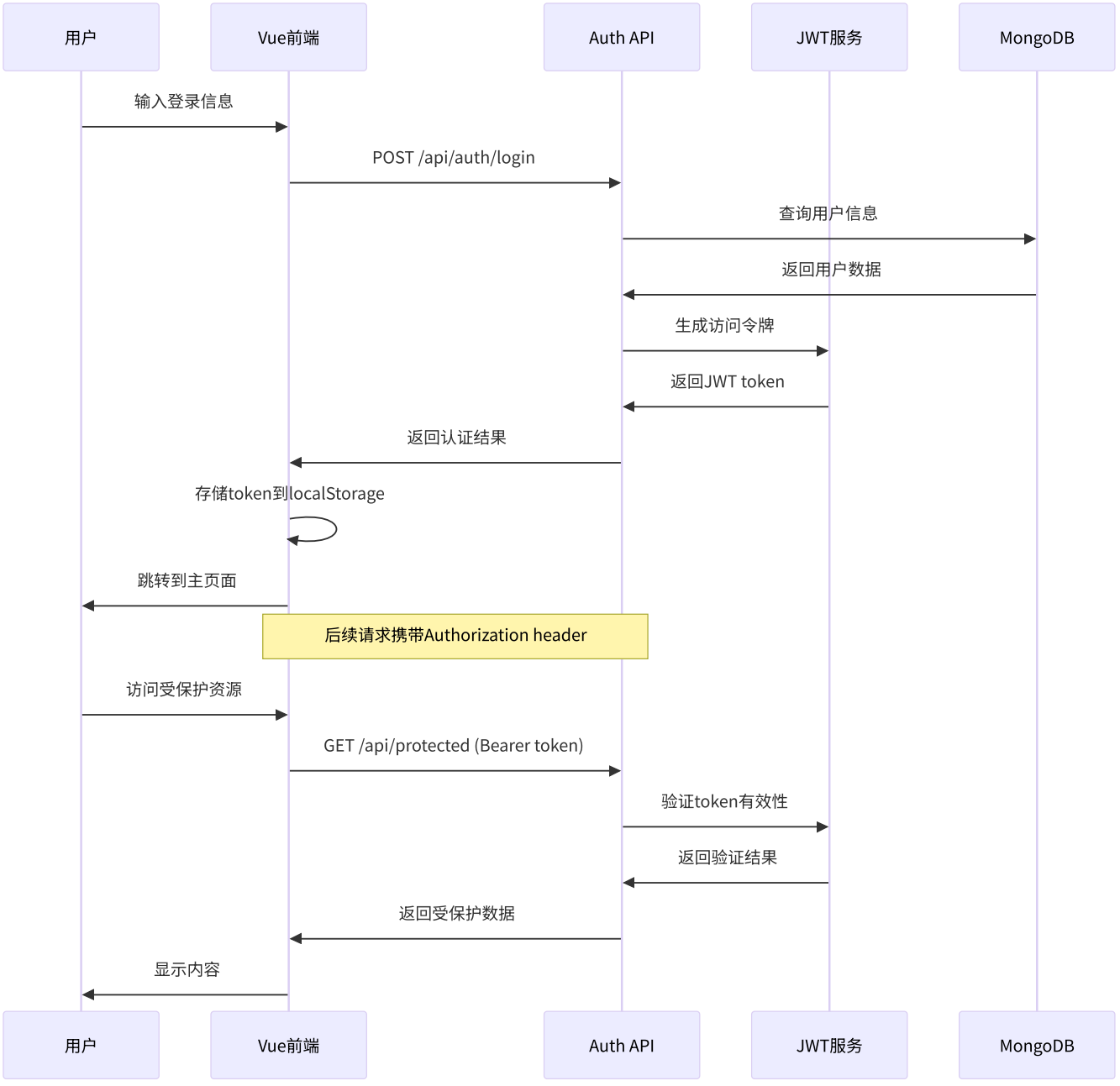
用例交互图

1. 用户认证与授权流程



详细解释

技术实现细节：

这个认证流程基于项目中的JWT（JSON Web Token）认证机制。前端使用Vue.js框架，通过Axios库发送HTTP请求到Express.js后端API。

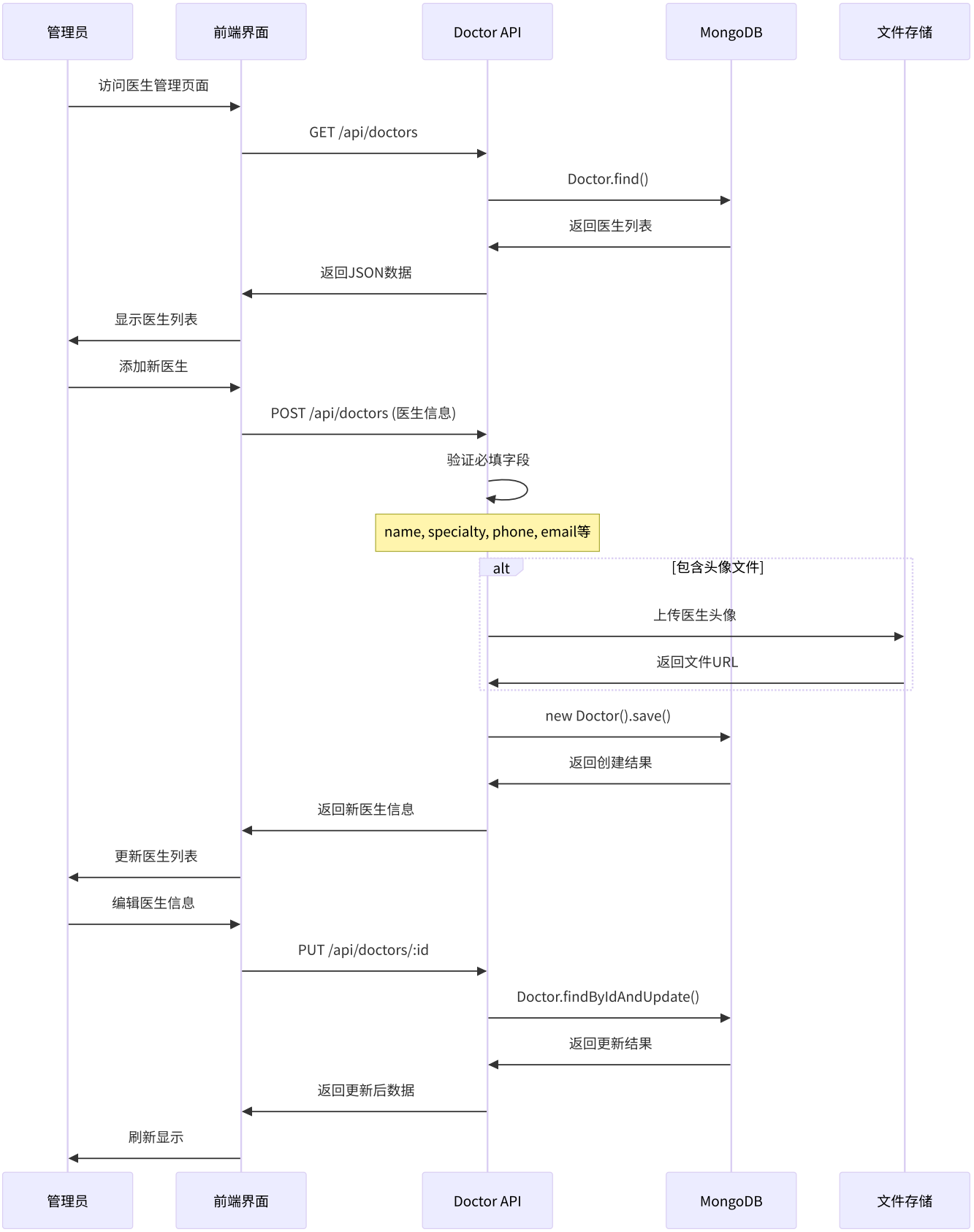
具体流程分析：

1. **用户登录阶段：**
   - 用户在前端登录表单中输入用户名和密码
   - Vue前端调用 `this.$http.post('/api/auth/login', credentials)` 发送登录请求
   - 后端Express路由接收请求，使用bcrypt库验证密码哈希值
   - MongoDB通过Mongoose ODM查询用户记录，执行 `User.findOne({email: email})`
   - 验证成功后，使用jsonwebtoken库生成JWT：`jwt.sign(payload, secretKey, {expiresIn: '24h'})`
2. **Token存储与使用：**
   - 前端接收到JWT后存储在localStorage中：`localStorage.setItem('token', token)`
   - 后续API请求都在header中携带token：`Authorization: Bearer ${token}`
   - 后端中间件验证每个受保护路由的token有效性
3. **安全机制：**
   - Token设置过期时间，通常为24小时
   - 使用环境变量存储JWT密钥
   - 实现了自动token刷新机制防止会话过期

代码对应关系：

- 前端：`src/store/auth.js` 中的login action
- 后端：`routes/auth.js` 中的login路由
- 中间件：`middleware/auth.js` 进行token验证

2. 医生信息管理系统（基于backend/models/Doctor.js）



详细解释

数据模型设计：

基于`backend/models/Doctor.js`中的Mongoose Schema，医生模型包含以下核心字段：

```
{
  name: String (必填),
  specialty: String (专科),
  phone: String,
  email: String (唯一),
  avatar: String (头像URL),
  experience: Number (工作年限),
  education: String (教育背景),
  hospital: String (所属医院),
  isActive: Boolean (是否活跃)
}
```

CRUD操作实现：

1. **查询医生列表：**
   - 前端调用`GET /api/doctors`获取所有医生
   - 支持分页查询：`Doctor.find().skip(offset).limit(pageSize)`
   - 可按专科筛选：`Doctor.find({specialty: req.query.specialty})`
   - 返回结果包含医生基本信息和统计数据
2. **创建新医生：**
   - 使用express-validator进行输入验证
   - 检查邮箱唯一性：`Doctor.findOne({email: req.body.email})`
   - 如果包含头像文件，使用multer中间件处理上传
   - 文件存储到`uploads/doctors/`目录或云存储服务
   - 创建医生记录：`new Doctor(doctorData).save()`
3. **更新医生信息：**
   - 验证医生ID的有效性
   - 使用`findByIdAndUpdate()`方法更新指定字段
   - 设置`{new: true}`返回更新后的文档

- 支持部分字段更新，避免覆盖未修改的数据

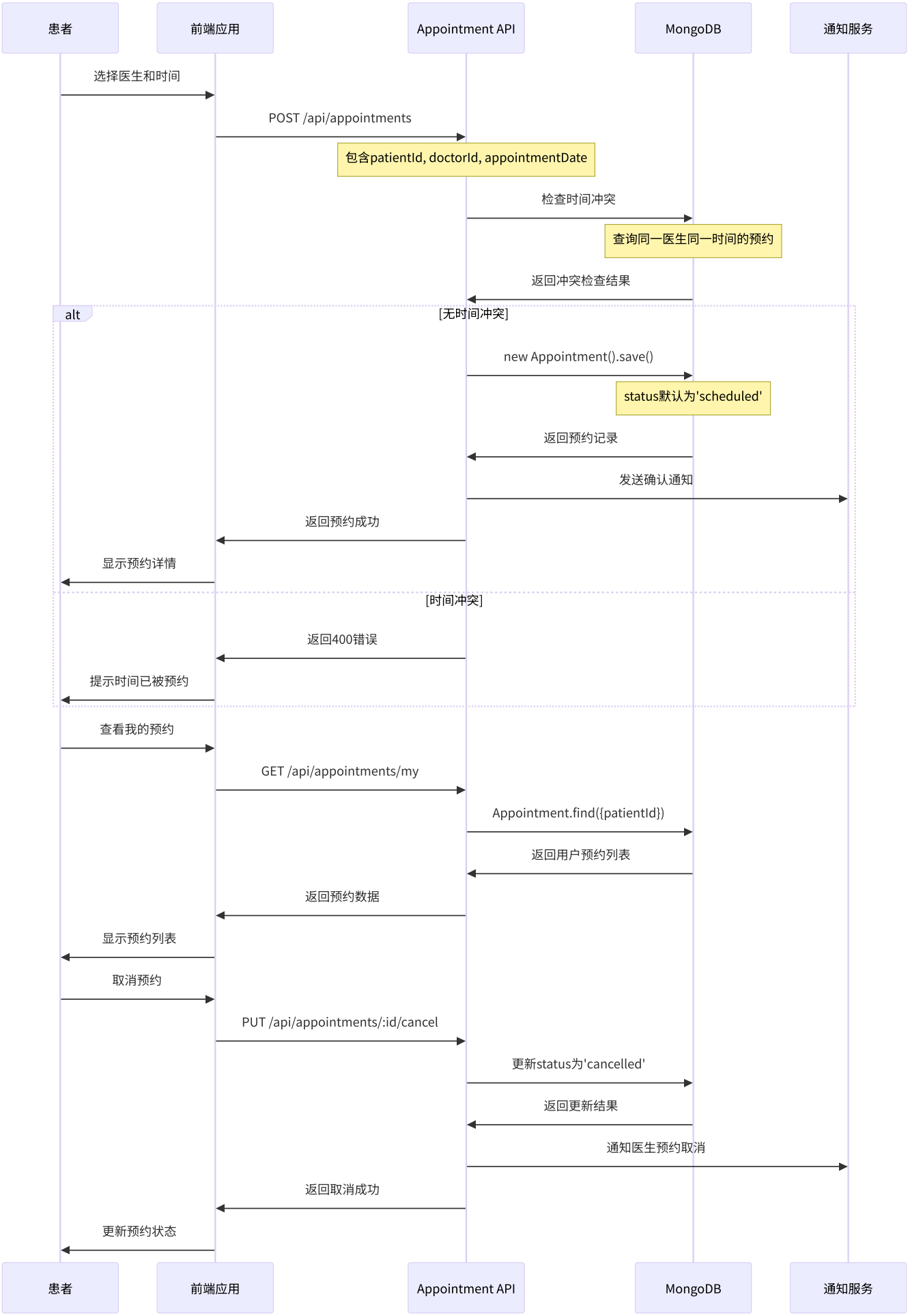
权限控制：

- 只有管理员角色可以进行医生信息的增删改操作
- 使用中间件验证用户角色：`requireRole('admin')`
- 普通用户只能查看医生公开信息

错误处理：

- 统一的错误响应格式
- 数据库约束违反的友好错误提示
- 文件上传失败的回滚机制

3. 预约管理系统（基于backend/models/Appointment.js）


患者
前端应用
Appointment API
MongoDB
通知服务
选择医生和时间
POST /api/appointments
包含patientId, doctorId, appointmentDate
检查时间冲突
查询同一医生同一时间的预约
返回冲突检查结果
alt
[无时间冲突]
new Appointment().save()
status默认为'scheduled'
返回预约记录
发送确认通知
返回预约成功
显示预约详情
[时间冲突]
返回400错误
提示时间已被预约
查看我的预约
GET /api/appointments/my
Appointment.find({patientId})
返回用户预约列表
返回预约数据
显示预约列表
取消预约
PUT /api/appointments/:id/cancel
更新status为'cancelled'
返回更新结果
通知医生预约取消
返回取消成功
更新预约状态
患者
前端应用
Appointment API
MongoDB
通知服务

详细解释

预约数据模型：

基于`backend/models/Appointment.js`的Schema设计：

```
{
  patientId: ObjectId (患者ID, 关联User),
  doctorId: ObjectId (医生ID, 关联Doctor),
  appointmentDate: Date (预约时间),
  duration: Number (预计时长, 分钟),
  status: String (scheduled/completed/cancelled/no-show),
  reason: String (预约原因),
  notes: String (医生备注),
  createdAt: Date,
  updatedAt: Date
}
```

核心业务逻辑：

1. 时间冲突检测：

```
const conflictCheck = await Appointment.findOne({
  doctorId: req.body.doctorId,
  appointmentDate: {
    $gte: startTime,
    $lt: endTime
  },
  status: { $ne: 'cancelled' }
});
```

   - 检查同一医生在指定时间段内是否已有预约
   - 考虑预约时长，确保时间段不重叠
   - 排除已取消的预约

2. 预约状态管理：
   - `scheduled`：已预约（默认状态）
   - `completed`：已完成
   - `cancelled`：已取消
   - `no-show`：爽约
   - 状态转换有严格的业务规则限制

3. 通知机制：
   - 预约成功后发送确认邮件/短信给患者和医生
   - 预约前24小时发送提醒通知
   - 预约取消时通知相关方
   - 使用队列机制处理通知任务

查询功能实现：

1. 患者查看预约：

```
Appointment.find({patientId: req.user.id})
  .populate('doctorId', 'name specialty')
  .sort({appointmentDate: -1})
```

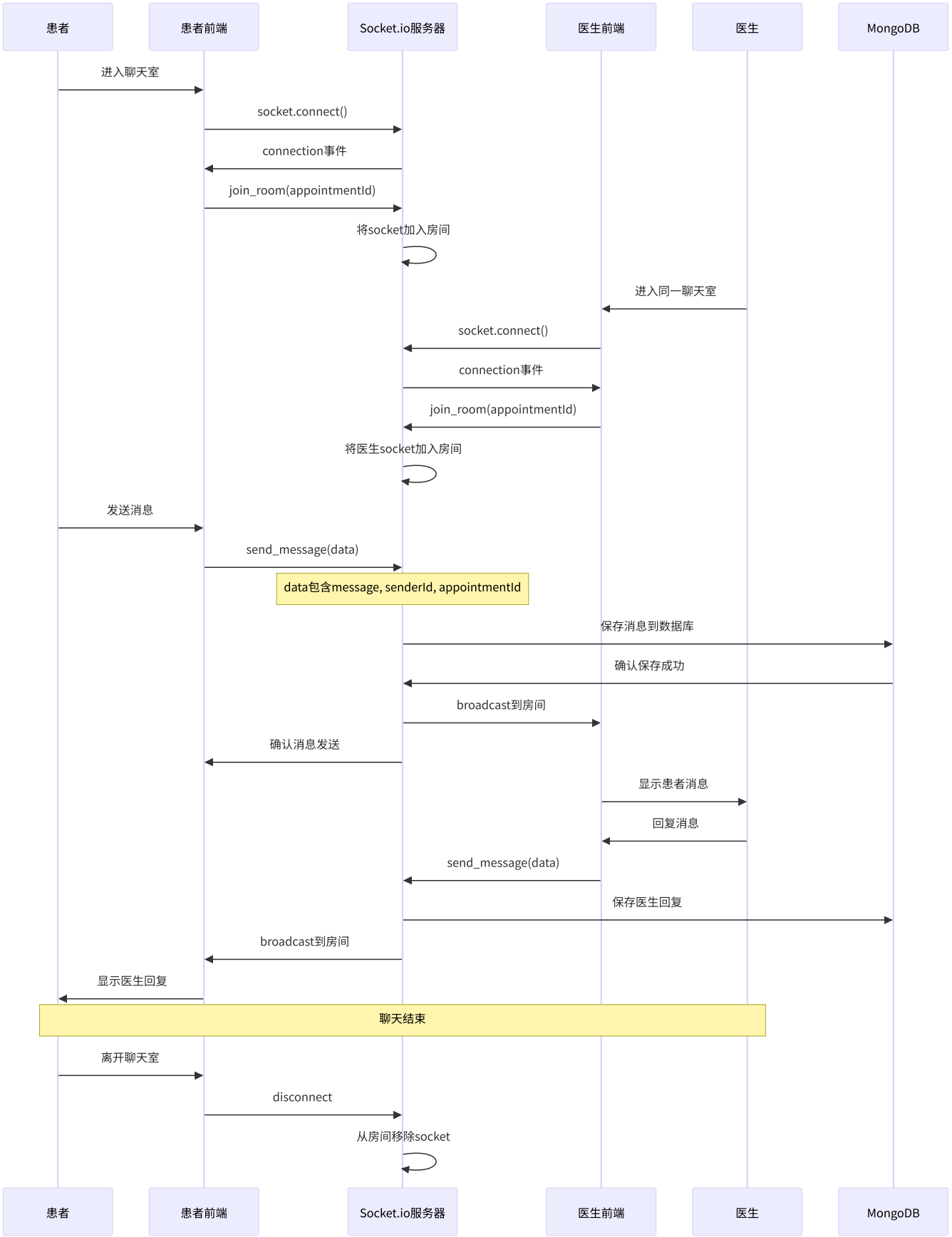
2. 医生查看预约：

```
Appointment.find({doctorId: req.user.doctorId})
  .populate('patientId', 'name phone')
  .sort({appointmentDate: 1})
```

业务规则：

- 只能预约未来时间
- 预约至少提前2小时
- 每个患者同一天只能预约同一医生一次
- 取消预约需要在预约时间前4小时

4. 实时聊天系统（基于Socket.io实现）



详细解释

Socket.io架构实现：

1. 服务器端配置：

```
const io = require('socket.io')(server, {
  cors: {
    origin: process.env.CLIENT_URL,
    methods: ["GET", "POST"]
  }
});
```

2. 房间管理机制：
   - 每个预约对应一个独立的聊天房间
   - 房间ID使用appointmentId确保唯一性
   - 只有预约相关的患者和医生可以加入对应房间

消息数据模型：

```
{
  appointmentId: ObjectId,
  senderId: ObjectId,
  senderType: String (patient/doctor),
  message: String,
  messageType: String (text/image/file),
  timestamp: Date,
  isRead: Boolean,
  metadata: Object (文件信息等)
}
```

实时通信流程：

1. 连接建立：

```
io.on('connection', (socket) => {
  socket.on('join_room', async (data) => {
    const { appointmentId, userId } = data;
    // 验证用户权限
    const appointment = await Appointment.findById(appointmentId);
    if (appointment.patientId == userId || appointment.doctorId ==
userId) {
      socket.join(appointmentId);
      socket.emit('joined_room', { success: true });
    }
  });
});
```

2. 消息发送与广播：

```
socket.on('send_message', async (data) => {
  const message = new Message(data);
  await message.save();

  io.to(data.appointmentId).emit('new_message', {
    message: message.message,
    senderId: message.senderId,
    timestamp: message.timestamp
  });
});
```

3. 消息持久化：
   - 所有消息实时保存到MongoDB
   - 支持消息历史记录查询
   - 实现消息已读状态跟踪

功能特性：

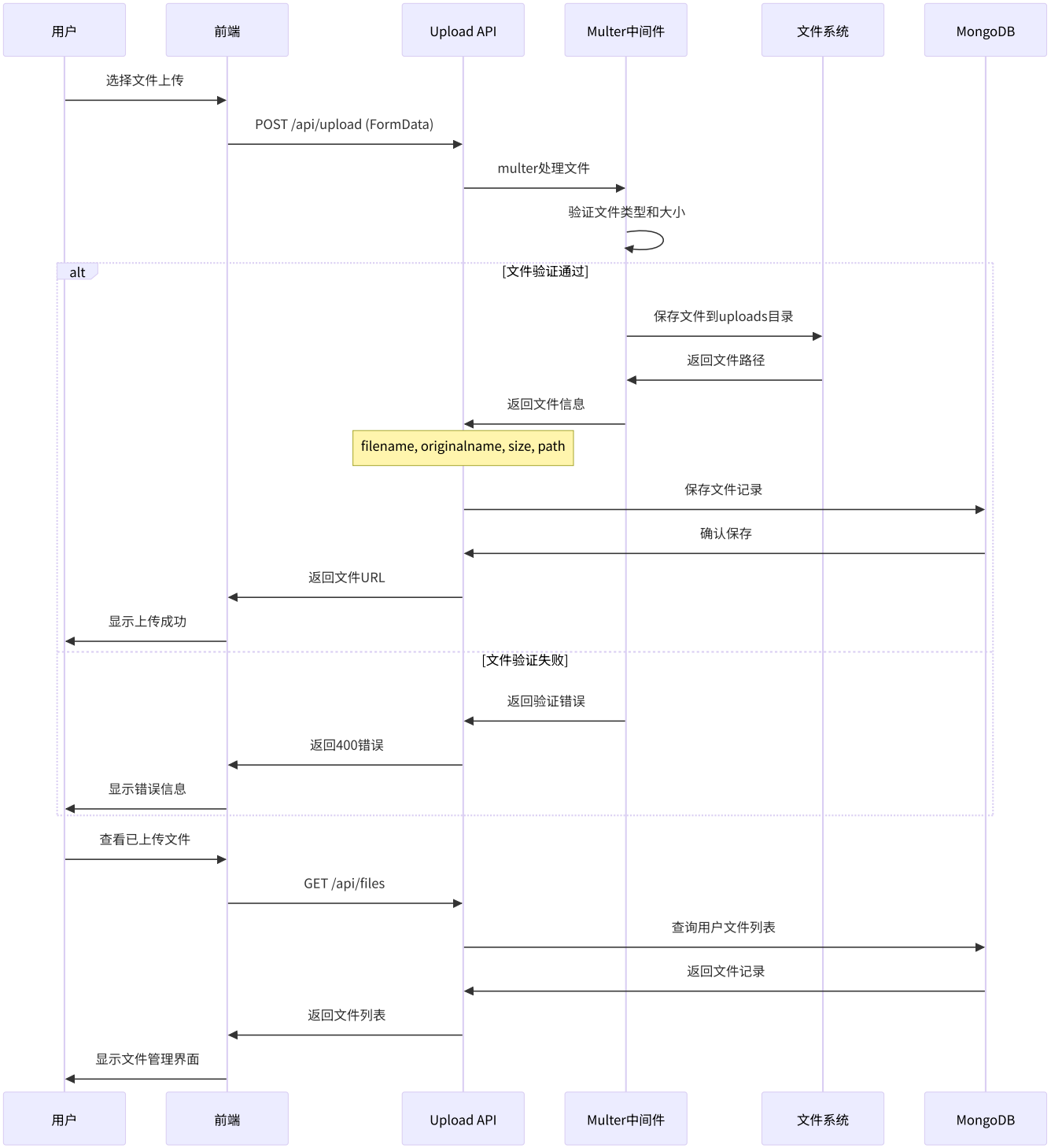
- 支持文本、图片、文件消息
- 消息状态指示（发送中、已发送、已读）
- 离线消息推送
- 聊天记录导出功能
- 敏感信息过滤

性能优化：

- 使用Redis作为Socket.io适配器支持水平扩展

- 消息分页加载减少内存占用
- 连接池管理避免资源泄露

5. 文件上传管理（基于multer和文件存储）



详细解释

Multer配置实现：

1. 存储配置：

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    const uploadPath = path.join(__dirname, '../uploads', getFileCategory(file));
    ensureDirectoryExists(uploadPath);
    cb(null, uploadPath);
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    cb(null, uniqueSuffix + path.extname(file.originalname));
  }
});
```

2. 文件过滤器：

```
const fileFilter = (req, file, cb) => {
  const allowedTypes = {
    'image': ['image/jpeg', 'image/png', 'image/gif'],
    'document': ['application/pdf', 'application/msword'],
    'medical': ['image/dicom', 'application/pdf']
  };

  if (allowedTypes[req.params.category]?.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(new Error('不支持的文件类型'), false);
  }
};
```

文件数据模型：

```
{
  filename: String (存储文件名),
  originalname: String (原始文件名),
  mimetype: String (MIME类型),
  size: Number (文件大小),
  path: String (文件路径),
  uploaderId: ObjectId (上传者ID),
  category: String (文件分类),
  isPublic: Boolean (是否公开),
  uploadDate: Date,
  lastAccessed: Date
}
```

安全机制：

1. 文件大小限制：

```
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 10 * 1024 * 1024, // 10MB
    files: 5 // 最多5个文件
  },
  fileFilter: fileFilter
});
```

2. 病毒扫描：
   - 集成ClamAV进行病毒检测
   - 可疑文件隔离处理
   - 定期安全扫描
3. 访问控制：
   - 基于用户权限的文件访问
   - 临时访问链接生成
   - 文件下载日志记录

存储策略：

1. 本地存储：
   - 开发环境使用本地文件系统
   - 按日期和类型分目录存储

- 定期清理过期文件

2. **云存储集成：**
   - 生产环境支持AWS S3/阿里云OSS
   - CDN加速文件访问
   - 自动备份和容灾

文件处理功能：

- 图片自动压缩和缩略图生成
- 文档格式转换（Word转PDF）
- 医学影像DICOM格式支持
- 文件版本管理
- 批量文件操作