502,  
   "title": "システム更新",  
   "customer": { "id":102, "name":"北見工大" },  
   "description": "10月定例Windowsアップデート",  
   "startTime": "2025-10-05T09:00:00",  
   "endTime": "2025-10-05T18:00:00",  
   "status": "IN\_PROGRESS",  
   "participants": [ { "id":2,"name":"山田" }, { "id":5,"name":"佐藤" } ],  
   "nodes": [  
   { "id":1001, "name":"事前通知メール送信", "order":1, "actionType":"EMAIL", "actionRef": { "id":21, "name":"通知メールテンプレ" }, "assignee": null, "expectedDuration":5,  
   "status": "DONE", "startTime":"2025-10-05T08:50:00", "endTime":"2025-10-05T08:55:00", "result":"メール送信済" },  
   { "id":1002, "name":"VPN接続", "order":2, "actionType":"REMOTE", "actionRef": { "id":30, "name":"VPN Template" }, "assignee": {"id":5,"name":"佐藤"}, "expectedDuration":10,  
   "status": "IN\_PROGRESS", "startTime":"2025-10-05T09:00:00" },  
   { "id":1003, "name":"サーバ再起動", "order":3, "actionType":"NONE", "assignee": {"id":5,"name":"佐藤"}, "expectedDuration":30,  
   "status": "PENDING" },  
   { "id":1004, "name":"結果報告", "order":4, "actionType":"IM", "actionRef": { "id":25, "name":"Slack通知" }, "assignee": null, "expectedDuration":5,  
   "status": "PENDING" }  
   ]  
   }  
  }
* 这里nodes数组按顺序列出节点，如果有子节点则可以嵌套children数组。或者序列化为平面列表由字段parentId表示父子关系。在JSON输出中为了清晰，可嵌套children结构。  
  每节点除了定义（name, actionType等）还有执行状态信息（status,start,end,result）。这些来自Node\_Instance，如果Plan未执行则无这些字段或status=PENDING。  
  actionRef包含模板简要信息，assignee给出姓名。
* **创建运维计划**  
  **POST** /api/v1/plans  
  **功能**：新建一个计划（进入设计状态）。请求中包含基本信息和流程模板(可选)。  
  **权限**：ROLE\_USER 或有创建权限的人都可，新计划创建者默认owner。  
  **请求示例**：
* {  
   "customerId": 101,  
   "title": "月次バックアップ",  
   "description": "DB月次バックアップ実施",  
   "startTime": "2025-10-31T22:00:00",  
   "endTime": "2025-11-01T02:00:00",  
   "participantIds": [1, 2],  
   "templatePlanId": null  
  }
* participantIds指定参与人用户id数组，若不提供默认当前用户为owner。templatePlanId若提供，可表示从现有计划复制流程模板。  
  **响应**：{ "code":0, "data": { "id": 510 } } 新计划ID。  
  后端逻辑：插入Plan记录（status=DESIGN），如果有templatePlanId则复制其process\_node结构为新plan的nodes。否则创建空流程或默认流程。
* **更新计划**  
  **PUT** /api/v1/plans/{id}  
  **功能**：修改计划信息（标题、时间、参与人等）。执行中的计划限制修改某些字段。  
  **权限**：计划owner或ADMIN  
  **请求**：例如更改时间或描述。  
  **响应**：成功空响应即可。
* **删除计划**  
  **DELETE** /api/v1/plans/{id}  
  **权限**：ADMIN或owner。若计划已发布或执行，不允许删除，可返回409。未发布可删除。
* **发布计划（切换执行模式）**  
  **POST** /api/v1/plans/{id}/publish  
  **功能**：将计划从设计模式切换为执行模式。可以理解为“启动计划”。  
  **权限**：计划owner或ADMIN。  
  **逻辑**：设置Plan.status=SCHEDULED或IN\_PROGRESS并创建Process\_Instance记录。若startTime是未来，则标记为SCHEDULED等待时间到自动IN\_PROGRESS；如果startTime<=now则直接IN\_PROGRESS并可以开始执行首节点。可以在publish接口中做判断：如果发布时间=当前即时开始，就返回Plan status=IN\_PROGRESS，否则=SCHEDULED。对于SCHEDULED的计划，可有后台调度在startTime时自动改状态为IN\_PROGRESS，也可由用户手动开始。  
  **响应**：{ "code":0 }  
  发布后，后端应冻结流程定义（不允许再改节点）。
* **取消计划**  
  **POST** /api/v1/plans/{id}/cancel  
  **功能**：取消一个未开始或执行中的计划。  
  **权限**：owner或ADMIN。  
  **逻辑**：将Plan.status=CANCELED，若已执行部分节点，做记录（可能日志注明谁取消）。不删除数据，可在列表依然看到状态已取消。
* **开始执行节点**  
  **POST** /api/v1/plans/{id}/nodes/{nodeId}/start  
  **功能**：标记指定节点开始执行。通常由前端在用户点击“开始”按钮时调用。  
  **权限**：参与该计划的用户（Plan.participant），且节点assignee匹配当前用户或当前用户是owner。  
  **逻辑**：创建对应Node\_Instance（或更新已有的status) start\_time=now, status->IN\_PROGRESS, assignee设当前用户如果原来为空。返回当前服务器时间或状态。  
  **响应示例**：{ "code":0, "data": { "status": "IN\_PROGRESS", "startTime": "2025-10-05T09:00:00" } }  
  如果节点已经开始或完成则返回错误码如409。
* **完成节点**  
  **POST** /api/v1/plans/{id}/nodes/{nodeId}/complete  
  **功能**：标记节点完成，并提交执行结果。  
  **请求**：
* {  
   "result": "成功（エラーなし）",  
   "log": "バックアップファイルをNASに保存済。",  
   "fileIds": [55, 56]  
  }
* result为简单结论，log为详细操作日志，fileIds为上传的相关文件（如日志文件或截图）ID数组。  
  **逻辑**：后台将对应 Node\_Instance 的end\_time=now, status->DONE, 保存result和log。关联文件则在Node\_File表批量插入(node\_instance\_id, file\_id)。接着检查：如果此节点有兄弟并属于某父节点，计算父节点子完成率。如果达到threshold，自动完成父节点（写父Node\_Instance.end\_time等），并跳过未完成子节点。若父节点完成且它本身有父则递归向上检查。然后返回新的流程状态，如下一个节点ID。  
  **响应示例**：
* {   
   "code":0,   
   "data": { "nextNodeId": 1003, "parentCompleted": true }   
  }
* 例如告诉前端父节点是否因为threshold完成，下一个执行节点是哪一个（也可以不返回，由前端自行刷新获取plan状态）。
* **节点添加日志**  
  **POST** /api/v1/plans/{id}/nodes/{nodeId}/log  
  **功能**：在节点执行过程中追加日志记录。  
  **请求**：{ "log": "実行中：50%完了" }  
  **逻辑**：插入或更新 Node\_Instance.log\_text (可以累加换行追加)。  
  **响应**：{ "code":0 }  
  （此接口可选，不一定需要，完全可以用complete接口一次性提交最终log即可。如果需要实时日志显示，用WebSocket更好。不过当前简化。）
* **节点上传附件**  
  **POST** /api/v1/plans/{id}/nodes/{nodeId}/files  
  **功能**：将已上传的文件（fileId列表）绑定到节点。  
  **请求**：{ "fileIds": [55] }  
  **逻辑**：在Node\_File表插入 records，将file\_object.biz\_type标记"NODE\_ATTACHMENT"并biz\_id=node\_instance\_id。也可以通过file\_object.biz\_id方案，则在此更新file\_object.biz\_id=node\_instance\_id。不过通常我们会先上传再调用此绑定，所以File\_Object一开始biz\_id可能为空，绑定时更新。  
  **响应**：{ "code":0 }
* 如果upload采用预签+complete模式，这步可能整合在complete，视实现可以省略单独接口。

**模板中心 API**

* **获取模板列表**  
  **GET** /api/v1/templates/{type}  
  **功能**：查询某类型的模板（email/im/link/remote），支持分页过滤。  
  **权限**：ROLE\_USER(一般运维只读模板), ROLE\_ADMIN(可管理模板)。  
  **响应**：如GET /api/v1/templates/email 返回Email模板列表。  
  **示例**：
* { "code":0, "data": { "list": [ { "id":21, "name":"通知メールテンプレ", "subject":"【{{customer\_name}}】メンテナンス通知", "enabled": true } ] } }
* **获取模板详情**  
  **GET** /api/v1/templates/{type}/{id}  
  返回模板详细内容（邮件内容、IM消息内容等）。
* **创建模板**  
  **POST** /api/v1/templates/{type} （管理员）  
  **请求示例**（email）：
* {  
   "name": "週報テンプレート",  
   "subject": "【{{week}}】週次報告",  
   "body": "今週の作業完了件数: {{count}}\n詳細は添付をご確認ください。",  
   "toList": "team@example.com,{{customer.email}}"  
  }
* **响应**：{ "code":0, "data": {"id": 22} }
* **更新模板**  
  **PUT** /api/v1/templates/{type}/{id} （管理员）
* **删除模板**  
  **DELETE** /api/v1/templates/{type}/{id} （管理员）
* **模板预览渲染**  
  **POST** /api/v1/templates/{type}/{id}/render  
  **功能**：提供给前端在编辑模板或调试时，使用一些样例数据渲染出实际内容。  
  **请求**：{ "variables": { "customer\_name": "北見工大", "count": 5, "week": "2025-W40" } }  
  **响应**：{ "code":0, "data": { "subject": "【2025-W40】週次報告", "body": "今週の作業完了件数: 5\n詳細は添付をご確認ください。" } }
* 这个API需要TemplateService实现简单的占位符替换。占位符格式可以统一用 {{var}} 或类似。

**文件上传/下载 API**

* **申请文件上传**  
  **POST** /api/v1/files/upload-url  
  **功能**：获取用于直接上传文件到MinIO的预签名URL和所需信息。  
  **请求**：{ "fileName": "backup\_log.txt", "contentType": "text/plain", "size": 20480, "bizType": "NODE\_ATTACHMENT" }  
  **响应**：
* {  
   "code": 0,  
   "data": {  
   "fileId": 55,  
   "uploadUrl": "http://minio:9000/mta-files/1/node\_attachment/550e8400-e29b-41d4-a716-446655440000.txt?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=...",  
   "method": "PUT",  
   "headers": { "Content-Type": "text/plain" }  
   }  
  }
* 前端据此用HTTP PUT上传文件内容到uploadUrl。fileId生成但此时文件数据未到达，记录可标记临时状态。  
  （如果使用表单直传S3风格，会返回policy、signature等信息，现用PUT简单模式。）
* **确认文件上传完成** (可选)  
  **POST** /api/v1/files/{fileId}/complete  
  **功能**：通知服务器文件已上传成功，让服务器验证并更新File\_Object状态。  
  **逻辑**：可以调用MinIO HEAD接口检查object存在大小匹配，然后更新File\_Object表，比如去除临时状态、填写size。  
  **响应**：{ "code":0 }  
  如果不做complete，则File\_Object记录可以在upload-url接口就填好全部信息假设上传成功，但万一失败就不知。可以通过定期检查MinIO清理孤儿。
* **文件下载**  
  **GET** /api/v1/files/{fileId}/download  
  **功能**：获取文件的访问（预签名）链接，或以附件形式返回文件内容。  
  **实现选项1**：返回 JSON 包含一个短效的URL:
* { "code":0, "data": { "url": "https://minio.example.com/mta-files/...&X-Amz-Signature=...", "expiresIn": 300 } }
* 前端可直接window.open该URL下载。  
  **实现选项2**：直接响应文件流，通过 Content-Disposition: attachment; filename=...。这样路径始终/api/files/id，不暴露MinIO地址。需要FileService用MinIO客户端取对象流后写Response。对于较大文件会消耗应用带宽。不首选。  
  无论哪种实现，都需校验当前用户有权限获取此file（比如文件关联的customer是其可见customer）。
* **删除文件**  
  **DELETE** /api/v1/files/{fileId}  
  **功能**：删除存储的文件。  
  **权限**：有管理权限或上传者。  
  **逻辑**：FileService从MinIO删除object，DB删除File\_Object记录及关联（Node\_File等）。如审计需要，可将记录转存到一个history表或日志中，或者将File\_Object.is\_deleted标记而不物理删以防丢证据，但文件可以删或改名保留占位。

**国际化词条 API**

* **获取词条**  
  **GET** /api/v1/i18n?lang={lang}&since={timestamp}  
  **功能**：获取指定语言的词条集合。如提供since，则仅返回修改时间>since的增量词条。可以使用HTTP If-Modified-Since 头代替参数。  
  **响应**：
* {  
   "code":0,  
   "data": {  
   "lang": "ja",  
   "messages": {  
   "customer.list.title": "顧客一覧",  
   "customer.list.search": "検索..."  
   // ...  
   },  
   "lastModified": 1695724800000 // 时间戳  
   }  
  }
* 前端拿到messages对象存入intl库。对于没有更新304，服务端可返回HTTP 304无body。  
  *注：如果前后端都从数据库拉资源，这接口必要性视实现而定。如果前端打包内置基础文案，只用这个接口获取更新或者某些服务端消息翻译。这里提供以满足需求。*
* **更新词条** (如提供后台管理词条时)  
  **POST/PUT** /api/v1/i18n （管理员）  
  **请求**：{ "lang":"zh", "code":"customer.list.title", "text": "客户列表" } 或批量messages数组。  
  **响应**：{ "code":0 }  
  更新后服务端应清理缓存以便立即对下次请求生效。

**用户/角色/权限 API**

* **用户注册**  
  **POST** /api/v1/auth/register  
  **功能**：用户自助注册申请。  
  **请求**：{ "username":"yamada", "email":"yamada@example.com", "password":"abc123??" }  
  **响应**：{ "code":0, "message":"登録完了。確認メールを送信しました。" } (提示已发送激活邮件)  
  **逻辑**：检查email在预留表，如果允许则创建User记录 is\_active=false, generate token, send email.
* **激活账号**  
  **GET** /api/v1/auth/activate?token=...  
  **功能**：用户点击激活链接由前端调用。  
  **响应**：重定向到前端激活结果页面或返回JSON表示成功/失败。  
  **逻辑**：UserService验证token（可JWT或UUID映射用户），然后设置用户is\_active=true, 清除token。
* **登录**  
  **POST** /api/v1/auth/login  
  **请求**：{ "username":"yamada", "password":"abc123??" }  
  **响应**：
* {  
   "code": 0,  
   "data": {  
   "token": "<jwt\_token\_string>",  
   "user": { "id":2, "username":"yamada", "name":"山田", "roles":["USER"], "preferredLang":"ja" }  
   }  
  }
* token是JWT，用于后续请求。返回基本用户信息用于前端显示及权限控制。  
  登录失败：返回 code如1001, message "ユーザー名またはパスワードが違います"。
* **注销**  
  **POST** /api/v1/auth/logout  
  **功能**：前端请求注销，服务端可不做实际操作，只是记录审计日志和建议前端删除token。  
  **响应**：{ "code":0 }
* **请求密码重置**  
  **POST** /api/v1/auth/forgot-password  
  **请求**：{ "email": "user@example.com" }  
  **响应**：始终返回成功以防暴露邮箱是否存在：{ "code":0, "message":"パスワードリセット手順をメール送信しました。" }  
  **逻辑**：如果email对应的用户存在，生成reset token保存，发送邮件。
* **重置密码**  
  **POST** /api/v1/auth/reset-password  
  **请求**：{ "token": "...", "newPassword":"NewPass123" }  
  **响应**：{ "code":0, "message":"パスワードがリセットされました。" }  
  **逻辑**：验证token有效后，更新用户password\_hash，令用户激活（如果之前未激活则激活），并使token失效。
* **获取当前用户信息**  
  **GET** /api/v1/auth/me  
  **功能**：根据JWT返回用户信息(roles等)，用于前端初始化权限。  
  **响应**：同login返回的user对象。
* **查询用户列表**  
  **GET** /api/v1/users （管理员）  
  支持过滤角色、状态。响应包括id, username, email, roles, is\_active, last\_login等。
* **创建用户**  
  **POST** /api/v1/users （管理员）  
  **请求**：{ "username":"sato", "email":"sato@foo.com", "roleIds":[2], "sendActivation": true }  
  如果sendActivation=true，系统发送激活邮件，否则可以返回一个初始密码。  
  **响应**：{ "code":0, "data": {"id": 10, "tempPassword": null} }  
  tempPassword如有返回说明需要告知用户。
* **更新用户**  
  **PUT** /api/v1/users/{id} （管理员或用户自己部分修改）  
  **请求**：管理员可改角色：{ "roles": ["ADMIN"] }， 或锁定：{ "isLocked": true }  
  用户自己可改姓名，preferredLang等。可能分不同接口，如 /users/me 修改个人资料。
* **删除用户**  
  **DELETE** /api/v1/users/{id} （管理员）
* 通常不物理删而是锁定，酌情。
* **角色与权限**
* **GET** /api/v1/roles （管理员） 列出所有角色以及包含权限
* **POST** /api/v1/roles （管理员） 新建自定义角色
* **PUT** /api/v1/roles/{id} 修改角色名称或权限分配
* **DELETE** /api/v1/roles/{id} 删除角色（若无用户使用）
* **GET** /api/v1/permissions （管理员） 列出所有权限key及描述，方便分配使用
* **分配用户角色**  
  **POST** /api/v1/users/{id}/roles （管理员）  
  请求：{ "roleIds": [1,2] }  
  响应：{ "code":0 }

**审计日志查询 API**

* **获取审计日志**  
  **GET** /api/v1/audit-logs （管理员或审计员角色）  
  **功能**：提供过滤参数：userId, entityType, entityId, action, date range 等。支持分页。  
  **响应**：
* {  
   "code":0,  
   "data": {  
   "list": [  
   { "id":10001, "timestamp":"2025-09-26T15:00:00", "userName":"admin", "entityType":"Customer", "entityId":"101", "action":"UPDATE", "detail":"顧客「北見工大」の特記事項を更新", "ipAddress":"10.0.0.5" }  
   ],  
   "total": 1  
   }  
  }
* 列表中只给概要，如果想看具体改动，可以在detail里写或提供**获取审计详情**接口。
* **获取审计详情**  
  **GET** /api/v1/audit-logs/{id}  
  **响应**：
* {  
   "code":0,  
   "data": {  
   "id":10001,  
   "timestamp":"2025-09-26T15:00:00",  
   "userId":1,  
   "userName":"admin",  
   "entityType":"Customer",  
   "entityId":"101",  
   "action":"UPDATE",  
   "oldData": { "特記事項": "旧情報..." },  
   "newData": { "特記事項": "新情報..." },  
   "ipAddress":"10.0.0.5",  
   "userAgent":"Mozilla/5.0 ...",  
   "requestId":"abc-123-xyz"  
   }  
  }
* oldData/newData根据实现可能整个对象或部分字段。

以上列举了主要API。接下来给出**端到端时序图**示例，演示一些关键流程。

**时序图1：执行模式下节点完成远程连接动作并记录审计与文件**

sequenceDiagram  
 participant U as 運維ユーザー（前端）  
 participant F as 前端应用  
 participant GW as API网关  
 participant PS as Plan服务  
 participant FS as File服务  
 participant AS as Audit服务  
 Note over U,F: ユーザーが現在のノードを実行完了し<br/>結果を記録（遠程接続を行いスクリーンショットを取得）  
 U->>F: 点击"上传截图"<br/>选择文件 screenshot.png  
 F->>FS: POST /api/v1/files/upload-url (文件名等)  
 FS-->>F: 返回 fileId=77 和 uploadURL  
 F->>MinIO: HTTP PUT 文件内容 至 uploadURL  
 MinIO-->>F: 返回 200 OK（上传成功）  
 F->>PS: POST /api/v1/plans/502/nodes/1002/files<br/>{ fileIds: [77] }  
 PS->>FS: 验证 fileId=77 权限并绑定到节点1002  
 FS->>PS: 绑定成功（记录 Node\_File）  
 PS->>AS: 记录审计日志 ("文件上传", user=U, file=77)  
 PS-->>F: 返回 {code:0}  
 U->>F: 点击"完成节点"  
 F->>PS: POST /api/v1/plans/502/nodes/1002/complete<br/>{ result:"成功", log:"VPN接続OK", fileIds:[77] }  
 PS->>PS: 验证节点状态（可完成）<br/>更新 Node\_Instance(1002) status=DONE<br/>保存result, log  
 PS->>PS: 计算父节点阈値/进度（如有）  
 PS->>DB: 保存节点执行数据和引用文件  
 PS->>AS: 记录审计日志 ("完成节点", user=U, plan=502, node=1002, oldStatus="IN\_PROGRESS", newStatus="DONE")  
 PS-->>F: 返回 {code:0, data:{ nextNodeId:1003 }}  
 F->>U: UI更新：节点1002标记完成<br/>节点1003高亮为当前

*解释：* 用户上传截图文件通过File服务获得URL并PUT至MinIO，然后调用Plan服务的files接口绑定文件到节点。Audit日志记录了文件上传操作。之后用户标记节点完成，Plan服务更新状态并记录结果日志，同时写审计日志记录节点状态变化。前端得到响应后更新流程UI。

**时序图2：计划发布并导出ICS供Outlook订阅**

sequenceDiagram  
 participant A as 管理员用户  
 participant F2 as 前端应用  
 participant GW2 as API网关  
 participant PS2 as Plan服务  
 participant CS2 as 客户服务  
 participant Cal as Outlook客户端  
 A->>F2: 在计划页面点击"发布计划"  
 F2->>GW2: POST /api/v1/plans/510/publish  
 GW2->>PS2: 路由请求 (带JWT)  
 PS2->>DB: 更新 plan(510).status = SCHEDULED  
 PS2->>DB: 创建 process\_instance 记录  
 PS2->>Scheduler: 注册定时任务(510) 在 startTime 触发开始  
 PS2->>Audit: 记录审计 ("发布计划510", by 管理员)  
 PS2-->>F2: 返回 {code:0, data:{ status:"SCHEDULED"} }  
 Note over F2: UI显示计划状态为"已发布"<br/>提供 ICS订阅URL 链接  
 A->>Outlook: 添加Internet日历 订阅<br/>URL = https://mta.example.com/api/v1/calendar/tenant/1.ics  
 Outlook->>GW2: GET /api/v1/calendar/tenant/1.ics  
 GW2->>PS2: 路由请求  
 PS2->>DB: 查询 tenant=1 所有未来计划  
 PS2->>PS2: 将计划列表转为ICS格式字符串  
 PS2-->>Outlook: 返回 ICS文件内容 (text/calendar)  
 Note over Outlook: Outlook日历导入计划510事件<br/>之后每隔数小时会重新GET订阅URL[1]  
 ... 时间到达计划startTime ...  
 Scheduler-->>PS2: 触发计划510状态变更  
 PS2->>DB: 更新 plan(510).status = IN\_PROGRESS, process\_instance.start\_time = now  
 PS2->>Audit: 记录审计 ("计划开始510")  
 Note over Outlook: 下一次订阅刷新时,<br/>ICS里该事件可能更新状态/描述

*解释：* 管理员发布计划，Plan服务创建定时任务到点启动。Outlook订阅通过Calendar URL获取ICS，Plan服务动态生成包含所有计划的ICS。注意Outlook刷新不及时[[1]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=Note%3A%C2%A0When%20you%20subscribe%20to%20a,take%20more%20than%2024%20hours)。计划开始时服务更新状态并审计。

**时序图3：“新增自定义字段即时在客户详情生效”**

sequenceDiagram  
 participant Admin as 管理员用户  
 participant F3 as 前端应用  
 participant GW3 as API网关  
 participant CustSvc as Customer服务  
 Admin->>F3: 打开"自定义字段管理"界面  
 F3->>CustSvc: GET /api/v1/custom-fields (加载已有字段定义)  
 CustSvc-->>F3: 返回现有字段列表JSON  
 Admin->>F3: 填写新增字段表单 (字段名: "担当者", 类型: 文本)  
 F3->>GW3: POST /api/v1/custom-fields<br/>{ fieldName:"担当者", fieldKey:"tantou\_sha", fieldType:"TEXT", fieldGroup:"基本情報" }  
 GW3->>CustSvc: 转发请求  
 CustSvc->>DB: 插入 customer\_custom\_field\_def(id=16,...)  
 CustSvc->>Audit: 写审计日志("新增自定义字段: 担当者(id=16)")  
 CustSvc-->>F3: 返回 {code:0, data:{id:16}}  
 Note over F3: 前端提示成功，并将新字段加入本地字段定义列表  
 Admin->>F3: 打开 某客户详情 页面  
 F3->>GW3: GET /api/v1/customers/101  
 GW3->>CustSvc: 请求客户详情  
 CustSvc->>DB: 查询customer(101)基本信息和关联值  
 CustSvc->>DB: 查询customer\_custom\_field\_val for customer 101 (包括刚定义字段16, 如无值返回null)  
 CustSvc-->>F3: 返回客户详情 JSON (fields 包含 "担当者": null)  
 Note over F3: 前端渲染详情: 出现新字段"担当者"一栏，值为空  
 Admin->>F3: 点击编辑客户  
 F3: (提供"担当者"字段可填写)  
 Admin->>F3: 输入担当者名字提交  
 F3->>GW3: PUT /api/v1/customers/101 { fields: { "担当者": "佐藤" } }  
 GW3->>CustSvc: 更新请求  
 CustSvc->>DB: 检查field\_def表找到key=tantou\_sha对应id=16  
 CustSvc->>DB: upsert custom\_field\_val (customer\_id=101, field\_def\_id=16, value="佐藤")  
 CustSvc->>Audit: 记录审计("客户101: 字段 担当者 新值 '佐藤'")  
 CustSvc-->>F3: 返回成功  
 F3->>Admin: 页面刷新显示 担当者 = 佐藤

*解释：* 管理员新增自定义字段后立即在客户详情查询时就能看到这个字段（值为空）。这个即时生效依赖前端从后端获取字段定义时已经包含新字段（前端可以在新增字段成功后更新本地字段定义或重新拉取）。客户编辑时新字段也可填写，提交update接口后，后台写val表并审计。整个流程用户无须刷新全站即应用了新字段。

以上时序图验证了设计中一些关键点（文件上传绑定、ICS订阅、动态字段更新）。实际实现中可能合并或调整顺序，但总体遵循这些流程。

## 7. 前端组件与状态流设计

本章聚焦前端实现细节，特别是UI组件结构和状态管理策略，以确保不同模块交互流畅，状态一致。

**客户一览页面组件划分：**

CustomerPage  
├─ CustomerSidebar (左侧客户列表)  
│ ├─ GroupFilter (分组/地区筛选控件)  
│ ├─ SearchBar (搜索框 & 拼音索引)  
│ ├─ CustomerList (列表项容器)  
│ │ ├─ CustomerListItem[] (客户列表项组件)  
│ └─ TagFilterPanel (标签过滤多选)  
└─ CustomerDetail (右侧详情区)  
 ├─ DetailHeader (标题、标签、编辑按钮等)  
 ├─ BasicInfoPanel (基本信息板块)  
 ├─ RemoteInfoPanel (远程连接信息板块)  
 ├─ ... (其他板块)  
 ├─ AttachmentsPanel (附件列表, 如客户相关文件)  
 └─ DetailFooter (操作按钮区：编辑/删除)

* **CustomerSidebar** 维护客户列表数据和当前过滤条件。通过Redux或内部状态存储 customers 列表，activeGroup, searchKeyword, selectedTags 等。当筛选条件变化或搜索时，Sidebar会过滤列表项。选中某客户后，将选中ID通知全局（Redux）或父组件 CustomerPage 状态，触发加载详情。
* **CustomerListItem** 渲染单个客户简要信息。点击后派发 selectCustomer 动作 (payload: customerId) 更新全局当前选中客户。
* **CustomerDetail** 订阅全局当前选中客户ID (如 Redux state.customer.selectedId)。初次加载或ID变化时 dispatch fetchCustomerDetail(id) 异步action（或 React Query）。Detail组件在等待时显示Skeleton加载骨架。当数据加载成功，将数据通过子组件props传入各Panel组件。
* Panels如 BasicInfoPanel、RemoteInfoPanel 负责展示各自分类字段。可以动态生成字段列表：BasicInfoPanel遍历字段定义里field\_group="基本情報"的字段，对于每字段创建 <FieldItem label=... value=...> 元素。RemoteInfoPanel同理。
* 标签显示在DetailHeader中利用 Tag 组件 (AntD Tag or custom) 列表，颜色样式通过标签定义 color 来设置背景色。支持可点（管理员可以点标签触发取消或跳转筛选）。TagFilterPanel 在Sidebar用checkbox列表，让用户筛选客户。
* **状态管理**：Customer数据可以放入 Redux的 customer slice。state结构：
* state.customer = {  
   list: [],   
   detailMap: { [id]: { ...customerDetail } },  
   selectedId: null,  
   filter: { group:null, tagIds:[], keyword:"" }  
  }
* Actions: fetchCustomers, fetchCustomerDetail, selectCustomer, setFilter, etc. Using Redux Toolkit createAsyncThunk for data fetching.
* Filter changes trigger filtering logic. Could do filtering on frontend for small data or request server with query params for large data.这里可由server筛选分页，所以 filter change => dispatch fetchCustomers with new params.
* 编辑客户：DetailFooter点击编辑，弹出 CustomerEditModal (使用Form初始化当前详情数据，包括自定义字段：可根据字段定义生成Form.Item列表)。保存时调用 API PUT /customers/id，然后 dispatch updateCustomerSuccess action 更新store detailMap和list (if needed).
* 删除客户：弹确认，调用DELETE后成功则 dispatch removeCustomer(id) 更新store.list移除，selectedId也需处理（可选中下一个或none）。UI返回列表空白或提示选中。
* **性能优化**：客户列表可能很多，使用 AntD List with virtual scroll or custom VirtualList to handle large number. Also splitting CustomerPage with memo or useMemo to avoid re-render detail when list changes and vice versa.

**运维计划页面组件划分：**

PlanPage  
├─ ViewToggleBar (切换按客户/日历视图)  
├─ PlanByCustomerView (按客户视图组件)  
│ ├─ CustomerSelector (客户下拉选择器)  
│ └─ PlanList (该客户的计划列表 - table or cards)  
├─ CalendarView (日历视图组件)  
│ ├─ CalendarToolbar (切换日/周/月/年, 今天按钮, 前后导航)  
│ └─ CalendarGrid (根据 mode 渲染相应日历)  
│ ├─ DayView (24h timeline)  
│ ├─ WeekView (7-day grid)  
│ ├─ MonthView (month grid)  
│ └─ YearView (year summary)  
└─ PlanDetailModal (计划详细对话框/抽屉)  
 ├─ PlanHeader (标题, 客户名, 时间, 状态, 参与人, 标签, 操作按钮:发布/取消等)  
 ├─ ProcessDesignPanel (流程设计模式主容器)  
 │ ├─ NodeEditorList (节点列表可编辑)  
 │ └─ NodeEditForm (选择某节点时右侧/下方编辑属性表单)  
 ├─ ProcessExecutionPanel (流程执行模式主容器)  
 │ ├─ NodeProgressBar / StepsVisualizer (图形化展示流程进度)  
 │ ├─ CurrentNodeCard (当前节点详情卡片, 含操作按钮)  
 │ ├─ NodeLogPanel (日志记录区域)  
 │ └─ AttachmentsPanel (附件上传查看区域)  
 └─ PlanFooter (关闭/保存等)

* PlanPage根据当前视图模式显示PlanByCustomerView或CalendarView。
* **PlanByCustomerView**：包含客户选择（下拉或左侧列表）和PlanList。PlanList可以用AntD Table或List，每行显示计划基本信息和状态。点击某行打开PlanDetailModal查看详情流程。
* **CalendarView**：考虑用库 FullCalendar for React 实现日历UI，或者AntD Calendar + custom date cell. FullCalendar支持日/周/月切换，以及 events data feed。Given complexity, likely use FullCalendar. If not, we manually create:
* DayView: vertical timeline (0-24h) with events placed at their times (like a schedule).
* WeekView: 7 columns (Mon-Sun) each column with events boxes at relative times. Possibly more complex to implement manually.
* Possibly integrate an existing calendar component to save time.
* In either case, Plan data is needed in date structure. We can fetch all plans of current month or so. Ideally, PlanService can provide all or paginated events. For year view, might just show counts per month or highlight days with events.
* Clicking an event in calendar triggers opening PlanDetailModal (with plan id).
* **PlanDetailModal** (or could be a separate page route /plans/:id, but likely a modal for quick access):
* If plan.status = DESIGN (未发布), show ProcessDesignPanel with editable node list.
  + NodeEditorList: list of NodeEditorItem components each representing a node (with fields: name, assignee, duration, etc) and possibly drag handles to reorder. Could utilize antd List + DragDropContext for reordering.
  + NodeEditForm: Alternatively, simpler is inline editing in the Node list (like each item can toggle to show fields).
  + Provide "Add Node" button (adds at end or allow insert in middle).
  + Provide indent/outdent for sub-process (if implementing sub-level).
  + In design mode, "Save" button saves node changes (persist via API calls for each change or one big save).
  + Possibly a "Publish" button to finalize.
* If plan.status != DESIGN (published or completed), show ProcessExecutionPanel:
  + NodeProgressBar/StepsVisualizer:
  + Could be horizontal steps if few steps (AntD Steps component) or custom timeline if many.
  + Alternatively a vertical list but with styling differences: current node highlighted, completed nodes with check mark, etc. Possibly use AntD Timeline for vertical with icons for done, current, etc.
  + However, requirement suggests "current node enlarged, previous/next smaller". This might be achieved with CSS in a vertical list: e.g. current node card has scale(1.1) transform, others scale(0.9) and partly transparent. Could implement with some custom CSS if list is not too large.
  + CurrentNodeCard: separate highlight card at top of Execution panel, showing node name, expected vs actual time, actions: "Start"/"Complete" or if done, show done icon. If not started, show "Start" (enabled if user is assignee or owner).
  + If node is in progress and type=REMOTE, show a "Open Connection" button. If type=EMAIL or IM and maybe system triggers auto sending, maybe show "Sent" or allow "Resend".
  + If node requires user action, they will use Start/Complete. Some nodes might auto complete if just informational? But likely user will mark complete anyway.
  + NodeLogPanel: shows any logs for current node (and possibly allows input if node is in progress or done? We can allow adding notes even after done if need).
  + Could be a simple textarea or a list of log entries with timestamp. Possibly a nice to have is to show a timeline of logs for node.
  + For initial, just a textarea input for adding log, and a "Add" button. Or only allow one log text finalized at completion, which is simpler. But they mentioned "record logs or screenshots during execution", implying multiple logs can be taken.
  + AttachmentsPanel: allow uploading files relevant to current node. Use Upload component (maybe AntD Upload which can directly integrate with our upload API via custom request). Show list of uploaded files with preview or download link. Possibly allow uploading in progress or after completed as evidence.
  + If using AntD Upload, can provide custom action (our upload-url -> then auto upload to that).
  + Or use a simple file input and call the file API endpoints as orchestrated in sequence diagram earlier.
  + Completed nodes in NodeProgress list could be clickable for viewing what was done (maybe showing result/log). We might show those details on clicking a node in progress list or maybe in a side or bottom panel.
  + Execution controls:
  + "Start Plan" (for scheduled plan that reached time, maybe manual).
  + "Cancel Plan" if need to abort (calls cancel API).
  + Possibly "Return to design" if we allow toggling back (not likely).
* PlanHeader: shows high-level info and has action buttons:
  + If status=DESIGN: "Publish" (calls publish API), "Delete Plan".
  + If status=IN\_PROGRESS: maybe "Mark Completed" (if need to forcibly close), "Cancel Plan".
  + If status=COMPLETED: "Export Report" or similar (maybe not in initial scope).
  + Tag management: maybe allow adding tags to plan (if permitted, through an editable tag list like in customer detail).
* If plan is COMPLETED, ExecutionPanel would show everything as read-only and logs/attachments as final.
* **状态管理**:
* We might keep plan list separate in state: state.plan.list (for calendar or by-customer listing). Possibly normalized by id.
* state.plan.current (for currently open plan detail).
* Or use React Query: one query for list (with filters), one query for detail (by planId). Could be effective, especially calendar needing to fetch events by range.
* Considering complexity, using Redux for plan might be fine but React Query could simplify caching multiple plan details.
* Example Redux:
* state.plan = {  
   list: [],   
   filter: { mode:"customer", customerId:null, view:"month", date:someDate },   
   calendarEvents: [] (if storing separately for calendar view),  
   currentPlan: { ...fullDetail, nodes:[] },  
   currentStatus: "idle"/"loading"  
  }
* Perhaps simpler: load plan detail on modal open with a thunk, store it in currentPlan. When modal closes, maybe clear it or keep last loaded.
* When switching to different plan from calendar or list, we fetch new detail.
* Node updates (start/complete) should update currentPlan's nodes state. Use events or optimistic update.
* WebSocket updates for plan progress to other users? Could be useful if multi-user on same plan. Possibly skip in initial but note it.
* Multi-user concurrency: if two people open same plan, one completes a node, the other should see update. Without push, they'd have stale view. Possibly add a polling or user refresh manually.
* **键盘/快捷操作**:
* Could allow using arrow keys to navigate nodes in Execution mode (e.g. up/down to see prev/next node details).
* Perhaps keyboard "n" for next node after completing one, etc.
* If desired, implement via keydown listeners on modal.
* **表单/校验**:
* Use AntD Form with form validation rules for things like required fields (customer name cannot be blank, user email format, etc).
* On save, catch errors in form and highlight fields. If error from server (like duplicate code), show as message or on field if possible (like antd can use help on Form.Item).
* For date/time, use DatePicker/TimePicker ensure format, possibly with timezone awareness (set default zone).
* For number fields (expectedDuration), use InputNumber with min=0 etc.
* Many fields like group, assignee will use Select drop-down with options loaded from server (customer group list, user list).
* We should load dictionary data: e.g. user list for assignee field, maybe directly call user API or at least for selection limit to participants list in plan context.
* Field type options: if new custom field type=DATE, front-end should show DatePicker for it, if =BOOL show Checkbox.
* We should maintain a map type->component or a generic FieldRenderer.
* **错误处理**:
* Global error: e.g. session expired (401) -> intercept and redirect to login with message.
* API error codes: can map code to friendly message via i18n or use message from server directly (since it should be already localized).
* Provide a notification or message pop-up for success/failure of operations. For instance, after saving, show "保存しました".
* Use AntD message or notification for quick feedback.
* **加载状态**:
* Use AntD Spin or Skeleton. Eg:
  + Customer list skeleton maybe not needed if quick.
  + Customer detail skeleton: show placeholder lines for each field group.
  + Plan detail modal open should show spinner if data loading.
  + Calendar might show a spinner overlay when loading events.
* Use React Suspense if using React Query to auto handle loading state.

By partitioning components and managing state carefully, the front-end ensures decoupling: e.g. editing a customer in modal will update global state so list and detail reflect changes.

## 8. 多语言与资源管理

如前述，本系统采用**后端集中**和**前端缓存**相结合的国际化资源方案。下面补充说明实现细节和运维：

* **词条命名规范**：遵循 模块.页面.元素 格式。例如：
* 登录页：auth.login.title = "ログイン"
* 错误信息：error.invalid\_password = "パスワード形式が無効です"
* 按钮文本：button.save = "保存"
* 通用词汇尽量复用，如 "保存" 不要每个页面定义。 命名避免使用硬编码ID，直接用英文语义有助开发识别。但注意不要直接用英文短语作为key，因为需要唯一性且易读性结合。 一种做法：英文翻译可以与key相同如果key已足够表意，比如 key="customer.list" text\_en="Customer List", 这样不需要双维护key和英文。
* **前后端资源映射**：建议**后端只存日文和中文**这些UI显示文本即可，不用存英文（除非以后要）。因为前端默认日语，可以将日语词条打包在前端bundle以防止界面闪烁。 实际上，如果前端等待API返回词条再渲染，会有短暂白屏或占位。优化：
* **Embed default language**: 把日语词条编译进前端，页面首次渲染使用本地日文文案（这些文案可能由开发时配置的messages\_ja.json提供）。
* 前端同时发请求获取远端词条更新（maybe including Chinese or updated Japanese).
* 当API返回后，如果发现有更新（根据ETag或版本），则合并/替换本地词条，然后force refresh UI or just update for future interactions (like new dynamic content). But if changes are minor, user might not notice. Possibly just use new ones for next page or on demand.

Alternatively, if allow a short delay, show a loading spinner for a moment and then show UI when messages loaded. But likely better to embed default language to avoid UX delay.

* **缓存策略**：
* 浏览器LocalStorage can store last fetched translations (by language). Next time app loads, it can use that as initial data (if not bundle). Or just rely on bundling default.
* ETag/If-None-Match: On initial load, front-end can provide If-None-Match with previously saved version (e.g. localStorage.i18nVersion). If unchanged, server returns 304. If changed, returns new data + new ETag (like a hash or version number).
* Alternatively, since we plan to implement incremental update (with since timestamp), the app can store lastModified and request only new keys. But that's more complicated, simpler is to always fetch full if changed.
* **增量灰度发布**：Imagine we add a new feature with new keys:
* Back-end get updated with new keys in DB, but if front-end code tries to show that text and did not fetch new keys, it would display fallback (maybe the key itself or a default).
* If front-end calls i18n API at startup, it will get the new keys anyway (provided we updated version).
* So as long as we update keys in DB slightly before or at same time as deploying front-end, user will fetch them. Even if not, front-end could show key or placeholder. Acceptable in short term.
* For safe side, new features keys should be added to DB and considered as part of release.
* **新语言接入**： Suppose adding English:
* Add locale 'en' in i18n\_message table (maybe copy from either Japanese or get translation).
* Front-end add an option in language switch dropdown.
* Possibly provide language detection (like if user's preferredLang (from user profile) is zh, we call i18n?lang=zh at start).
* Confirm number/date formats: If using libraries, might need to set locale for them (like antd date picker if needed).
* **词条样例**：已在前面给JSON对照例。此处强调一下存储形式： If using DB:
* i18n\_message: | id | lang | code | text | |----|------|----------------------|---------------------| | 1 | ja | customer.list.title | 顧客一覧 | | 2 | zh | customer.list.title | 客户列表 | ...
* Possibly maintain the resource files as fallback (like messages\_ja.properties) in code, to seed the DB initially.
* **词条发布流程**：
* Developer defines new keys and uses them in front-end code (calls intl.formatMessage({id: 'new.feature.key'})).
* Developer adds these keys with translations to DB (or a translation sheet, then DB).
* Deploy new front-end and ensure new keys present in back-end DB (via migration or config).
* Users get new front-end code, which on load calls i18n API, gets updated dictionary including 'new.feature.key'.
* UI properly displays new text. If DB not updated, they'd see 'new.feature.key' (if fallback prints key).

If there's a process where translator adds translations via a UI, you could: - Provide an admin UI to add a new key (maybe for minor text adjustments). - Or use external system like Phrase or localize and import. But likely out of scope here.

## 9. 部署与运维

系统提供**Docker Compose**文件用于快速部署所有组件。以下是参考的 docker-compose.yml 内容：

version: '3.8'  
services:  
 # 前端服务 - 使用NGINX托管静态文件  
 web:  
 image: nginx:latest  
 container\_name: bobmta\_web  
 ports:  
 - "8080:80" # 将前端通过8080对外  
 volumes:  
 - ./frontend/build:/usr/share/nginx/html:ro # 假设前端打包输出放在frontend/build目录  
 - ./deploy/nginx.conf:/etc/nginx/conf.d/default.conf:ro # 自定义nginx配置  
 depends\_on:  
 - api  
 restart: unless-stopped  
  
 # 后端API服务 (Java Spring Boot)  
 api:  
 image: bobmta/api:1.0.0  
 container\_name: bobmta\_api  
 ports:  
 - "8081:8081" # 应用内部端口  
 environment:  
 - SPRING\_DATASOURCE\_URL=jdbc:postgresql://db:5432/mta  
 - SPRING\_DATASOURCE\_USERNAME=mtauser  
 - SPRING\_DATASOURCE\_PASSWORD=mtapass  
 - SPRING\_JPA\_HIBERNATE\_DDL\_AUTO=flyway # 或 none 如果flyway单独运行  
 - JWT\_SECRET\_KEY=${JWT\_SECRET\_KEY}  
 - MINIO\_ENDPOINT=http://minio:9000  
 - MINIO\_ACCESS\_KEY=minioadmin  
 - MINIO\_SECRET\_KEY=minioadmin  
 - MINIO\_BUCKET=mta-files  
 - SMTP\_HOST=smtp.example.com  
 - SMTP\_PORT=587  
 - SMTP\_USER=noreply@example.com  
 - SMTP\_PASS=yourpassword  
 - TZ=Asia/Tokyo  
 depends\_on:  
 - db  
 - minio  
 restart: unless-stopped  
  
 # PostgreSQL 数据库  
 db:  
 image: postgres:15-alpine  
 container\_name: bobmta\_db  
 environment:  
 - POSTGRES\_USER=mtauser  
 - POSTGRES\_PASSWORD=mtapass  
 - POSTGRES\_DB=mta  
 - TZ=Asia/Tokyo  
 volumes:  
 - ./data/db:/var/lib/postgresql/data # 数据卷  
 ports:  
 - "5432:5432"  
 restart: unless-stopped  
  
 # MinIO 对象存储  
 minio:  
 image: minio/minio:latest  
 container\_name: bobmta\_minio  
 command: server /data --console-address ":9001"  
 environment:  
 - MINIO\_ROOT\_USER=minioadmin  
 - MINIO\_ROOT\_PASSWORD=minioadmin  
 volumes:  
 - ./data/minio:/data # 对象存储数据卷  
 ports:  
 - "9000:9000" # MinIO API端口  
 - "9001:9001" # MinIO 控制台GUI端口  
 restart: unless-stopped  
  
 # MinIO 控制台（可选，用于UI管理）  
 minio-console:  
 image: minio/mc:latest  
 container\_name: bobmta\_minio\_console  
 depends\_on:  
 - minio  
 # MinIO Console can be accessed via minio:9001 if mapped above, so separate service may not be needed  
  
 # Nginx 反向代理（可选，用于统一域名和SSL）  
 # 若需要通过同一域名对外提供web和api，可使用nginx将 /api/\* 转发至api容器  
 nginx:  
 image: nginx:latest  
 container\_name: bobmta\_nginx  
 ports:  
 - "80:80"  
 - "443:443"  
 volumes:  
 - ./deploy/nginx\_reverse.conf:/etc/nginx/nginx.conf:ro  
 - ./deploy/certs:/etc/nginx/certs:ro # SSL证书  
 depends\_on:  
 - web  
 - api  
 restart: unless-stopped

**说明：**

* web 服务：使用官方nginx镜像，将前端构建好的静态文件挂载。外部端口8080映射nginx的80。nginx.conf可配置将请求的路径直接映射静态文件，并且如果有history routing的SPA，需要配置 try\_files fallback 到 index.html。我们这里也把 /api 转发给后端，也可以通过单独nginx服务做。上面的compose已提供一个nginx服务做反代，如果使用它，则web仅供nginx内部访问，不映射8080外部。
* api 服务：假设已经构建好Spring Boot应用的Docker镜像 bobmta/api:1.0.0。它暴露8081端口（内部）。Compose把8081未映射到宿主，只被nginx或compose网络访问。我们通过depends\_on确保先启动db和minio。环境变量里配置数据库连接，MinIO地址，以及其他配置如JWT密钥等。JWT\_SECRET\_KEY应在 .env 文件或compose里的 ${JWT\_SECRET\_KEY} 来自环境，保证安全。
* db：Postgres数据库，使用数据卷保数据持久。POSTGRES\_USER/PASS/DB在此配置，和api服务的环境对应。注意生产应更强密码。TimeZone设为Tokyo以便Timestamp无偏移地按JST（可选，通常UTC更好，但为了时区直观选择这里JST）。
* minio：MinIO对象存储服务。Root credentials用默认minioadmin/minioadmin，这在生产必须改强随机值，通过环境变量覆盖。MinIO data挂载本地目录 ./data/minio。端口9000和9001分别映射，9001可以访问MinIO自带Web管理页面（http://localhost:9001，用上述账号密码登陆）。
* minio-console：这里用了minio/mc (client)镜像，但MinIO console is actually built in at 9001. So this might not be needed. Could remove or use for initial bucket creation by a command. Actually, the minio service command with --console-address ":9001" already runs the console. So minio-console is not necessary. We might skip it in final version.
* nginx (reverse proxy): If we want a single domain e.g. app.example.com serving both UI and API with nice URLs and HTTPS, we deploy this:
* It listens on 80 and 443. It would be configured to:
  + Serve static files (could either directly mount or proxy to web service).
  + Proxy /api/ path to the api service.
  + Handle SSL termination using certs in ./deploy/certs (assuming we place fullchain.pem and key).
* This requires writing an nginx.conf. (We included a placeholder path in volumes.)
* If not using this, an alternative is to let front-end calls http://localhost:8081 for API and avoid proxy, but that might cause CORS. So probably better to use the integrated approach with one domain.

For simplicity, one might skip external Nginx and just use web:8080 and api:8081 with CORS config. But production likely uses Nginx or similar to serve on standard ports and enable TLS.

**环境变量与初始账号：** .env file (not shown) can store sensitive defaults like JWT secret, and Compose references it. For initial admin user, we can: - E.g. if user table empty on startup, code can create admin/admin or email etc, printing password to log or requiring to change. - Or take from env INIT\_ADMIN\_PASSWORD. - It's important to not have default cred in code, so maybe require admin to run a DB script to create user or supply in env.

**卷与网络：** Compose by default puts all in one network, so services can refer by name. We used db, minio etc as host in config (like jdbc:postgresql://db:5432/mta). depends\_on ensures startup order (though app might need to wait for db ready, better to implement retry or healthcheck). We can include healthcheck for db:

healthcheck:  
 test: ["CMD", "pg\_isready", "-U", "mtauser"]  
 interval: 10s  
 timeout: 5s  
 retries: 5

This could be good for dependency but not strictly needed if app does its own attempt.

**健康检查：** - For api, if it has Actuator health endpoint at /actuator/health, we can add healthcheck in compose:

healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:8081/actuator/health"]  
 interval: 30s  
 timeout: 5s  
 retries: 3

So that Compose can detect and auto-restart if unhealthy.

**生产注意事项：** - **数据备份**：Mount volumes for db and minio to ensure data persistence outside containers (as we did). Then regularly backup ./data/db (or do a pg\_dump via cron or admin). - Actually, direct backing up Postgres data dir isn't ideal for hot backups; better to use pg\_dump or use volume snapshots. But maybe a nightly pg\_dump to file could be easier. Could include a backup container or instruct to do externally. - **MinIO credentials**: those defaults are unsafe. Should be changed via environment variables to unique values per deployment. Also consider restricting console or requiring strong pass. - **HTTPS**: For real deployment, one should configure nginx with SSL using a certificate (from Let's Encrypt etc.). The provided path ./deploy/certs indicates where to put them. Nginx config can also redirect 80->443. Without nginx service, user might rely on some external LB/ingress for HTTPS. - **日志**: We might want to mount a volume for logs if we log to file. But currently we plan JSON to stdout, which Docker can capture. If needed, one can do:

volumes:  
 - ./logs/api:/app/logs

and configure logback to write to /app/logs. - **滚动升级**: Using Compose, one could do docker-compose pull && docker-compose up -d to update images. If wanting zero downtime, might need multiple instances behind a load balancer (K8s scenario). In Compose alone, update means replacing container, short downtime unless external LB splitting between old and new separate composes, which is advanced. For small use it's fine to have small maintenance window.

**数据库迁移**: - We used environment SPRING\_JPA\_HIBERNATE\_DDL\_AUTO=flyway as a placeholder. Actually, one should integrate Flyway or Liquibase. E.g. include Flyway dependency which auto runs migrations at startup pointing to classpath:db/migration scripts. - Or run a separate migration container using Flyway CLI or a custom image:

flyway:  
 image: flyway/flyway:latest  
 command: -url=jdbc:postgresql://db:5432/mta -user=mtauser -password=mtapass migrate  
 volumes:  
 - ./db/migrations:/flyway/sql  
 depends\_on:  
 - db

That would run migrations then exit. - Alternatively, use Liquibase similarly. - It's crucial to manage schema changes for production, so included initial DDL as part of migrations.

**监控与日志**: - Could spin up a Prometheus container to scrape metrics from the API (if Actuator Prometheus enabled at /actuator/prometheus). - Grafana container to visualize metrics. - For logs, an ELK stack (Elasticsearch, Logstash, Kibana) or OpenSearch stack. Possibly heavy for small deployment, but mention: - We can include filebeat to send logs to a central system, or directly run a container for opensearch. - Or at least ensure our container logs can be accessed (by docker logs or aggregator). - Not necessary to include in initial compose, but plan optionally:

elk:  
 ...

We'll not detail due to scope.

**Docker Compose 文件** (provided above) is ready to run after adjusting environment values, building images (especially the api and web if needed). - .env.example would list needed variables (like JWT\_SECRET\_KEY, SMTP credentials, etc) for easy configuration. - Ensure file permissions appropriate (if on Linux, the user running Postgres inside container might need proper ownership of ./data/db, usually default user id 70 or 999? Typically the docker-entrypoint takes care by running as root then dropping to postgres user). - Provide a default admin seed: - Could utilize an env ADMIN\_EMAIL and ADMIN\_PASSWORD, and the app on startup checks if no users then create one. - We mention it in documentation, such as set ADMIN\_DEFAULT\_PASSWORD in env and the app uses it on first-run.

**运行应用**: - Developer or deployer runs docker-compose up -d. - Then uses MinIO console to create the bucket "mta-files" if the app doesn't auto create (MinIO allows auto bucket creation via command, or app might try to create bucket on startup via SDK if not exist). - Confirm API connecting to DB (if flyway runs, initial tables created). - Possibly pre-load with initial data (like default roles and permission, could either be done via data.sql or manual). - The front-end could then be accessed at http://localhost (if using nginx on 80) or http://localhost:8080 (if directly web). - Provide initial admin login to client. If not auto-created, instruct to insert admin user in DB.

**Volume backup**: - The ./data/db holds PG data; a backup procedure might be to exec into the db container and run pg\_dump. - The ./data/minio holds objects; one might rely on replicated storage or backup by copying. - Encouraging to use cloud storage might not apply if on-prem. So likely manual backups.

**日志挂载**: - If needed, we mount logs. But often in container environment, logs to stdout are aggregated by platform. - For debugging, might map a local log.

**滚动升级**: - If using Compose, not trivial to do zero-downtime because we have single instance. For real scenario, one could replicate instances behind LB. But if not, mention steps: 1. Bring up new version of containers maybe with different name/ports, then switch LB, then bring old down (complex outside compose). 2. Or just accept short downtime with docker-compose down && up. - In small internal systems, short maintenance might be acceptable, but good to mention how to minimize: - E.g. run DB migration first (ensuring backward compat if possible), then update API, then front-end, such that JWT tokens remain valid, etc.

## 10. 安全与合规

本节总结安全措施及确保系统合规性：

* **账号与密码策略：**
* 用户密码策略应该在注册或重置时强制执行。规则如：至少8字符，包含大小写字母和数字，推荐包含特殊字符。后台在设置密码时可用正则检查，弱密码拒绝。
* 密码存储使用强哈希（BCrypt等），并可能配置安全参数如work factor（BCrypt默认10，考虑性能可设12）。
* 支持用户修改密码功能：要求提供旧密码验证，以防被冒用。
* 账户多次登录失败锁定：如同一账户连续5次错误输入密码，则锁定账号 maybe 30分钟或需管理员解锁。也可结合验证码以防暴力破解。
* 初始默认密码（如admin）应在首次登录后强制修改：可以在登录成功但检测到密码=默认时，要求其更改再继续。
* 禁止密码明文出现在日志、审计，密码找回通过重置而非告知原密码。
* **邮件激活机制：**
* 新注册或由管理员创建的用户账号，需要通过邮件验证激活确保邮箱有效性和用户身份。激活链接应该包含一次性token（例如JWT或随机UUID），该token与用户对应且设置过期时间（如24小时）。
* 邮件内容应友好并不包含密码。激活完成后通知用户即可登录。
* 如果用户长时间未激活，可以由管理员重新发送激活邮件或删除账户。
* 注册接口不直接激活用户，有效防止机器人批量注册无效账号。
* **防暴力破解：**
* 除了账户锁定策略，还应对IP进行速率限制。可以采用如**Sliding Window算法**限制同一IP在一定时间内的登录尝试次数。例如每IP每分钟最多10次登录请求，超出则返回429或错误。
* 可使用中间件如 Nginx limit\_req 或应用层记录IP计数。
* 也可以在多次失败后对该IP要求输入验证码才能继续尝试登录。验证码服务可使用前端图形验证码或发送邮件验证码等。因实现复杂，此功能可列入拓展。
* **密码重置链路：**
* 当用户请求重置密码时，生成一个随机且单用途的token（长度足够，如32字符随机或JWT）存数据库，并发邮件。邮件中提供重置页面链接，包含token参数。
* Token应设置短有效期（比如30分钟），且使用一次即失效。
* 用户点击链接，到达前端Reset Password页面，输入新密码。提交后后台验证token，更新密码（使用强hash存储）。然后使该token失效（从DB删除或标记已用）。
* 若token无效或过期，提示重新申请。
* 重置完成后，可以发通知邮件告诉用户“您的密码已更改，如果非本人操作请联系管理员”，以防账户被盗修改密码而用户不知情。
* **审计与留痕：**
* 正如前述Audit Log设计，系统对**所有重要操作**都记录审计日志。这包括：用户管理（新增、删除、修改用户角色）、权限变更、客户信息修改（字段变更具体内容）、计划创建修改发布、流程步骤执行、模板修改、登录登出等。
* 审计日志确保包含**操作人**、**时间**、**目标对象**以及**改动详情**。通过前后值对比，[[5]](https://community-archive.progress.com/forums/00019/21255.html#:~:text=archive,new%20values%20is%20fully%20available) 用户可以清楚看到改了什么[[6]](https://www.sqlservercentral.com/forums/topic/audit-trail-logging-specific-changes-on-a-column-by-column-basis#:~:text=Part%20of%20the%20purpose%20of,so%20that%20they%20can)。例如“Admin在2025-09-26将客户101的特記事項从'旧内容'修改为'新内容'”。
* 审计日志应无法被一般用户篡改或删除。只有具备审计管理权限的管理员可以查看。删除审计日志（如归档老数据）应经过严格流程，最好逻辑删除或转存冷备，不允许任意删。
* 日志保存期限根据安全要求，可以设定如保存至少一年。数据库审计表需要定期归档迁移以防过大。
* 如有合规需要，可以将审计日志通过合规系统（如SIEM）收集分析，检查敏感操作。
* **敏感数据加密与访问控制：**
* **存储加密：** 对存储在数据库中的敏感信息（如远程连接密码、VPN密钥等），进行加密存储。可使用AES对称加密，密钥保存在服务器环境变量或密钥管理服务。这样即使数据库泄露，那些字段也是加密的。
  + 实施：Template\_Remote.credential 字段在插入前执行 encrypt(明文) 得到密文保存，取出时 decrypt(密文) 返回明文在内存使用。
  + 如果无法提供密钥管理，可将密钥硬编码不妥，因此建议通过配置文件提供密钥，生产环境慎重保管。
* **传输加密：** 所有网络传输使用HTTPS，特别登录、文件传输必须HTTPS以防窃听。我们的compose可以配置nginx使用TLS。内部容器通信也建议在安全网络，但因为在同主机内Docker网络相对安全，可以不额外加密（有需要可以使用TLS通信PG, MinIO也支持HTTPS配置）。
* **最小化暴露：** 前端对于敏感字段不展示明文。如远程连接密码，即使后端有，也不在UI明文显示。若一定要显示，需有额外验证（如管理员点击“显示密码”需要输入自己的密码再次确认）。
* **权限控制**：限制只有特定角色可以访问敏感数据。例如只有管理员角色的用户能创建/修改远程连接模板（因为含密码）。普通运维即使可以使用，也不应看到密码值。
* **日志脱敏：** 日志中避免输出机密值。比如当记录模板变更，不要记录原始密码，可记录字段变更但值用掩码表示（**\***\*）。同理，用户密码永不出现在日志。
* 数据库备份也要保护，备份文件应该加密存储，防止泄露。
* **文件安全**：
* **文件类型检查**：前端上传组件和后端FileService都应校验文件扩展名和MIME类型是否允许。如我们的允许列表ex: images, pdf, docx, xlsx, txt, csv等。不允许 .exe, .js, .sh 等可能危险的可执行或脚本文件，以防有人上传恶意程序。
* **文件扫描**：对上传文件可集成杀毒扫描。ClamAV开源杀毒可用。流程：用户上传文件->保存为暂存->调用ClamAV scan->若发现病毒，删除文件和记录并告警, 返回错误给用户。实现上在FileService收到上传完成后调用ClamAV容器(scan CLI)扫描 ./data/minio/对应文件。ClamAV库返回结果用于判断。虽然这种攻击机率小(内部运维人员上传病毒), 但考虑安全性还是值得。可以在文档中列此功能为选配，让运维可配置启用。
* **下载控制**：默认要求用户登录并有权限才能下载文件（通过获取签名URL）。避免直接公开文件链接。如果需要对外分享文件，可为该文件生成一个特定期限的公开URL（MinIO presign done)交给外方，而不直接开放整个bucket。
* **目录遍历**：由于MinIO通过object key管理，不存在web目录概念，但要防止设计上的目录泄露。如我们objectKey中使用 tenantId 等，不要可被轻易猜测。如用了UUID足够。万一有人知道bucket和一个objectKey，没有签名也取不到内容(权限不足), 所以还好。
* **文件名**：保留原始文件名用于下载时Content-Disposition，但存储用UUID避免恶意文件名造成问题。比如文件名不要用于路径直接，否则可能../之类理论没影响MinIO, 但最好sanitize文件名用于下载header。
* **防止XSS via 文件**: 如果允许上传HTML/SVG等文本文件, 日后下载时用户双击可能在浏览器打开而执行恶意脚本。通常下载的文件在用户环境打开不算我们应用负责。但是为了安全，可限制不允许html等。或者将Content-Type强制"application/octet-stream"让浏览器下载而非渲染。另外MinIO存object with its content-type, we should ensure correct type.
* **病毒感染**：上传文件可能被站点其他用户下载，如果带病毒影响他们。因此扫描必要。
* **备份**：文件存储应备份，与DB数据一起，不然客户文档丢失也是安全事故。
* **大文件**：限制单文件大小(配置Max). Default maybe 100MB or what suits environment to avoid storage exhaust or denial of service by large files. Enforce at front (upload component file size limit) and back (upload-url can include content-length-limit in S3 policy or check in finalize).
* **权限最小化**：
* 赋予用户和角色的权限遵循最小够用原则。默认用户角色只可查看和执行，不可改关键配置。只有管理员能管理用户、字段、模板等。
* 数据级最小权限：如之前所述，可将用户只关联其负责客户，这也是最小权限的一种形式。
* 服务组件方面，数据库用户只对mta库有DDL/DML权限，不授予管理模板或其他库。MinIO AccessKey也是限定在自己的bucket范围(不过MinIO root能管所有, 可创建 sub-user with policy).
* 操作系统层面，如果部署在服务器，应采用least privilege accounts运行容器，不以root权限运行应用容器(our Java app likely runs as root in container by default, better create a user in Dockerfile to run it). MinIO and Postgres images handle internally but we trust official images.
* 容器之间网络: Compose by default allows all internal comm, one might implement network segmentation if needed (like separate networks for db and others). But for moderate security, ensure no unnecessary ports exposed to host. Our example only exposes needed ones (we might not need to expose 5432 to host in production, except if DB admin needed remote connect. If not, keep it internal).
* Routine updates: ensure components kept updated (like security patches for Postgres, etc).
* **合规审查**:
* 如果此平台用于处理个人信息，需要遵守相关隐私法规。当前用户信息有限主要用户名/邮箱, 客户信息里可能有联系人电话? For now, likely just institutional info. Still, if personal data present, ensure encryption if needed and allow deletion if requested (like GDPR).
* 合规日志保存至少X年, role-based access for audit logs as we have done.
* 提供服务级协议(SLA) compliance: ensure uptimes etc. Achieving SLA might require monitoring and quick recovery which we've addressed with backup and replication strategies (RPO/RTO target).
* If any external compliance (ISO27001 etc), many of above measures align (access control, audit, data encryption).
* If the maintenance platform deals with something like healthcare or sensitive categories, might need additional encryption or segmentation. But likely not in this scenario (looks like university IT maintenance context from CSV data).
* Summaries: Each user action traceable, data protected at rest and in transit, principle of least privilege enforced.

## 11. 测试与验收

为确保系统质量，需制定全面的测试方案和验收标准，包括功能、性能、安全等方面。

**测试范围与类型：**

* **单元测试**：对后端各Service和Util方法编写单元测试用例，使用JUnit/Mockito等，覆盖核心业务逻辑，如：
* CustomerService增删改查方法对输入输出正确性测试；
* PlanService发布计划、阈值判断逻辑测试（模拟不同子节点完成比例是否正确标记父节点）；
* TemplateService渲染模板变量替换测试；
* Security相关，如密码加密验证流程、权限校验helper方法等。
* 每个DAO Mapper的SQL脚本可在H2数据库上做简单验证（MyBatis的话可通过SpringBootTest加载）。
* **集成测试**：启动应用加载完整上下文，用模拟HTTP请求测试各API接口功能是否按预期。可使用Spring Boot的MockMvc或者RestTemplate/WebTestClient:
* 模拟用户登录获取token，访问需要认证的接口验证权限拦截；
* 测试跨模块流程，比如创建客户后能否被计划引用；
* 文件上传接口可以用模拟字节流测试presign和MinIO存储 (本地MinIO容器或mock S3 if available)；
* 测试审计日志AOP拦截：执行一次修改操作后查询audit\_log表校验记录存在且字段正确。
* **端到端测试(E2E)**：针对前端界面流程，使用自动化工具（如Cypress、Selenium）模拟真实用户操作：
* 打开系统登录页 -> 输入凭证 -> 验证页面跳转客户列表；
* 在客户列表执行搜索、分组展开收起等，看结果是否符合预期；
* 点击客户打开详情，验证详情信息分块显示正确，对比已知测试数据；
* 新增一个自定义字段，通过后台UI操作完成，然后在客户详情是否出现该字段。
* 设计/执行一个运维计划流程：创建计划->添加几个节点->发布->在执行界面点击完成节点->最后验证流程状态变成完成且ICS输出正确。
* 注册流程E2E：打开注册页输入信息->检查邮箱收到激活邮件(可使用测试SMTP server抓邮件) -> 点击激活 -> 登录。 这些UI操作用Cypress可以无头模拟，也可手工验收。
* **UI测试**：对前端组件的交互状态进行验证：
* 组件单元测试：用Jest/React Testing Library测试关键组件渲染逻辑，如CustomerList根据store数据正确渲染过滤结果；PlanDetailModal根据不同状态渲染不同UI元素(Design vs Execution)。
* Snapshot测试保存UI输出避免意外改动。
* **性能测试**：使用JMeter、Gatling等模拟并发负载，重点测试：
* **并发用户数**：模拟50-100用户同时登录、浏览客户列表、打开计划。看系统响应时间（目标95%请求<1s）和资源占用。
* **大数据量**：在测试数据库中填充比如1000个客户，每个客户有多字段、计划几十个节点，看页面加载和接口查询性能是否下降可接受（比如客户列表请求返回< 300ms, 计划列表<500ms）。
* **文件上传**：模拟多个用户并行上传文件各50MB，检测MinIO和应用的处理稳定性。检查上传完成率和耗时。
* **长时间运行**：进行 soak test 让系统跑24小时、不断执行一些操作，看有无内存泄漏或性能衰退。
* **安全测试**：
* **漏洞扫描**：使用工具(ZAP/Burp)扫描Web应用常见漏洞，如SQL注入、XSS、CSRF。验证输入过滤是否生效。比如尝试在输入框输入 <script>alert(1)</script> 看前后端如何处理，不应弹框。
* **权限绕过**：尝试用普通用户的token调用管理员接口，应返回403。尝试不带token访问需认证接口返回401。
* **会话管理**：JWT是否过期正确刷新/拒绝。注销后旧token是否还能用(如果实现黑名单需测)。
* **文件安全**：上传一个不允许类型文件(如 .exe)，应该被拒。上传一个含EICAR测试病毒的文件，看是否被拦截(如果杀毒集成)。
* **大输入**：测试异常，比如上传超大文件(超配置)，应该有错误提示。文本字段超长输入应在后端fail(比如customer.name限长)。
* **恢复测试**：
* 模拟数据库故障：停掉db容器，看应用日志是否健壮处理(比如连接重试)。恢复db后应用能否自动恢复服务(需要pool重连)。
* 备份恢复：用备份文件恢复数据库，启动系统检查数据完整性。MinIO同理。

**验收标准** (关键功能验收点)：

下面以**测试用例表**形式列出主要验收用例及期望结果：

1. **客户列表分组/搜索/排序**  
   **步骤**：登录普通用户 -> 在客户一览左侧选择分组“国立大学” -> 列表应只显示该分组客户 -> 输入搜索关键词 "北海道" -> 列表应筛选名称含"北海道"的客户 -> 点击假名索引 "は" -> 列表跳转到以“は”开头客户。切换无分组排序 -> 列表按客户编码顺序排列。  
   **期望结果**：分组过滤准确，搜索实时生效，索引定位准确，无权限问题。
2. **自定义字段创建与前端渲染**  
   **前提**：管理员登录。  
   **步骤**：打开系统设置->自定义字段 -> 新增字段 名称="担当者", 分组="基本情報" -> 保存。然后进入任一客户详情页。  
   **期望**：客户详情“基本情報”区域出现“担当者”字段，值为空。编辑该客户->看到“担当者”输入框->填值并保存->详情页实时显示填写值。审计日志记录此字段的添加和修改动作。
3. **标签颜色/过滤**  
   **步骤**：管理员新增标签"重要"颜色红色并赋给客户A。普通用户登录->客户列表点标签过滤选择"重要" -> 仅客户A出现。详情页客户A标签区呈红色"重要"标签。  
   **期望**：颜色和文字正确渲染。过滤后列表结果正确无其他客户。
4. **流程设计/执行模式切换**  
   **步骤**：创建新计划P含两节点->发布计划->进入执行模式UI。  
   在设计模式下验证：可新增、编辑、删除节点，顺序调整能够保存，再发布。发布后UI切换执行模式：节点列表不可编辑，有开始/完成按钮。  
   **期望**：设计模式所见节点顺序与发布后执行模式一致。发布操作将状态变为执行准备。未发布前无执行控件，发布后设计控件消失。
5. **节点动作调用与审计**  
   **用例**：计划执行中包含一个EMAIL节点、一个REMOTE节点。  
   **步骤**：执行EMAIL节点 -> 系统发送邮件 (可检查测试收件箱) -> Audit日志有发送记录。执行REMOTE节点 -> 点击"打开连接" -> 系统记录一次Remote\_History，Audit日志记录用户X点击远程连接。上传文件附件->Audit日志记录上传操作。  
   **期望**：Email实际收到（如果SMTP通），至少日志里记录attempt -> audit\_log有 "EMAIL sent to ..."，Remote audit有 "User X triggered remote action Y", 文件 audit 有 "User X uploaded file Z for node".
6. **计划导出ICS并Outlook订阅**  
   **前提**：有计划P时间未来。  
   **步骤**：管理员在计划详情点击"导出ICS"，保存文件P.ics。用文本打开检查包含事件SUMMARY=计划标题，DTSTART=计划开始时间(JST)等。将P.ics导入Outlook日历 -> 确认日历上出现对应事件。再尝试订阅webcal：使用链接 webcal://<server>/api/v1/calendar/tenant/1.ics 在Outlook添加 -> 触发后台日志显示访问 -> 人工修改计划标题 -> 等待Outlook刷新或手动触发 -> 检查Outlook事件名更新 (考虑Outlook延迟这可能手动验证)。  
   **期望**：ICS文件格式正确，被Outlook接受并显示；订阅链接工作（Outlook可能不立即刷新，在测试可检查Outlook无报错订阅成功即可）。
7. **用户自注册-激活-登录-改密**  
   **步骤**：打开注册页->输入邮箱test@example.com等->提交->系统提示邮件发送->查看测试SMTP收到激活邮件包含链接->点击链接->前端显示激活成功->使用用户名密码登录->成功进入。然后在个人设置修改密码->使用新密码可登录，旧密码不能。  
   **期望**：未激活前登录被拒绝，激活后可登录。修改密码后旧密码失效。
8. **审计日志记录字段级前后值**  
   **步骤**：管理员修改客户A的某字段 (如特記事項从 "abc" 改 "xyz") -> 查询审计日志筛选 entity=Customer, id=A -> 最新一条action=UPDATE, detail应包含"特記事項: 'abc' -> 'xyz'".  
   **期望**：审计记录准确捕捉更改，尤其前值后值清楚[[5]](https://community-archive.progress.com/forums/00019/21255.html#:~:text=archive,new%20values%20is%20fully%20available)。其他操作如添加用户、删除计划也有日志。
9. **MinIO上传/下载与DB一致性**  
   **步骤**：上传文件 X 通过系统-> File\_Object产生记录 -> 直接用MinIO client或 UI 查看bucket里object存在。然后通过系统下载接口获取URL下载文件->内容完整。删除该文件->MinIO对象消失->DB记录删除。  
   **期望**：文件元数据与实际存储匹配；删除后无残留。

每条用例测试通过则标记通过。如有不符，需调试修正后重新测试。验收时可由客户随机指定若干场景现场演示以确认。

性能和安全测试需达到指标：如并发100时系统平均响应<800ms，无错误；OWASP Top10无高危漏洞；以上功能测试100%通过。满足这些则可认为达到验收标准“研发与测试照此即可实现与验收”。

## 12. 迁移与集成

**数据迁移（客户CSV导入）：** 系统应提供将遗留的客户信息导入功能，以加快上线初始数据准备。给定用户提供的“客户信息列表（CSV）”文件（见附件），需设计映射规则和导入工具。

* **CSV格式分析**：附件示例（冒头几行）显示列：
* 機関コード (机构代码)
* 機関名 (机构名)
* 接続形態 (连接形态)
* 更新情報 (更新信息)
* 接続ツール (连接工具)
* 接続制限 (连接限制)
* 特殊運用 (特殊运用)
* 顧客別情報参照 (客户别信息参照)
* 特記事項 (特记事项)
* 接続先IPアドレス/電話 (连接目标IP/电话)
* 之后可能还有许多列 (示例显示 Unnamed: 8..)，但实际意义可能是补充说明、多个IP电话等。猜测CSV在Excel里有合并单元格或换行。 但至少从前几列看，我们关心如上字段。
* **系统字段映射**：
* 客户基本信息：
  + CSV“機関コード” -> Customer.code
  + CSV“機関名” -> Customer.name
  + 若CSV包含“所在地”或类似(没看到，但region可不填或由外部给)
* 自定义字段定义：我们可将 CSV其余列大部分作为自定义字段，而非拓展Customer固定字段。具体：
  + “接続形態” -> 定义 custom\_field\_def (field\_key=connection\_type, 名称=接続形態)
  + “更新情報” -> field\_key=update\_info (名称=更新情報)
  + “接続ツール” -> connection\_tool (名称=接続ツール)
  + “接続制限” -> connection\_limit (名称=接続制限)
  + “特殊運用” -> special\_operation (名称=特殊運用)
  + “顧客別情報参照” -> info\_reference (名称=顧客別情報参照)
  + “特記事項” -> note (名称=特記事項)
  + “接続先IPアドレス/電話” -> remote\_address\_phone (名称=接続先IPアドレス/電話)
* 这样8个字段都可预先在系统配置好对应自定义字段（通过后台或直接SQL插入）。
* 标签：看CSV内容，没有直接的标签列，但可以推导一些标签:
  + 如果接続形態 = VPN / アナログ / etc，可以将不同类型作为标签定义（比如标签“VPN”，“アナログ”），然后在导入时给对应客户打上该标签。这其实重复了字段信息，但标签用于筛选视图，所以有用。Alternatively, 只用字段就够，但要求里有标签过滤，所以建议在导入时：对常见连接类型、工具等生成标签以方便过滤。
  + Example: CSV row says 接続形態=VPN -> mark tag "VPN".
  + If CSV had something like "統合しているためグレーアウト" (like in row 3 special notes), maybe that implies a tag like "統合先"? But uncertain. Possibly skip.
  + At least ensure major categories like connection type as tags.
* 分组：CSV没有明确分组字段比如“所属組織类型”，不过从机构名可猜都是大学。我们可统一把国立大学放一起；CSV Row 1-4 all seems universities. Perhaps all entries are universities (maybe not, name suggests so). If needed, we can categorize by region or institution type (if code indicates region).
  + 例えば、機関コード0100=北海道大学, likely "0100" series for Hokkaido region? Actually not sure. The code might not correlate to region, maybe just unique.
  + Alternatively, we can assume all are "大学" group and later admin can refine grouping if needed.
* 顾客ID: We'll need to assign our Customer.id (serial). Possibly we want to preserve the 機関コード in code field. That code might be unique, we can use as code.
* **导入工具**：
* 提供一个导入脚本，可以是Python或 CSV parser in Java admin CLI.
* 步骤:
  1. 预先在系统创建好 custom\_field\_def entries for each relevant field (with appropriate data types). This can be done via a migration or just before import.
  2. 脚本读取CSV文件 (UTF-8 or SHIFT\_JIS?), parse lines.
  3. For each line, do:
  4. Trim whitespace, unify encoding (some might have newlines inside columns as seen).
  5. Construct a Customer object:
     + code = 機関コード (e.g. "0100")
     + name = 機関名
     + group = default group (like "大学" if we want, else none).
     + region: If we can parse from name? Might skip or fill later by admin.
     + Then for each custom field, prepare Customer\_Custom\_Field\_Val:
     + find field\_def by name or key, then set value = that column.
     + Determine tags:
     + if 接続形態 value is not empty, find tag\_def with same name (or create if not exist).
     + same for 接続ツール possibly as tag.
     + Possibly if connection requires "要" (顧客別情報参照 = 要?), could set a tag "要参照".
     + If 特記事項 mention something like "V6バージョンアップ", maybe skip tagging that level of detail.
     + Save customer (via API or direct DB).
     + Save each custom field val.
     + Save customer\_tag if any.
  6. The script should be idempotent:
  7. If run second time, it should not create duplicates. So check if Customer.code already exists:
     + If yes, either skip or update existing record.
     + We can allow updating fields if CSV changed (but initial load likely one-time).
  8. Also ensure custom\_field\_def and tag\_def creation are done only once (check existence by key/name).
  9. Report any errors (like missing required fields, or if code duplicates).
* This script can be delivered with the project for one-time use. Could also integrate into admin UI an import function where admin uploads CSV and server processes, but likely easier offline.
* **编码**: CSV可能是Shift-JIS if it's from a Japanese source. Need to ensure reading with correct encoding to get Japanese text properly. Also ensure our DB uses UTF-8 (Postgres usually does, or set Collation).
* **异常行处理**: If some row has missing code or name, script can skip and log "Line X missing essential field, not imported". Output a summary of how many imported, skipped, duplicates etc.
* **幂等**: As said, using code as natural key ensures multiple runs won't create new entries for same code. If code changed in CSV (rare), then it's a new record or we treat as changed code which we might not handle automatically. But let's assume codes stable.

**邮件集成 (SMTP)**: - Our TemplateService uses SMTP config to send emails: - Provided environment variables for host, port, user, pass. Possibly TLS needed (if port 587). - Use JavaMail or Spring's MailSender. - We should test with a known SMTP (maybe a local fake SMTP in dev). - One limitation: If SMTP fails or slow, how to handle? We probably just attempt and catch error, log audit as failure. Could consider asynchronous queue for email sending to not block user operation if needed. But as initial, maybe sync is fine if quick. - **模板变量**: - We should define what variables are available for templates: - Likely global: {{customer\_name}}, {{plan\_title}}, etc depending on context of use. - For a "通知メールテンプレート" (maintenance notice), needed info could be customer, plan time, maybe contact info. - Possibly allow referencing any custom field: e.g. {{customer.接続形態}} if needed. But implementing a full expression might be complex. - Simpler: define a dictionary to feed into template: \* For plan-related email: supply customer\_name, plan\_title, plan\_start, etc. \* For general: supply known ones or instruct user what can use. - In UI for template editing, we can list available placeholders per template type. e.g. Email for plan might support {customer\_name}, {plan\_time}, etc. - Implementation: use a simple replace or MessageFormat, so format would be "Hi {0}" etc. Or define moustache style and use a small moustache library or simple regex replace for {{key}}. - Decision: probably use a basic library or custom: - Could use Spring's MessageFormat if placeholders are numeric like {0}, or use handlebars/mustache via a library for Java (like Mustache.java or handlebars). - Or simplest: string replace, but careful with curly braces colliding in text. Possibly find {{\w+}} with regex and replace from map. - Provide example: \* Template: "メンテ開始：{{start\_time}}\n対象：{{customer\_name}}" \* Variables map: {"start\_time": "2025/10/01 10:00", "customer\_name": "北見工大"} -> replaced accordingly. - Also for IM templates, similar approach. - Integration testing such replacing logic with actual multiline content is needed.

**IM集成**: - The requirement mentions "IM (Webhook/机器人)". So likely Slack or Teams or LINE etc. - If Slack: we can get an Incoming Webhook URL, which when posted to will drop message in a channel. Template\_IM.endpoint can store that URL or part of it, or we might store at global config and just store channel id. - Possibly generic: Each Template\_IM has an endpoint (like full URL for webhook) and message content. - Then TemplateService does: - For sendIM: if endpoint present, do an HTTP POST (Slack expects JSON with text). - If endpoint not present but perhaps configured globally (some systems might code in) then also manage accordingly. - If multiple IMs needed, we might integrate with known APIs: - For Microsoft Teams, similar incoming webhook method exists. - If one wanted to send via LINE bot or others, they'd need API keys etc, probably out-of-scope. We can mention possibility but not implement by default. - We'll treat it generically: just HTTP POST to given URL with JSON {"text": "<message>"} if Slack or similar. We may need to adjust for different format per platform: - Slack and Teams both accept a JSON with "text" for basic message. - We could just assume Slack formatting, for others user may tailor by customizing Template\_IM as needed or using that same format. - If no external IM needed, at least we show architecture can support.

**远程连接模板 integration**: - As pointed, fully executing RDP/SSH from a web app is tricky. - We mention "预留执行接口，后续使用其他技术接口触发执行" means perhaps in future we have an agent or remote execution service. - For now: - The remote template might store info. What can we do with it now? - Option 1: Provide the connection details to user to manually connect. - Option 2: Provide a file or link to aid connection (like RDP file). - RDP: - .rdp file is basically text with settings. We can generate a minimal one:

screen mode id:i:1  
use multimon:i:0  
Full Address:s:158.208.128.219  
Username:s:someusername

Save that as .rdp, when user opens it, Windows Remote Desktop will launch. - We have to include the credentials? Usually, you wouldn't include password in RDP file (there is a way but it's encrypted with user certificate). So likely not, just server address, maybe username if not sensitive, and user will be prompted. - Could be helpful if many servers. - SSH: - Could create an ssh:// link. If user has a handler (like if they installed an SSH app that registers ssh:// scheme), clicking it might open that app. - Many Windows machines might not have such by default. On Mac/Linux, maybe Terminal can handle ssh:// link. - Alternatively, just show "ssh user@host -p port". - VPN: - Possibly just instruct "connect to VPN using GlobalProtect with given credentials". - Or if GlobalProtect has a URI scheme (some apps do have custom URI). - For now, likely just show "VPN GlobalProtect needed". - Considering these are mostly manual, we choose: - Provide relevant info on UI (like "Protocol: RDP, Host: X, User: Y"). - Provide a "Download RDP file" button if RDP template. - Provide a "Launch SSH" button if we can format an ssh:// link. For example, ssh://username@host:22 might prompt OS to open default ssh client if any. Possibly not widely supported except PuTTY maybe has its own scheme (like putty://). - Without reliable cross-platform method, maybe skip direct open, just copy details to clipboard or show QR code if mobile, etc. Probably out-of-scope. - Summation: For RDP, implement .rdp download. For others, just present info or minor. - If we had remote execution: - Eg. integrated with a remote management tool's API to actually run a script or connect, then Template\_Remote could store the required parameters and the new ActionHandler would handle connecting or instructing the agent. - But absent an actual remote agent, we just design for possible future injection (like said, plugin interface). - So in code: - remote ActionHandler do: If protocol == "RDP": generate .rdp content, store maybe in MinIO or directly serve as file download (like one-time link). Or maybe on clicking "Open RDP", front-end calls an endpoint which responds with file content and a Content-Disposition forcing download as "connection.rdp". That's possible with a Spring endpoint producing application/rdp (which is basically plain text with .rdp extension). - For SSH: front-end could simply present a clickable link ssh://user@host in anchor. If OS has a handler (some might not), worst case it just gives not found. Or have a copy icon to copy "ssh user@host". - For VPN: present info (like "Use GlobalProtect to connect to IP X"). - If a remote style not covered, at least user can retrieve needed info from those fields.

**附件客户信息列表CSV导入脚本** (Pseudo or description): We can include as a demo or in appendices code snippet: like a Python snippet reading csv and calling API.

Example pseudo-code:

import csv, requests  
with open('customers.csv', encoding='cp932') as f:  
 reader = csv.DictReader(f)  
 for row in reader:  
 code = row['機関コード']  
 name = row['機関名']  
 # build payload  
 fields = {  
 "接続形態": row['接続形態'],  
 "更新情報": row['更新情報'],  
 ...  
 }  
 payload = {  
 "code": code,  
 "name": name,  
 "fields": fields,  
 "tags": []  
 }  
 # determine tags  
 if row['接続形態']:  
 payload["tags"].append(tag\_map[row['接続形態']])  
 # etc for tools  
 # call API with admin token  
 resp = requests.post(API\_URL+"/customers", json=payload, headers={"Authorization":f"Bearer {token}"})  
 ...

This script usage would be manual, but can be included in docs or as part of deliverables.

**邮件SMTP**: - Provide configuration interface or at least environment config as we did. Could also have a UI page for admin to set SMTP settings (some apps do so you can change without redeploy). - Considering MVP, config via environment is fine, assume stable. - If multiple customers each with own SMTP (for sending from their domain?), out-of-scope likely. One global SMTP is enough for notifications from the system.

**IM Webhook**: - Provide a configuration to store default webhook per channel or per usage? Our Template\_IM has endpoint per template. So for Slack, each channel you'd create separate Template\_IM with respective webhook. - For example, a template "OpsAlert" has a Slack webhook for ops channel, you might have another "ClientNotice" for a Teams channel to client, etc. - If wanting to use one template to send to different customers channels, you'd need to store multiple endpoints or one endpoint with placeholder part (like include in template, but that complicates). - Simpler: separate templates for each channel. - But if one wants e.g. each customer has a different Slack channel, perhaps Template\_IM could include placeholders replaced by some customer field containing channel URL or name. Not doing now.

**远程连接触发**: - Emphasize to stakeholders: currently platform does not directly remote control, it's just storing info. - But future integration could be through: - Launching remote control via native app: e.g. building a small local agent that listens on a port or via custom protocol, which the web can call (some enterprise solutions do that). - Or integration with systems like RPA or remote management. Possibly out-of-scope, but mention extension points.

## 13. 交付清单

项目交付需提供以下文档和资源，确保研发、运维及测试团队可以顺利使用和二次开发：

1. **《BOB MTA综合运维平台 详细设计说明书》** – 即本说明书，包含所有架构设计、模型、接口规范和图表。建议提供可编辑格式和PDF，用于评审及存档。
2. **数据库DDL脚本** – 包括创建所有表、索引、约束的SQL文件。例如 schema\_postgresql.sql，开发可直接执行初始化数据库。应与ERD一致，包含初始基础数据插入（如基本角色、权限、默认管理员账户如需要）。
3. **示例数据** – 若有，提供一些CSV/SQL示例数据文件，如客户CSV文件样例，或初始化演示用的小数据集，用于开发联调测试。比如 sample\_customers.csv 对应附件，以及一些Plan/Template示例SQL插入。
4. **多语言词条文件** – 提供默认日文和中文的资源文件（JSON或Properties）。如 messages\_ja.properties, messages\_zh.properties 或 i18n\_messages.sql（若直接数据库插入）。以便后续翻译和校准。
5. **Docker Compose** – 完整的 docker-compose.yml 文件和相关配置文件（nginx.conf, .env.example）。确保开箱即用部署所有组件。附带 .env.example 列出所需环境变量及说明（如JWT\_SECRET\_KEY如何生成，SMTP如何配置等）。
6. **接口测试集合** – Postman或Insomnia的API集合文件(JSON格式)，包含所有主要接口的定义、示例请求和响应。测试人员可用其方便地调用各接口验证。应包括认证流程（获取token）和各模块CRUD操作示例。
7. **前端路由与菜单配置表** – 列出前端实现的路由路径、对应页面组件、以及导航菜单项与权限的对应关系的文档。例如表格：路径/customers – 页面CustomerPage – 菜单“客户一览” – 访问权限“All Users”；/admin/users – 页面UserAdminPage – 菜单“用户管理” – 权限“Admin”。方便运维和测试检查菜单权限。
8. **代码仓库地址及分支说明** (如果适用) – 提供源码版本库的URL，及推荐的分支模型 (如 main/dev branches)，以及如何检查代码提交历史 (以验证遵循规范)。
9. **其他配置文件** – 如后端application.yaml样例（脱敏后），说明重要配置项含义；前端打包配置等（若需复现构建）。
10. **用户手册初稿** (选配) – 虽不在研发交付重点，但如有初步用户操作说明，包括如何添加客户、发布计划、订阅日历等简要指引，会有助测试和用户验收。

以上清单确保交付内容完整，使接收方能够按照说明成功部署、测试并使用系统。所有文档需保持一致和最新状态，如在开发中有出入，以详细设计说明书和代码为准并在交付前更新。

## 14. 附录

**代码风格规范：**

* **后端Java**：遵循Java标准编码规范和团队约定，可参考Google Java Style。关键：
* 包命名全小写，类名CamelCase，常量全大写下划线。
* 使用4空格缩进，不使用Tab。
* 每行尽量不超过120字符，方法注释使用Javadoc格式描述作用。
* 控制语句（if/for等）使用括号，即使单行也明示。
* 日志使用SLF4J的logger，占位符填参数避免字符串拼接。敏感信息不打印。
* 注释必须准确反映代码意图，不要陈述显然易见逻辑。复杂算法加详解说明。
* Spring组件如Service/Controller类添加类注释描述用途。Public方法写Javadoc注释。
* SQL使用MyBatis XML，在XML中缩进对齐，关键字大写（SELECT, WHERE）。
* 单元测试代码命名以Test结尾，每个方法测试一个场景，使用有意义的断言消息。
* **前端TypeScript**：使用ESLint + Prettier统一风格。如Airbnb或Standard风格为基础。
* 使用TypeScript类型，避免使用any，能明确接口类型就定义interface。
* 组件文件名大写开头 (PascalCase.jsx/tsx)，其他工具模块camelCase.ts。
* JSX中属性多行时每个属性换行缩进对齐。尽量自闭标签 <Component prop="x" />。
* 状态变量、函数命名清晰语义。避免缩写。
* 行尾不加分号 (若采用Standard)，或统一加分号（Airbnb）。关键是全局一致。
* 引入AntD组件时，尽量按需加载以减小包大小（babel-plugin-import）。
* CSS使用Less/Sass模块或 styled-components，类名使用BEM或CamelCase，防止全局污染。
* 注释：复杂逻辑处添加行内注释。函数/组件上方用JSDoc简单描述用途、参数。
* Git提交前跑Prettier格式化，避免diff中出现样式问题。

**提交规范 (Conventional Commits)**: 采用约定式提交消息，格式：

<type>(<scope>): <subject>

如： - feat(customer): 支持自定义客户字段导入 - fix(plan): 修复流程节点阈值判断错误 - docs: 更新详细设计说明书附录 - refactor(auth): 重构JWT验证逻辑 Type类别主要有： - feat (新功能) - fix (Bug修复) - docs (仅文档变更) - style (格式改动, 不影响代码逻辑的如空格) - refactor (重构既有代码，不加新功能不修bug) - perf (性能优化) - test (添加或修改测试) - chore (构建或辅助工具变动) Scope 用模块或文件名标识影响范围，如customer, plan, auth, etc. Subject简洁明了，建议日英均可，这里团队内或用中文。但为了有意义，英文可能通用。可在body加详细说明和影响。

所有成员遵循此规范有助于生成CHANGELOG、回溯问题。

**分支模型**: - 项目选择 Git Flow 模型：有长期分支 main(或master)、develop，以及功能分支 feature/*, 补丁分支 hotfix/*, 版本分支 release/*。流程： - 日常开发基于 develop，新的需求/任务开 feature 分支，完成后PR合并回 develop。 - 定期从 develop 切 release/x.y 分支做集成测试和bugfix，稳定后合并到 main，打 tag 发布版本。然后release分支也合回 develop (带上release期间的修复)。 - 紧急生产修复从 main 切 hotfix分支，修好后合并 main 和 develop。 - 如果团队规模小、持续集成快速，也可采用 Trunk-Based 模式： - 即主要在 main 分支开发，通过 feature toggle 控制未完成特性，上CI跑测试后直接部署。开发者feature分支寿命很短，完成功能即合并 main。 - 本项目可根据实际情况决定。但GitFlow能较好管理发布节奏，因此推荐*\* Git Flow在较正式交付阶段使用。开发初期节奏快可在 develop上直接版本迭代。

**错误码字典及国际化消息模板**: - 定义错误码和对应消息的对照表。例如： | 错误码 | 含义 (中文) | 日文消息 | |-------|--------------------|----------------------------------| | 0 | 成功 | (不需要消息，一般用 "成功") | | 4000 | 参数校验失败 | リクエストパラメータ不正です。 | | 4010 | 未认证（需要登录） | 認証が必要です。ログインしてください。 | | 4031 | 权限不足 | 権限がありません。管理者にお問い合わせください。| | 4040 | 资源不存在 | リソースが見つかりません。 | | 4090 | 操作冲突/业务规则不符 | 要求は現在の状態と矛盾しています。 | | 5000 | 服务器未知错误 | サーバーエラーが発生しました。 | - 在后端配置这些错误码，返回时附带对应 message key 或直接消息文本。前端也可以有同样映射用于展示（若后端只给码，前端本地解析；或后端给消息，前端直接显示）。 - 国际化模板：将这些消息加入 i18n资源，比如 error.unauthorized: "認証が必要です" 这样。使得无论后端还是前端都可以引用统一翻译。 - 统一管理错误码有利于测试枚举各种错误场景和前端处理。 - 其他如验证错误消息(Spring Validation)返回时最好转换为规范格式（列表错误详情，并翻译字段名)。可以提供前端统一显示。

**典型性能基线与压测脚本思路**: - 性能基线：在Staging环境（4Core8GB服务器,PG在同机或独立）下，100用户并发情况下主要接口性能: - 登录：TPS >= 20, 平均响应 < 300ms - 查询客户列表(1000客户)：TPS >= 15, < 500ms - 查询计划日历(当月30计划)：TPS >= 10, < 400ms - 完成节点(带日志文件上传)：< 800ms including upload - 系统资源利用：CPU < 70% 平均, PG CPU < 50%, Memory足够无明显GC长暂停。 - 压测可使用JMeter: - 设计线程组100用户，随机执行以下流程: 登录->获取客户列表->打开随机客户详情->打开计划页面->随机挑选一个计划详情->如有当前节点可完成则调用complete->退出。 - 文件上传压测：用 JMeter JMS or custom plugin for S3? Or skip heavy file, or simulate with small file in loop. - 测量90%Line响应是否在目标内。 - 关注长事务：比如一次导入处理大量数据，会不会锁表影响别的操作。测试批量操作时候的表现。 - 通过这些脚本，找出瓶颈SQL（explain analyze查看slow queries，如 tags join查询)进行调优（加索引等）。

所有这些附录规范帮助团队在开发过程中保持一致性，提高质量，并减少沟通成本，使项目顺利进行和交付。

[[1]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=Note%3A%C2%A0When%20you%20subscribe%20to%20a,take%20more%20than%2024%20hours) [[2]](https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c#:~:text=When%20you%20import%20an%20,or%20phases%20of%20the%20moon) Import or subscribe to a calendar in Outlook.com or Outlook on the web - Microsoft Support

<https://support.microsoft.com/en-us/office/import-or-subscribe-to-a-calendar-in-outlook-com-or-outlook-on-the-web-cff1429c-5af6-41ec-a5b4-74f2c278e98c>

[[3]](https://phrase.com/blog/posts/database-stored-messages-for-i18n-in-spring-boot/#:~:text=The%20most%20common%20approach%20to,is%20not%20an%20optimal%20approach) [[4]](https://phrase.com/blog/posts/database-stored-messages-for-i18n-in-spring-boot/#:~:text=Hence%2C%20we%20are%20going%20to,existing%20localized%20messages%20at%20runtime) Database-Stored Messages for I18n in Spring Boot | Phrase

<https://phrase.com/blog/posts/database-stored-messages-for-i18n-in-spring-boot/>

[[5]](https://community-archive.progress.com/forums/00019/21255.html#:~:text=archive,new%20values%20is%20fully%20available) Best practice for record based log of changes. - Forum

<https://community-archive.progress.com/forums/00019/21255.html>

[[6]](https://www.sqlservercentral.com/forums/topic/audit-trail-logging-specific-changes-on-a-column-by-column-basis#:~:text=Part%20of%20the%20purpose%20of,so%20that%20they%20can) Audit Trail / Logging Specific Changes on a Column by Column Basis

<https://www.sqlservercentral.com/forums/topic/audit-trail-logging-specific-changes-on-a-column-by-column-basis>