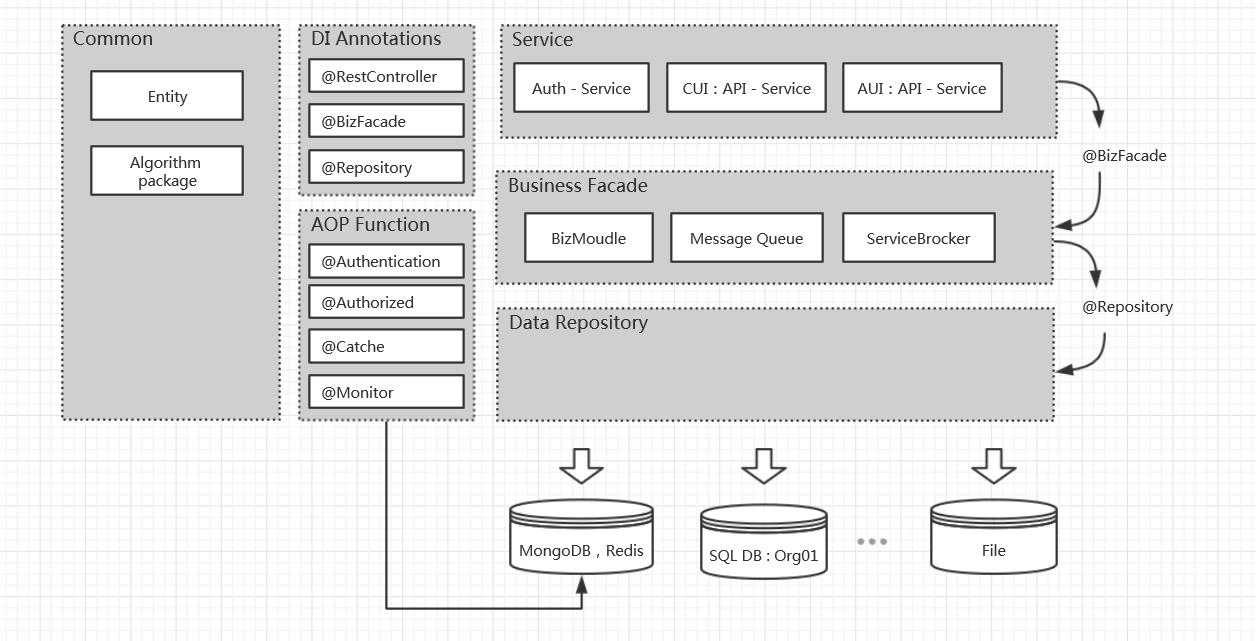
# API服务架构

服务本身纵向包含了三个层面的设计：

Service：包含Auth服务，AUI-API服务，CUI-API服务的入口。

Business Façade：整合业务逻辑，以业务外观的形式提交给上层（各个入口）调用。

Data Repository：Repository模式封装数据访问操作。



*PS：切向相关的注解，目前只实现了认证。其它的需要后续完成。*

# 约定

**[Common]：**

* 跨层的类的调用需要实现相应的接口。
* 跨层的类访问请使用依赖注入，不要使用new关键字。

*以上两条有利于封层进行单元测试测试。*

**[Controller]：完成基本的输入校验，输出格式化相关的工作**

* 对于需要授权认证的API请在Action方法上添加@Authentication注解。
* 所有的Controller请继承ApiController
* API-Service的实现类请使用@RestController注解标记。

**[Business Facade]：按照业务领域完成逻辑操作**

* Business Facade 的实现类请使用@BizFacade注解标记。
* 请为每个Business Façade预先定义接口。
* 对于不能使用常规方式注入的底层Repository请使用RepositoryFactory构造。

PS：如果实现类内部调用了老的BO，建议使用其他注解标记。以方便后面的重构。

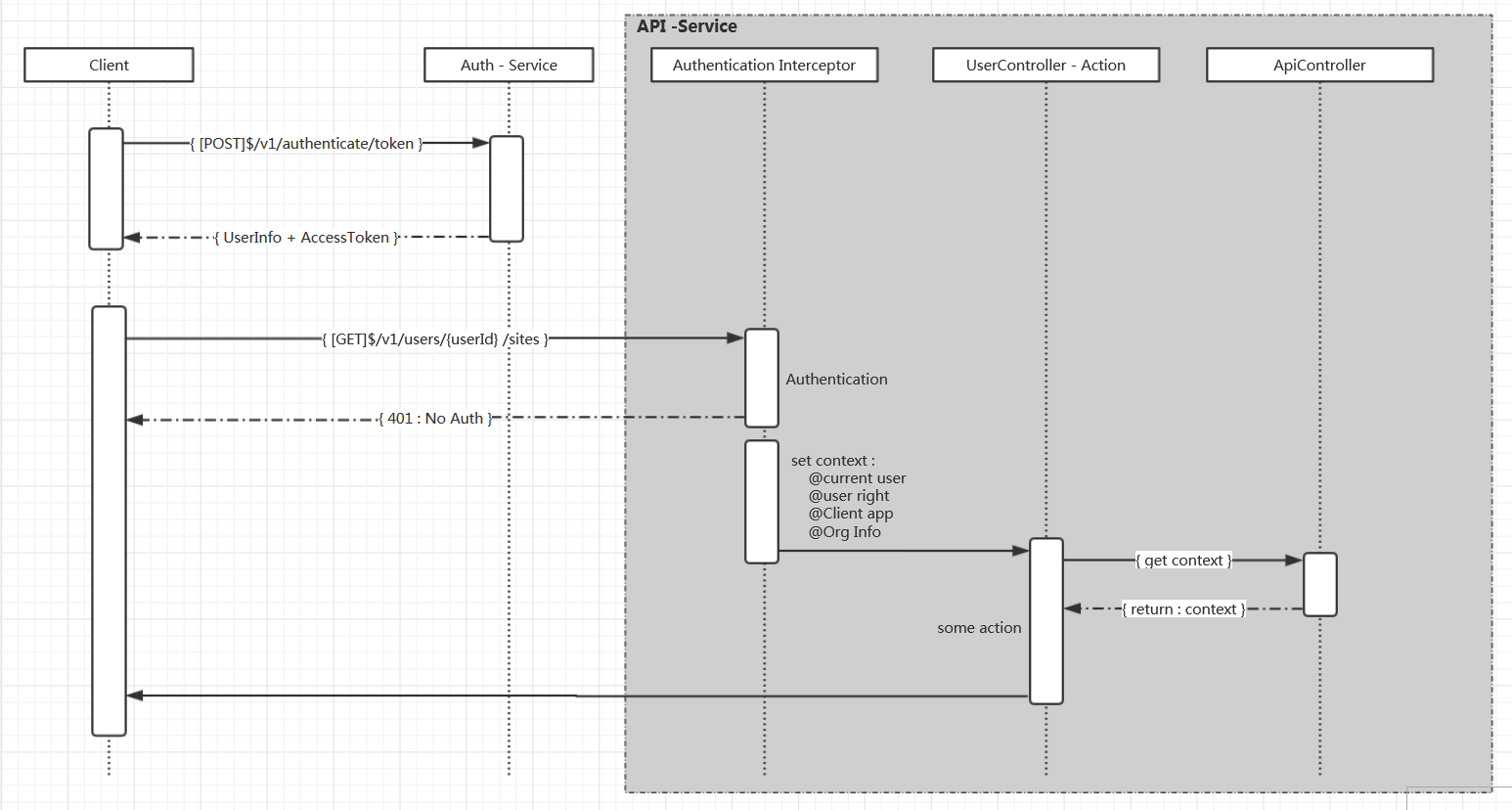
**[Data Repository]：封装数据访问操作（数据库，文件，高速缓存）**

* Data Repository 的实现类请使用@Repository注解标记。
* 请为每个Data Repository预先定义接口。
* 仓储类的具体实现类需要实现接口Repository（实现基本的操作方法）
* 仓储类如果有额外的方法需要在自身的接口内定义（通常都会有）

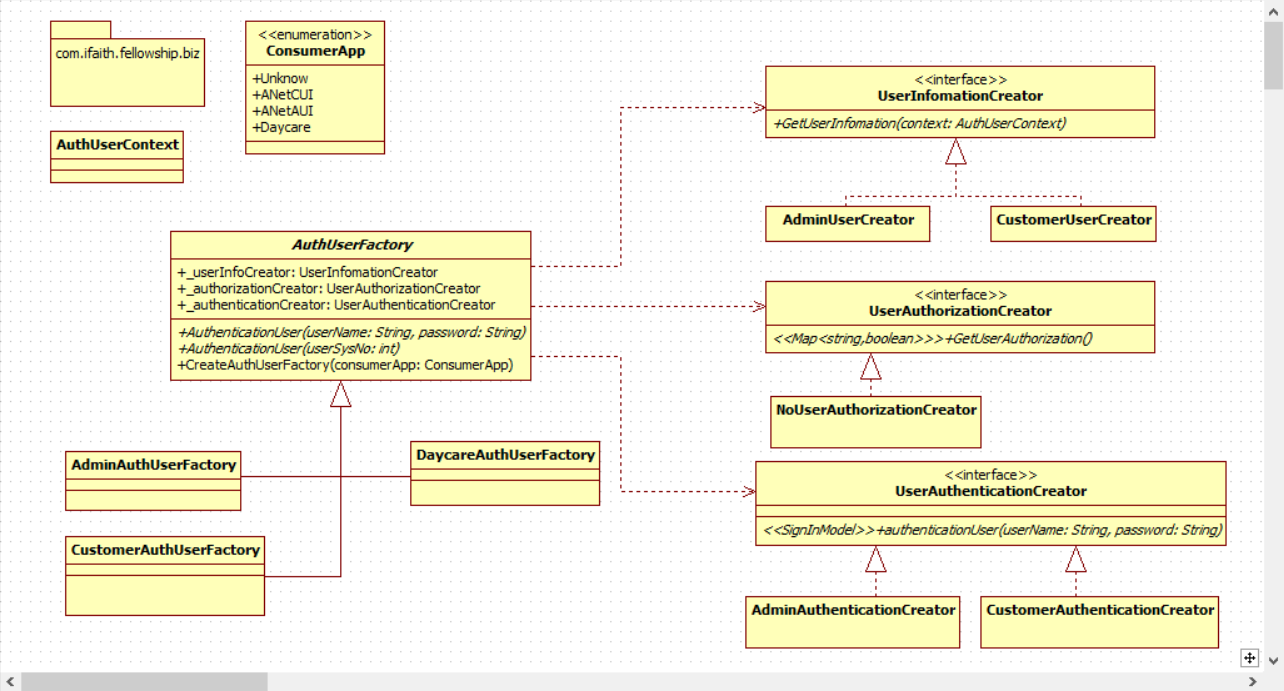
# 认证流程

需要用户认证的API，都会按照上述流程进行验证和上下文的设置。几乎所有的认证和设置对消费者（功能开发人员）都是透明的。后续可以通过缓存和高速存贮的技术提升性能。认证流程如下图所示。

另外，在拦截器的部分可以注入更多的切向相关接口实现。以方便完成后续的缓存，日志，监控等切向工程。目前，只是简单的实现了用户的认证和授权。



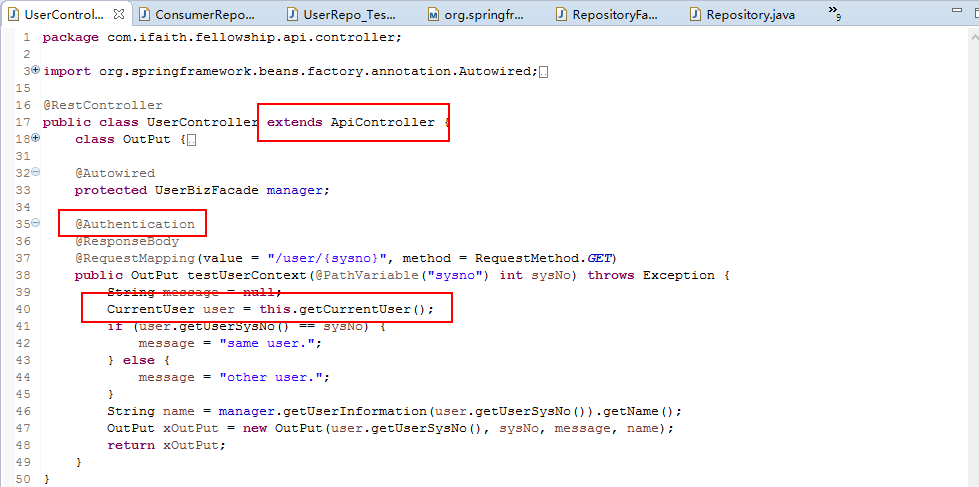
# 认证系统Design



认证系统采用抽象工厂的设计模式，独立出了用户信息获取、用户权限加载、用户权限认证三个部分的接口。AuthUserFactory的实现代表了不同的系统（每个系统都会一个对应的实现）。客户端调用认证接口时，服务会根据key和 scare构造相应Factory。每个Factory都会知道他自身的用户权限、认证方式、以及用户信息获取方式。

新的系统接入时，只需要扩展一个 AuthUserFactory的实现类（代码在20行以内）

# 客户端可用API、注解、以及接口



消费者（功能开发人员）在编写Controller时可以参照上面的Code实现：

@.继承ApiController

@.使用@Authentication注解标记需要认证才能调用的API。

@.获取当前调用者的信息this.getCurrentUser() 方法。

@.获取当前调用者的权限列表（未完成）。

@.获取当前调用者的来源（未完成）。

*PS：很多功能可以在ApiController之中进行扩展。这部分扩展不会影响开发人员原有的Code。*

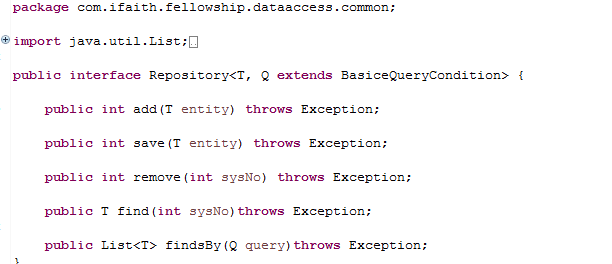
***Code 路径：***[***git@github.com:TheFaith/ifaith.git***](mailto:git@github.com:TheFaith/ifaith.git)

***url ：***[***https://github.com/TheFaith/ifaith***](https://github.com/TheFaith/ifaith)

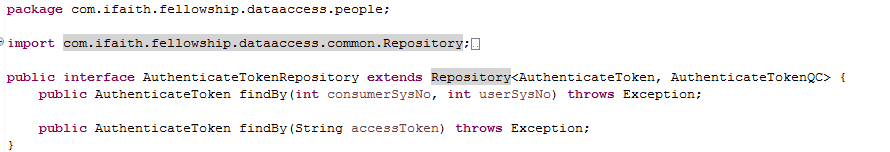
以下是按照之前的约定实现的部分示例代码，仅供参考：

*PS : 关于异常 的管理后续在实际的项目中再进行分类细化*

Repository 接口定义了基本操作：



实际的业务中建议根据实际Repository的需求继承并添加接口方法，如下所示。而业务层的Code以依赖注入的方式得到自定义Repository接口的实现：



而在业务层的使用如下所示：

