|  |
| --- |
| Technical Note 687  Issued: |
| Design and implementation of Integrated Health Mangement Cardiac Care Solution |
|  |
| QiZhong Lin; Zhu Jane; Bu Jonah; Chen Jasmin; Mosis Georgio  Philips Research China |
| Philips Restricted  © Koninklijke Philips N.V. 5 |

|  |  |  |
| --- | --- | --- |
| QiZhong Lin |  | QiZhong.LIN@philips.com |
| Zhu Jane |  | Jane.ZHU@philips.com |
| Bu Jonah |  | Jonah.BU@philips.com |
| Chen Jasmin |  | Jasmin.CHEN@philips.com |
| Mosis Georgio |  | Georgio.MOSIS@philips.com |
|  |  |  |

Authors’ address

© KONINKLIJKE PHILIPS NV   
All rights reserved. Reproduction or dissemination in whole or in part is prohibited without the prior written consent of the copyright holder .

|  |  |
| --- | --- |
| Title: | Design and Implementation of Integrated Health Management Cardiac Care Solution |
|  |  |
| Author(s): | QiZhong Lin; Zhu Jane; Bu Jonah; Chen Jasmin; Mosis Georgio |
|  |  |
| Reviewer(s): |  |
|  |  |
| Technical Note: | 687 |
|  |  |
| Additional Numbers: |  |
|  |  |
| Subcategory: |  |
|  |  |
| Project: | Integrated Health Management Cardiac Care Solutions (2013-0068) |
|  |  |
| Customer: |  |

|  |  |
| --- | --- |
| Keywords: | Cardiac, software architecture, micro-service, spring boot, Android, RESTful API |
|  |  |
| Abstract: | There are some key benefits of connection between physician, patient and CHC. Patient gets continual care with personalized intervention tool, physician gets a better overview of patients’ status for better quality of care, meawhile physician collects more data for clinical research, the resource of CHC can be fully leveraged to release overburnden of level 3 hospital.  The project focus on integrated health management solution which targets to cardiac care and this technical note describes in detail software design and implementation including the back-end server and android native app. |
|  |  |

|  |  |
| --- | --- |
| Conclusions: | The system can be deployed inside hospital, or public cloud. New features can be added like retrohab, CDS and data analysis. This mobile healthcare system can be sold to hospital as standalone application or bound to Philips CDR system |
|  |  |

Management Summary

One-page summary for Management.

Currently in China there are some issues related to heathcare. For example, the healthcare system is not optizimed like over burdened higher level hospitals and no GP system in place. Clinical and technology focus on treating health issues so it is difficult to monitor patients over time. The relationship between patients and physicians is poor. Even though Healthcare system is under pressure but change is initiated. In future primary care system will spring up and hospitals will be vertically integrated. Clinical and technology will focus on early detection, data integration and analysis. There is why government policies focus on prevention, evidence-based medicine, and healthy cities.

The project aims to integrate health management solution, which currently targets to Cardiac Care. The workflow is as follows:

* cardiologist writes discharge plan via his tablet, nurse educates patient via patient’s tablet during the discharge meeting.
* Patient measures blood pressure/heart rate daily and records symptoms when they occur via his tablet
* Patient brings his tablet to the CHC during follow-up meetings, the GP writes comments to the treatment plan if he makes a change
* Patient brings his tablet to the hospital during follow-up meetings, the physician or nurse updates the treatment
* Physician reviews all patient’s data and does research analysis via his tablet anytime

The solution adds benefits to all related people. Patient owns more information from and contact with their specialist. Patient’s family understands their parents’ current disease status easier. Cardiologist can collect data to get a better overview of patients’ status for better quality of care from patient at home and other care providers, in this case they can do clinical research. GP in CHC have more training and knowledge from L3 specialists and share workload from overburdened level 3 hospital.

Considering potential business models, the solution can be deployed and sold to hospitals as standalone. As platform the solution is scalable, more features can be added like clinical decision support, retrohab and data analysis etc.

Contents

[Design and implementation of Integrated Health Mangement Cardiac Care Solution i](#_Toc436911991)

[1. Introduction 7](#_Toc436911992)

[1.1. Motivations 7](#_Toc436911993)

[1.2. Scenario 7](#_Toc436911994)

[2. Solution approach 11](#_Toc436911995)

[2.1. Workflow 11](#_Toc436911996)

[2.2. Deployment architecture 11](#_Toc436911997)

[2.3. software architecture 13](#_Toc436911998)

[3. Software implementation 15](#_Toc436911999)

[3.1. Back-end 15](#_Toc436912000)

[3.1.1. Micro-service 15](#_Toc436912001)

[3.1.2. Authentification and Authorization service 16](#_Toc436912002)

[3.1.3. Static resource upload and download 17](#_Toc436912003)

[3.1.4. Spring Data – MongDB 18](#_Toc436912004)

[3.2. Front-end 19](#_Toc436912005)

[3.2.1. UI wireframe 19](#_Toc436912006)

[3.2.2. Physician portal 19](#_Toc436912007)

[3.2.3. Patient portal 21](#_Toc436912008)

[3.3. Data Model 23](#_Toc436912009)

[3.4. Activity Diagram of Patient portal 24](#_Toc436912010)

[3.5. Offline support 25](#_Toc436912011)

[3.6. Data sync 26](#_Toc436912012)

[4. Acknowledgements 27](#_Toc436912013)

[5. Appendix I - Micro service 28](#_Toc436912014)

[5.1. Why micro-service compared to monolith application? 28](#_Toc436912015)

[5.2. How to implement micro-service? 29](#_Toc436912016)

[6. Appendix II – Security 32](#_Toc436912017)

[6.1. Authentification and Authroization Service 32](#_Toc436912018)

[6.2. HTTPS 38](#_Toc436912019)

[7. Appendix III - RESTful request of android app 41](#_Toc436912020)

[References 43](#_Toc436912021)

# Introduction

In this section, first the motivations of this project is introduced, then based on the motivation, the general scenario is shown.

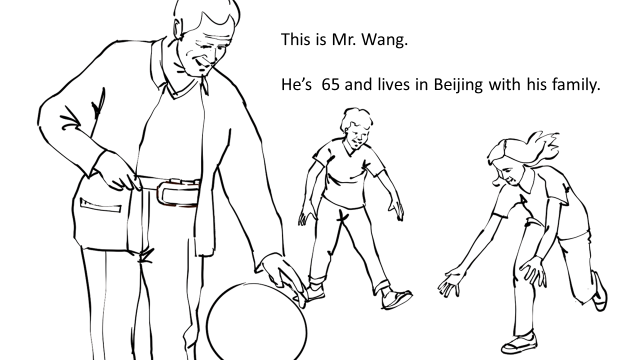
## Motivations

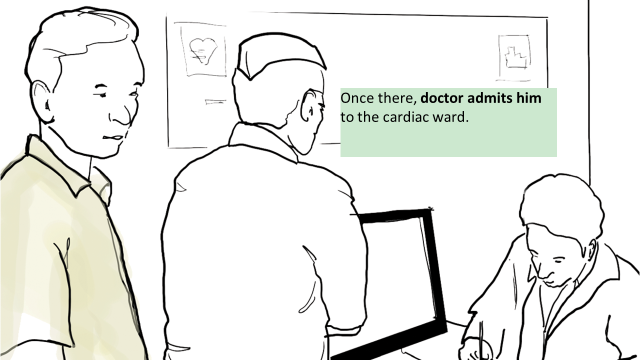
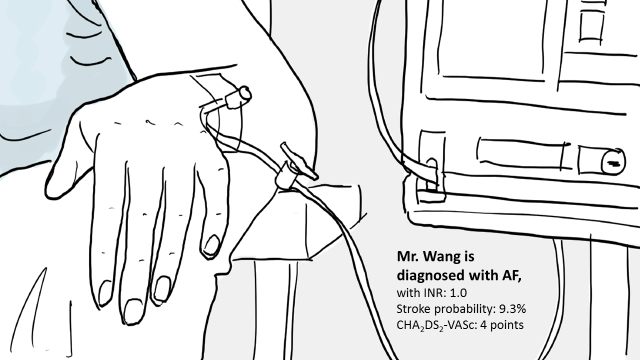
After interview with many patients, physicians and GP in CHC, general voices are collected and summarized as follows:

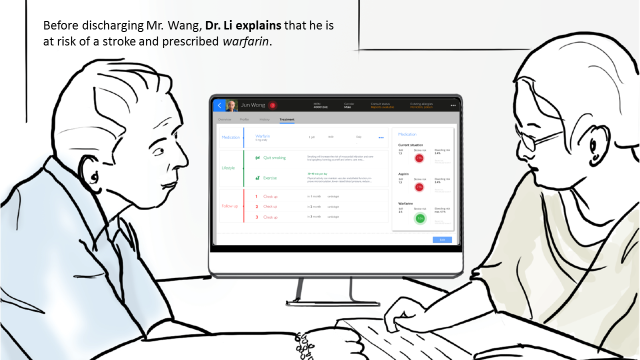
* patient: I want more information from and contact with my specialist, sometimes I forget or don’t comply with treatment as prescribed.
* patient’s family: it is difficult to understand my parents’ current disease status.
* Cardiologist: collect data to get a better overview of patients’ status for better quality of care from patient at home and other care providers; want to collect data for clinical research; want to know if patients follow their treatment as prescribed; check patients’ status after an intervention.
* GP in CHC: require more trust from patients for our medical knowledge; want to have more training and knowledge from L3 specialists.

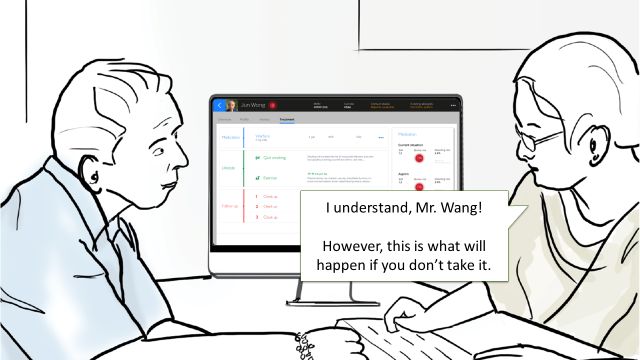
Based on the above requirement, the integrated health management solution is designed and developed to connect patient at home, physician at level 3 hospital and GP in CHC. The following section will show you the scenario as a clear use story.

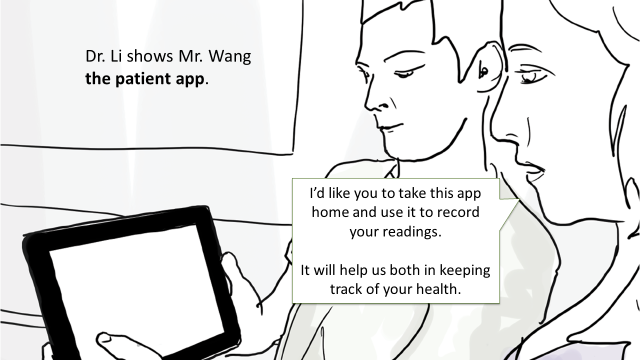
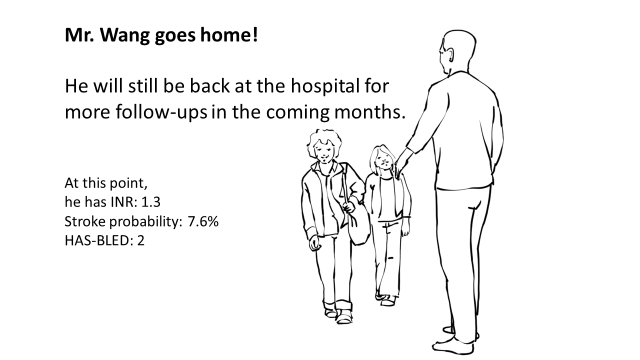
## Scenario

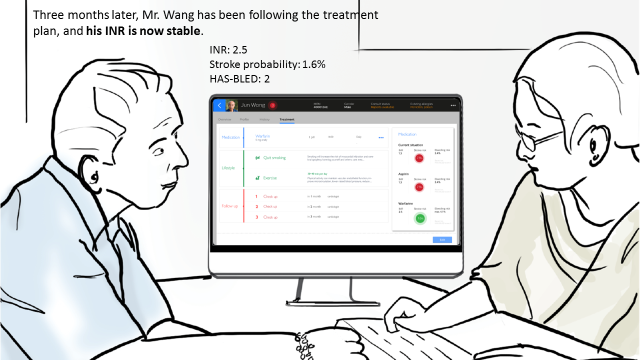
 

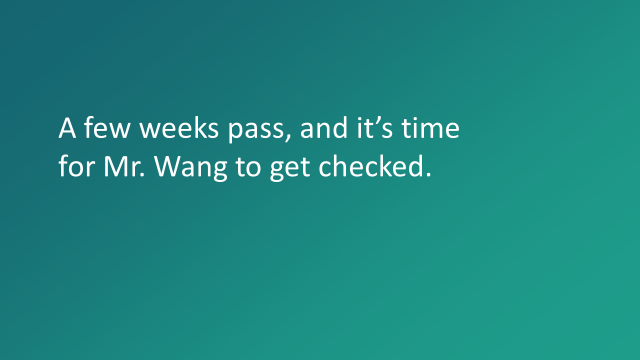
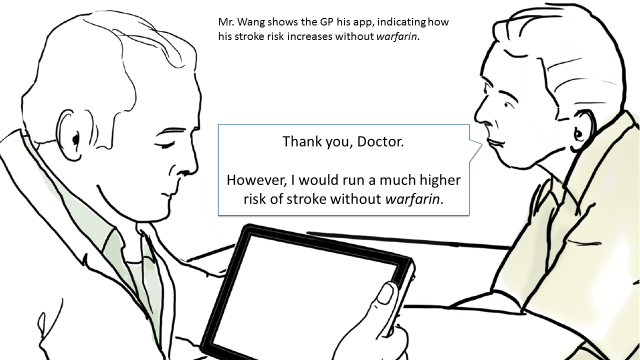
 

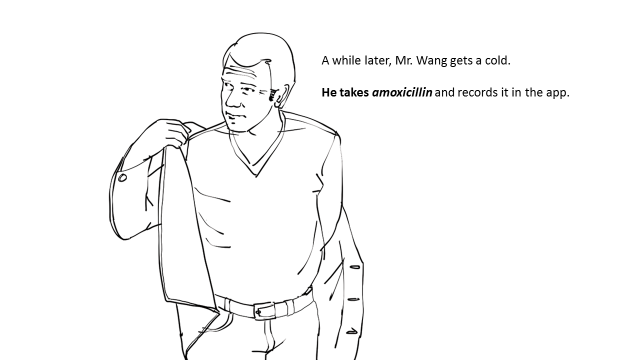
 

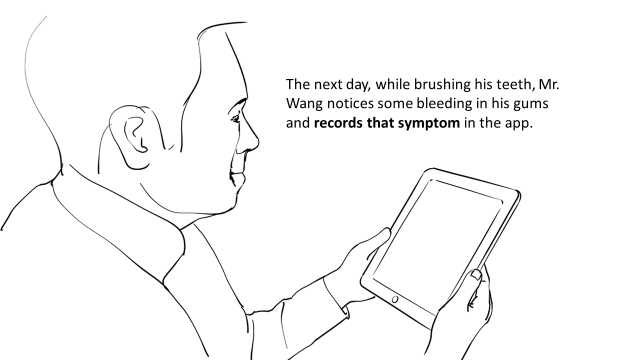
 

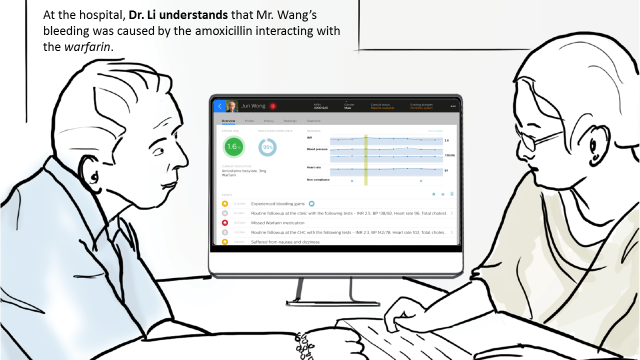
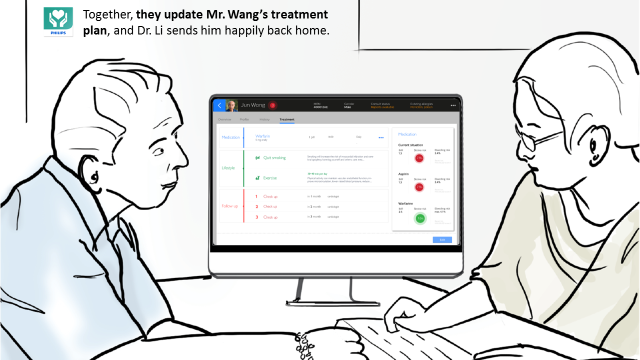
 







# Solution approach

## Workflow

Taking Cardiac care for consideration, the scenario is simplifized and concreted as shown in figure 1

* cardiologist creates discharge plan via his tablet, nurse educates patient via patient’s tablet during the discharge meeting.
* Patient measures blood pressure/heart rate daily and records symptoms when they occur via his tablet
* Patient brings his tablet to the CHC during follow-up meetings, the GP writes comments to the treatment plan if he makes a change
* Patient brings his tablet to the hospital during follow-up meetings, the physician or nurse updates the treatment
* Physician reviews all patient’s data and does research analysis via his tablet anytime



* Cardiologist creates treatment plan
* Patient gets education
* Patient reviews plan
* Patient gets medication reminders
* Patients measures BP / HR
* Patient records symptoms



* Physician reviews and updates
* GP reviews treatment plan
* GP notes medication changes

Figure 1. The workflow of Integrated Health Management Cardiac Care

* GP reviews treatment plan
* GP notes medication changes
* Patient reviews plan
* Patient gets medication reminders
* Patients measures BP / HR
* Patient records symptoms
* Physician reviews and updates

## Deployment architecture

As shown in figure 2, cardiology CDR is Philips BIU’s product, Ganges inside blue box is Data integration server between CDR and Ganges Cloud (Ali), both CDR and Ganges inside blue box are deployed inside hospital via intranet. Ganges Cloud (Ali) is our system back-end server, deployed in public via internet. The way of communication between Inside hospital with ourside via hospital VPN. Both physician’s tablet and patient’s tablet are connected to our system back-end server.

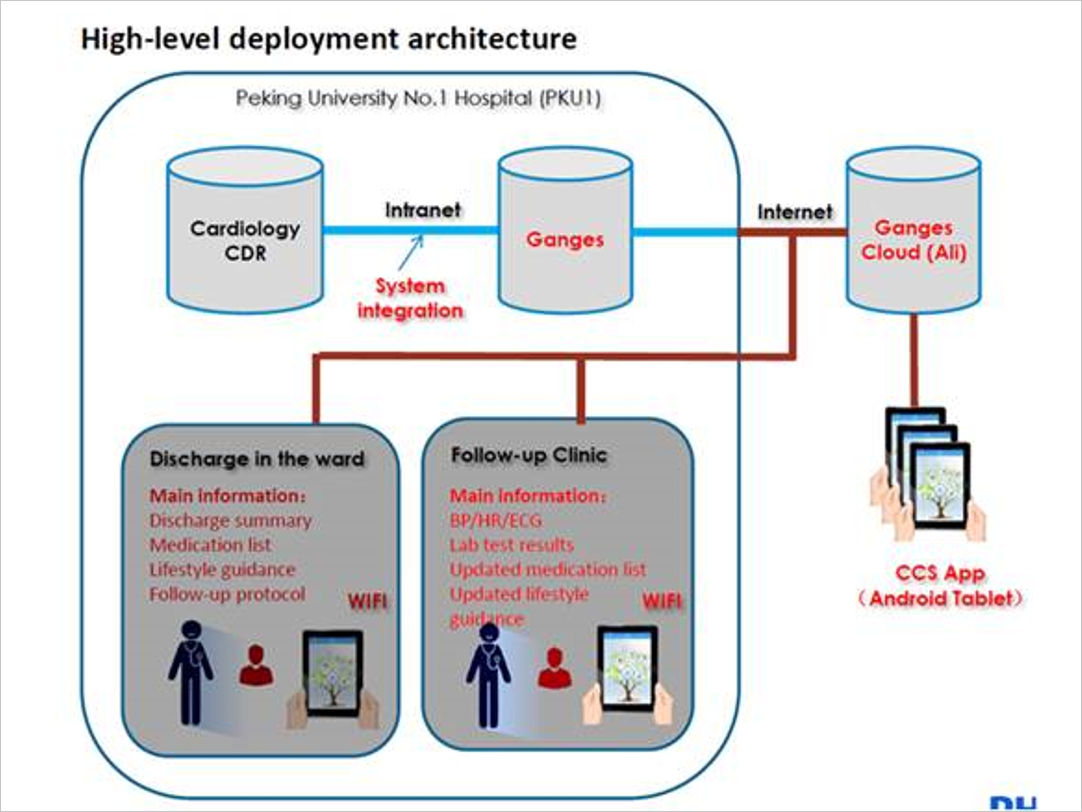


Figure 2. the deployment architecture of integrated health management cardiac care

## software architecture

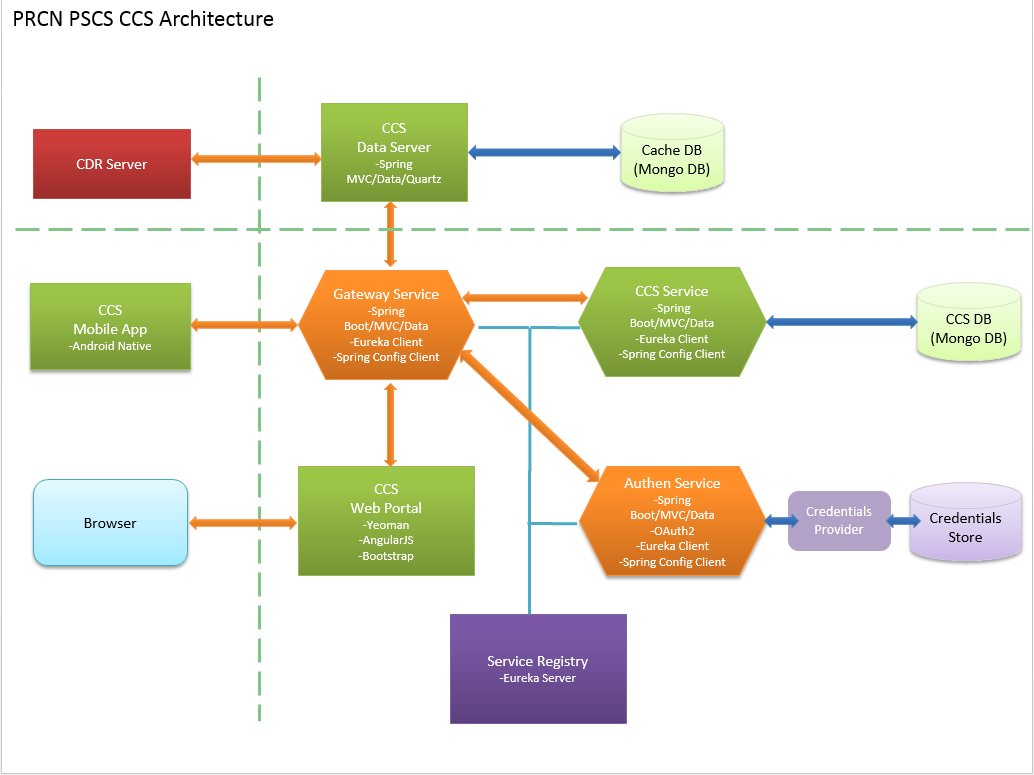


Figure 3. the software architecture of of integrated health management cardiac care

As shown in Figure 3, our back-end server is integrated with CDR server(Cardiac Repository) via CCS data Server, so all patients data are populated into CCS DB(database) via Gateway service and CCS service.

When the physician logins in physician portal, which is same app as patient portal, views all patients data from CCS DB, and then creates patient account for clinical trial. Created patient contains discharge information, medication plan, medication alert plan, lifestyle and education etc…all these data are submitted into CCS DB.

During discharge time, the patient receives the tablet and accepts the training of how to use app. Then at home every day the patient receives the medication reminder for taking medicine, inputs the measurement like blood pressure and heart rate, and records the symptom etc. all data are synced with CCS DB if there is internet connection.

The physician logins in physician portal to track patient’s record.

Current version only Android native application as front-end is supported for physician and patient, in the next version, web browser portal will be added for physician.

The software architecture utilizes the latest technology like micro-service, Spring boot framework and Spring data etc. In the rest of this tech note, each point is described in detail.

A protected resource like services (for CRUD patient information), or any static resource requiring access restrictions, is a resource stored on (or provided by) the resource server (like api-gateway) which requires authentication in order to access it. Protected resources are owned or controlled by the resource owner. Anyone requesting access to a protected resource must be authorized to do so by the resource owner (enforced by the resource server). Spring security is applied to authentification and authoritification process.

To protect against eavesdropping and man-in-the-middle attacks, HTTPS is applied to replace HTTP.

# Software implementation

## Back-end

### Micro-service

To easier maintain and extend the service, micro-service architecture(more information is referred to Appendix I) is applied to decompose the application, like Authen Service for general Authentification and Authority and CCS Service for current project function. In next year, Retrohab service will be added too.

Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, as described above, a client needs pass the Authentication service then communicate with CCS service, so gateway server is the single entry point of API for all clients. API gateway may simply proxied/routed requests to the appropriate service, or may handle other requests by fanning out to multiple services.

But how does the client of a service like api-gateway discover the location of a service instance like CCS Service If the number of instances of a service and their locations changes dynamically ? here Eureka[1] is applied, **Service Registry is Eureka Server, all eureka client including Authen service, CCS service and even API gateway service should register to Service registry, each eureka client find each other via eureka server**. Appendix I show you more detail about Eureka.

Figure 4 shows the micro-service of back-end server. Gateway Service contains GatewayApplication, GatewayController and IHMSController. Authen Service contains AuthorizationApplication and AuthorizationController. CCS service contains PatientController. Via declarative program @EnableEurekaClient and configuration file application.yml, three services are registered and connected together. The simple information flow is as follows: httprequest is received by GatewayController or IHMSController inside Gateway service, firstly forwarded to AuthorizationController inside Authentication service for Authentification and Authorification, then forwarded to PatientController inside CcsApplication for querying resource. Detailed authentication and authorification process is referred to the section of Authentification and Authorization service

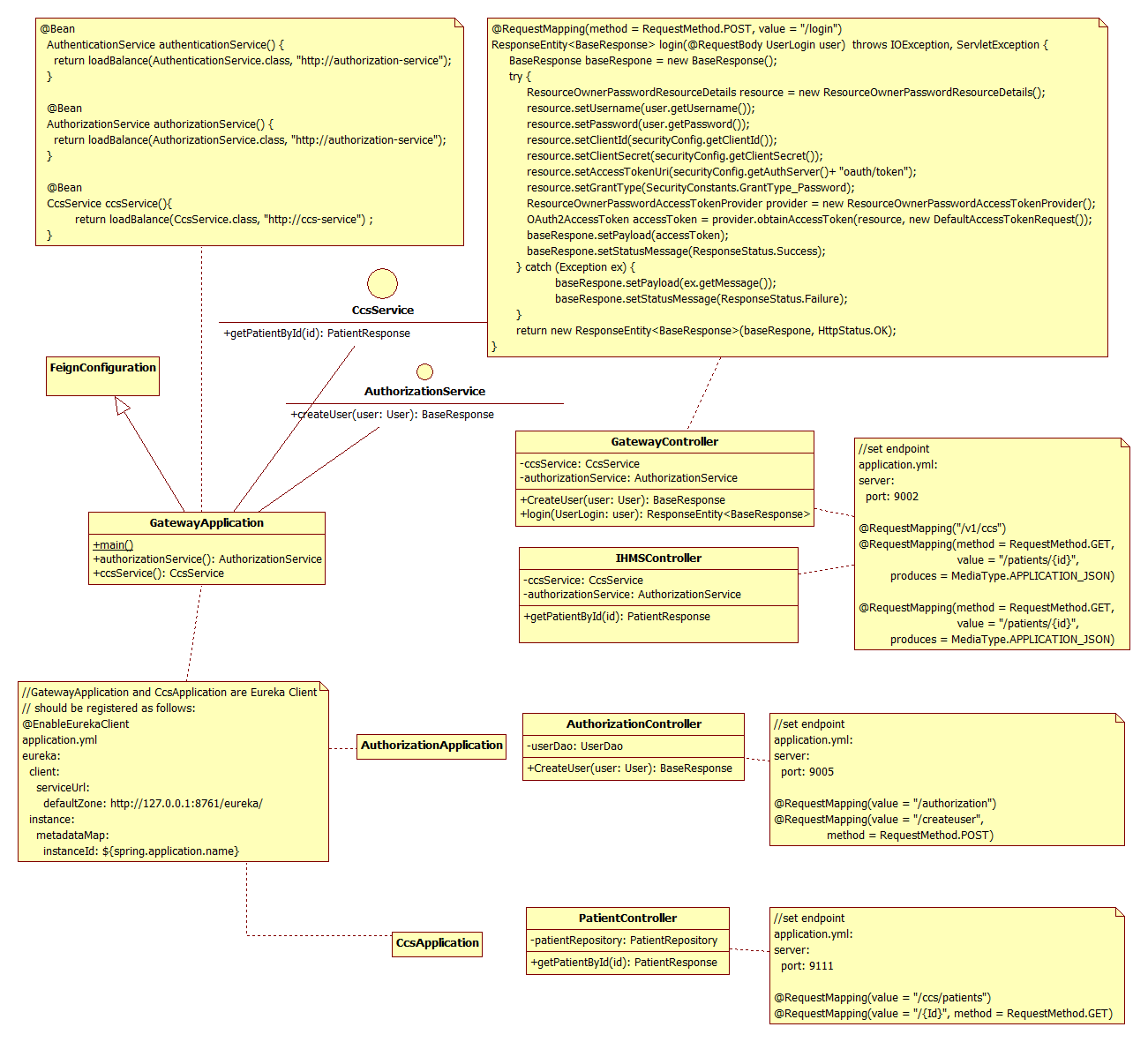


Figure 4. micro-service of back-end server

### Authentification and Authorization service

Authentification and authorization process is shown in figure 5

In physician portal, physician launchs RESTful end-point CreateUser() to create patient account and save it into Authorization MongoDB, inside the info flow, the password is encrypted, and communication between front-end and back-end should be encrypted too via SSL

In patient portal, patient launchs RESTful end-point login() to login patient account with username and password. Inside GatewayController calls obtainAccessToken of ResourceOwnerPasswordAccessTokenProvider with ResourceOwnerPasswordResourceDetails which is populated with credential info, by indirectly Authorization server will validate the credential info, if valid, access-token is obtained.

Later in patient portal, every RESTful end-point operation is launched with http header “Authorization: bear access-token”, GatewayController calls Authorization server to authenticate the access-token, if valid, GatewayController forward httprequest to CCS server.

More basic information about Authentification and Authorization is referred to Appendix II

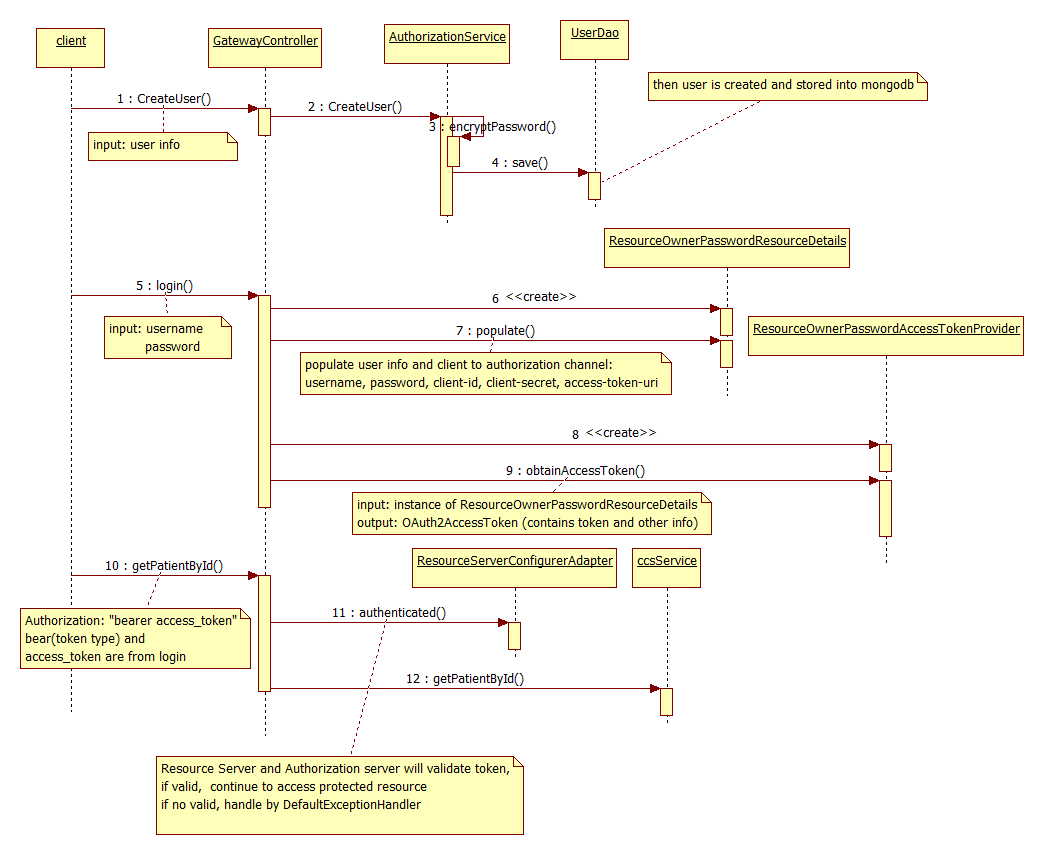


Figure 5. the sequence diagram of Authentification and Authorization service

### Static resource upload and download

Spring MVC considers static resource upload and download in decent way, just inside WebMvcConfigurerAdapter, http end-point is mapped to resource location of file system

application.yml of api-gateway

resourceServer:

resourceLocation: <file:///c:/data/resources/>

locationPrefix: file:///

WebConfig.java of api-gateway

@Configuration

@EnableWebMvc

public class WebConfig extends WebMvcConfigurerAdapter {

@Autowired

private ResourceServerConfig resourceConfig;

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*").addResourceLocations(

resourceConfig.getResourceLocation());

}

}

### Spring Data – MongDB

MongoDb[2] is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data.

Spring Data MongoDB[3] brings MongoDB to Spring applications in three ways:

* Annotations for object-to-document mapping
* Template-based database access with MongoTemplate
* Automatic runtime repository generation

In our back-server, spring data MongoDB is applied

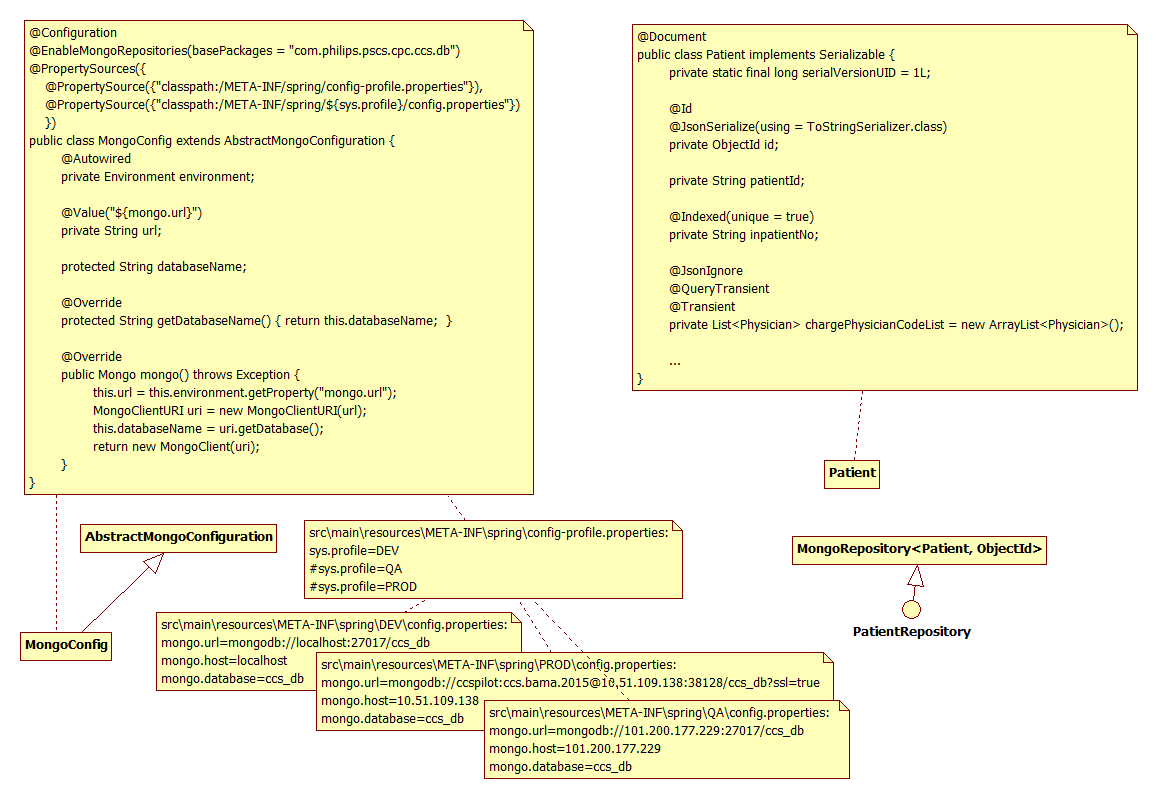


Figure 6. the diagram of spring data -mongodb

MongoConfig set configuration like MongoDB ip, port and database name. if database name does not exist, Spring MongoDB will automatically create one.

Patient is document like entity of JPA, will be mapped into collection of database ccs\_db.Patient

PatientRepository extends MongoRepository, it transitively extends the Repository marker interface. Any interface that extends Reposiotry will have an implementation automatically generated at runtime. PatientRepository will be implemented to read and write data to a MongoDB document PatientRepository interface has two parameters. The first is the type of @Document-annotated object that this repository deals with. The second is the type of the @Id-annotated property.

There are 3 common ways of defining query methods:

* Spring Data automatically generate 18 convenient methods [3]
* Spring Data supports a method-naming convention that helps Spring Data to automatically generate implementations for the methods that follow that convention.

For example, nothing else needs to be done to implement API: Patient findByPatientId(String patientId). When creating the repository implementation, Spring Data will examine any methods in the repository interface, parse the method name, and attempt to understand the method’s purpose in the context of the persisted object. In essence, Spring Data defines a sort of miniature domain-specific language (DSL) where persistence details are expressed in repository method signatures.

* Declare custom queries

You can use the @Query annotation to provide Spring Data with the query that should be performed. You still don’t write the implementation of the findPatientByPhilipsMail() method. You only give the query, hinting to Spring Data about how it should implement the method.

**public** **interface** PatientRepository **extends** MongoRepository<Patient, ObjectId>{

Patient findByInpatientNo(String inpatientNo);

@Query(“select p from Patient p where p.email like ‘%philips.com’ “)

List<Patient> findPatientsByPhilipsMail() ;

}

## Front-end

### UI wireframe

To intuitively understand how to use physician/patient portal, UI wirefireme is shown as follows.

### Physician portal

Page 1. Login

Page 2. Task - physician select discharge and followup task

Page 3. Patient list – physician include/exclude patient account into trial

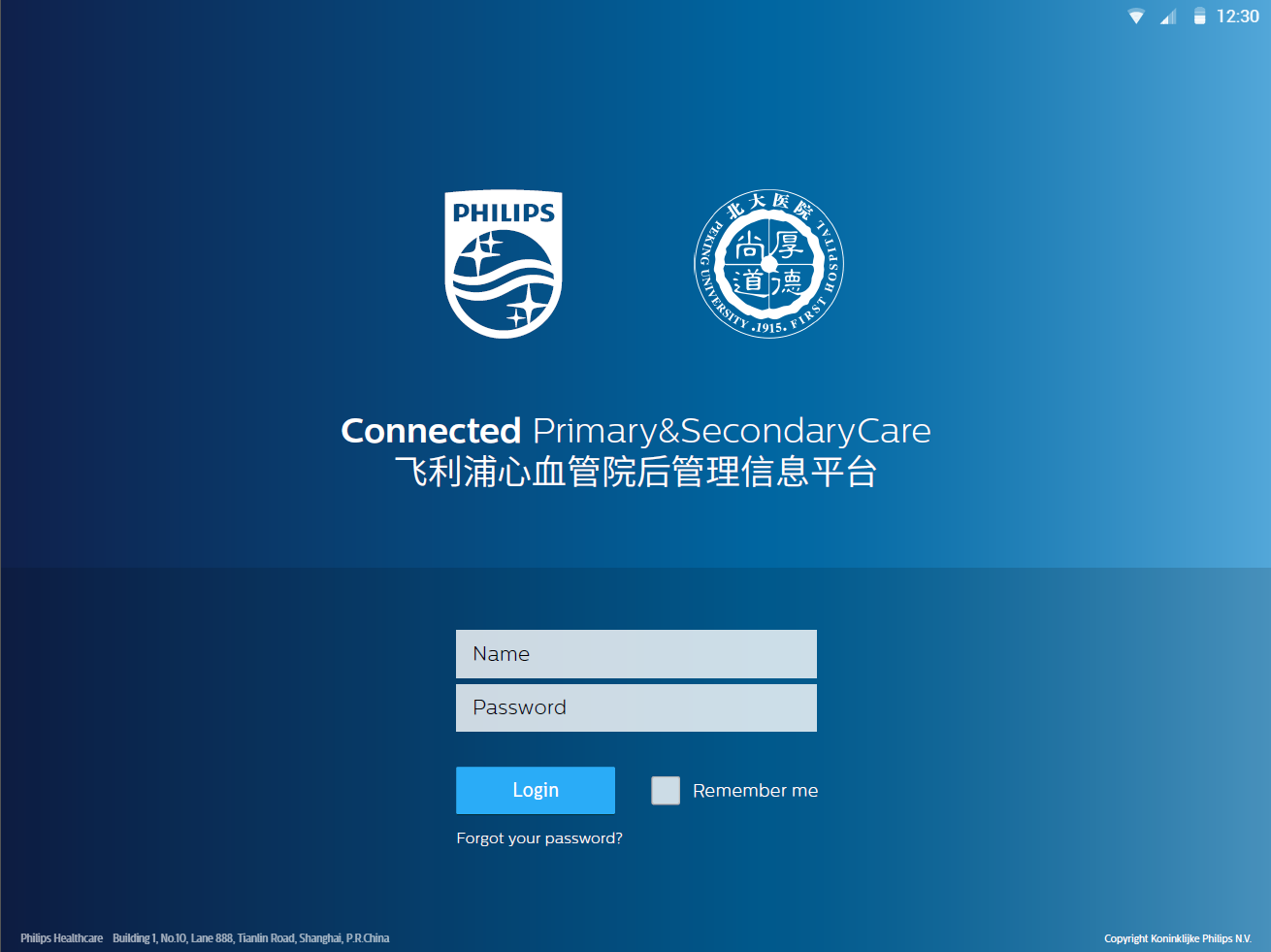
Page 4. Basic info of patient

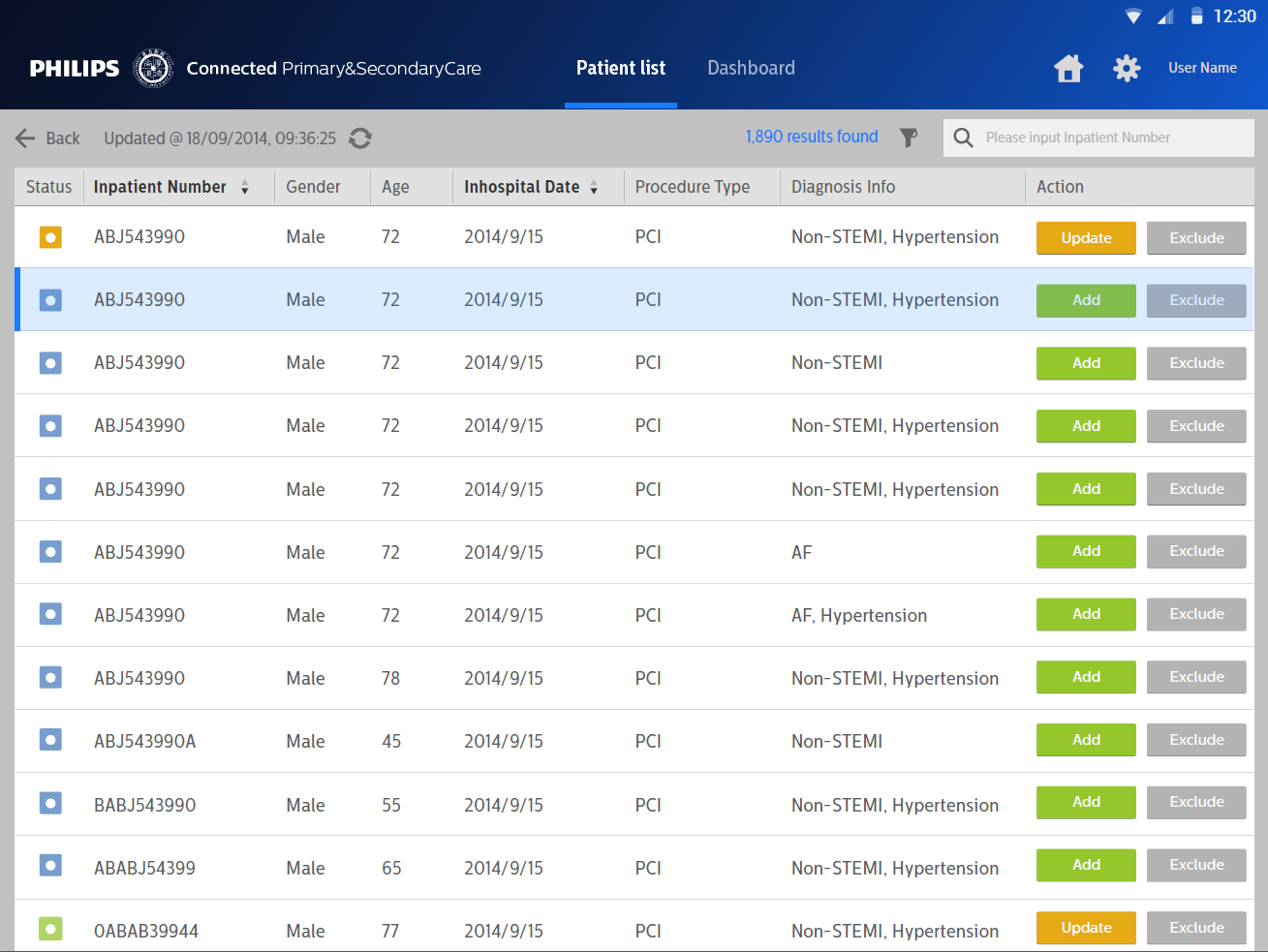
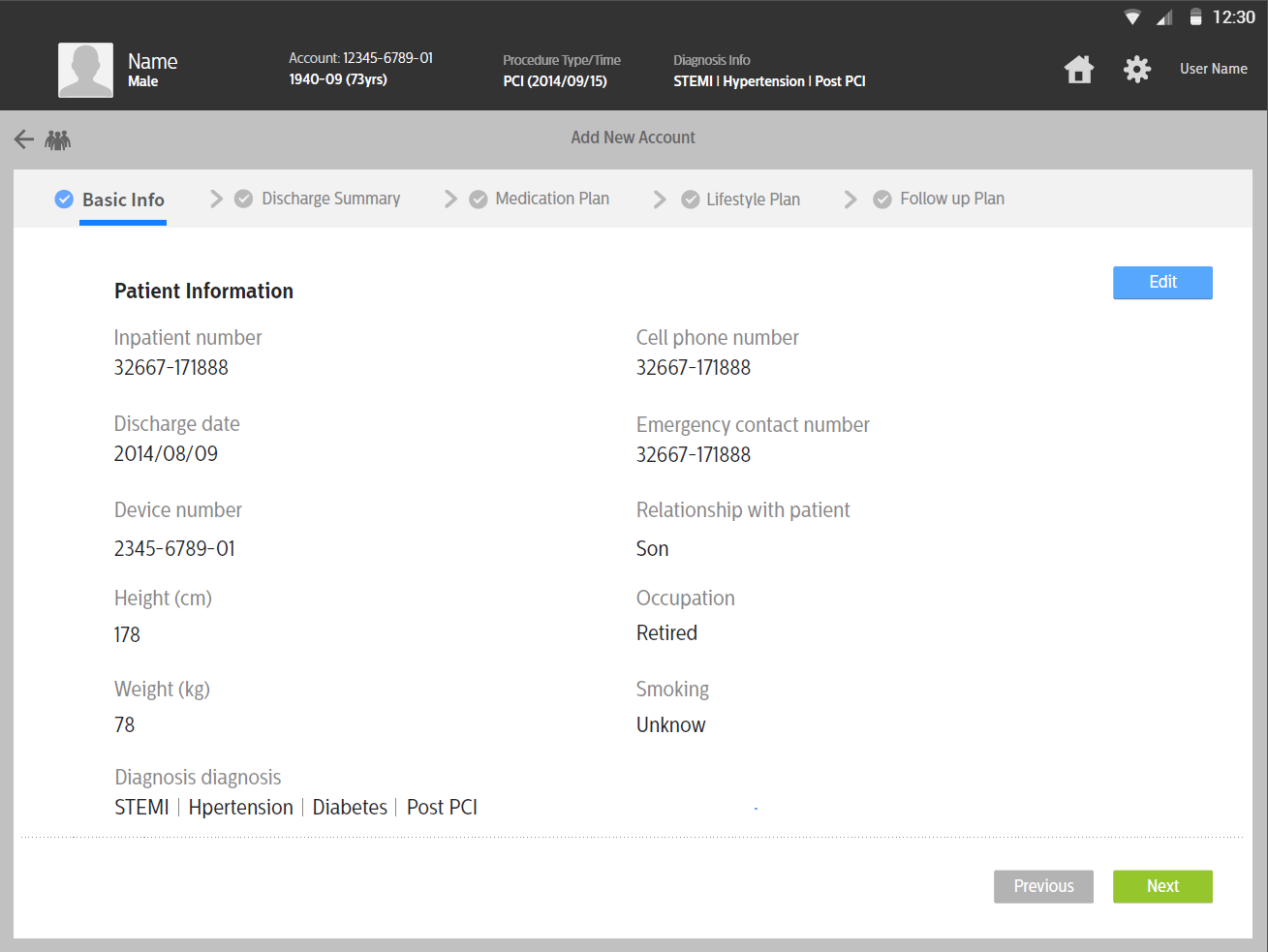
Page 5. Discharge summary of patient

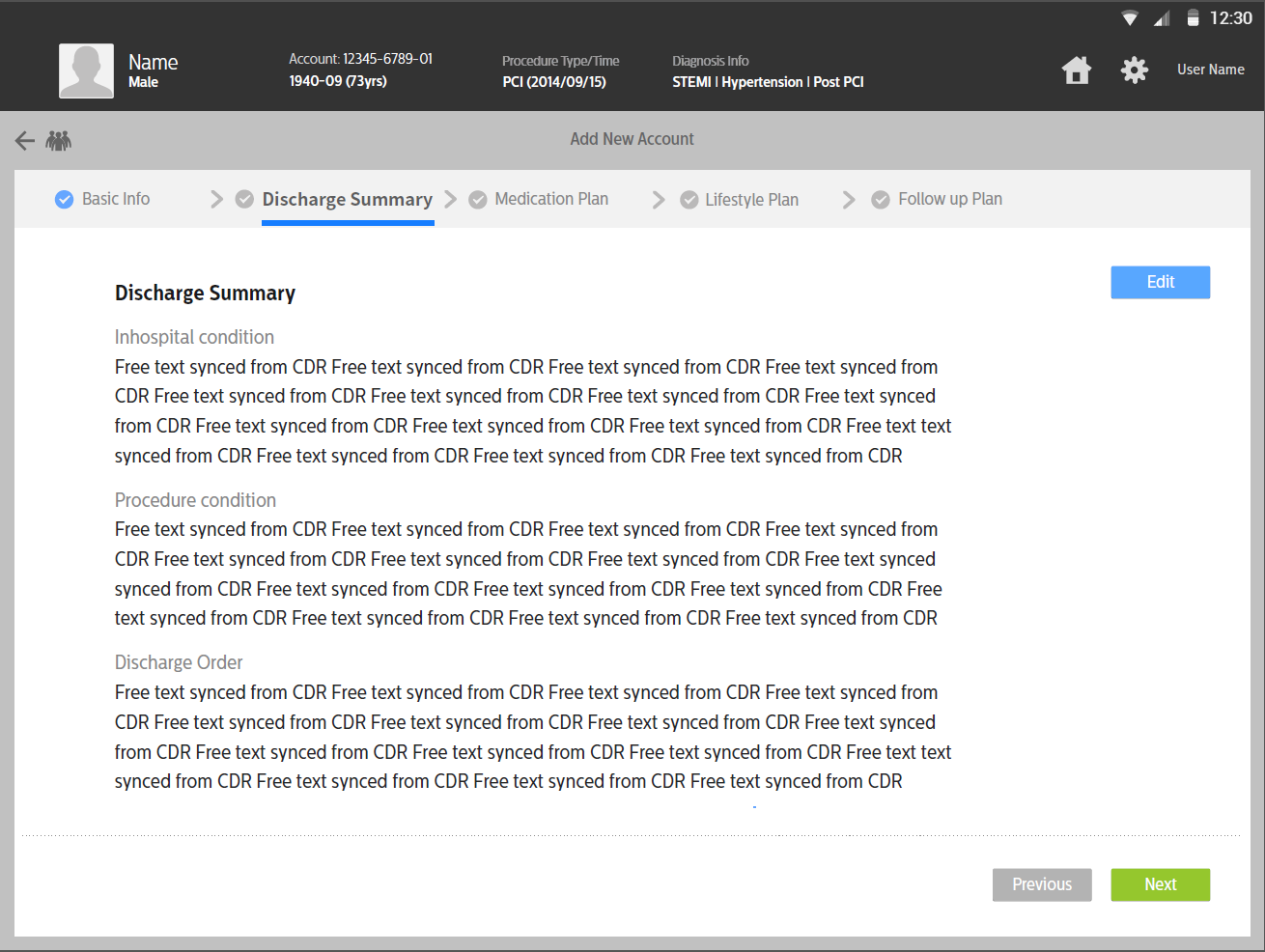
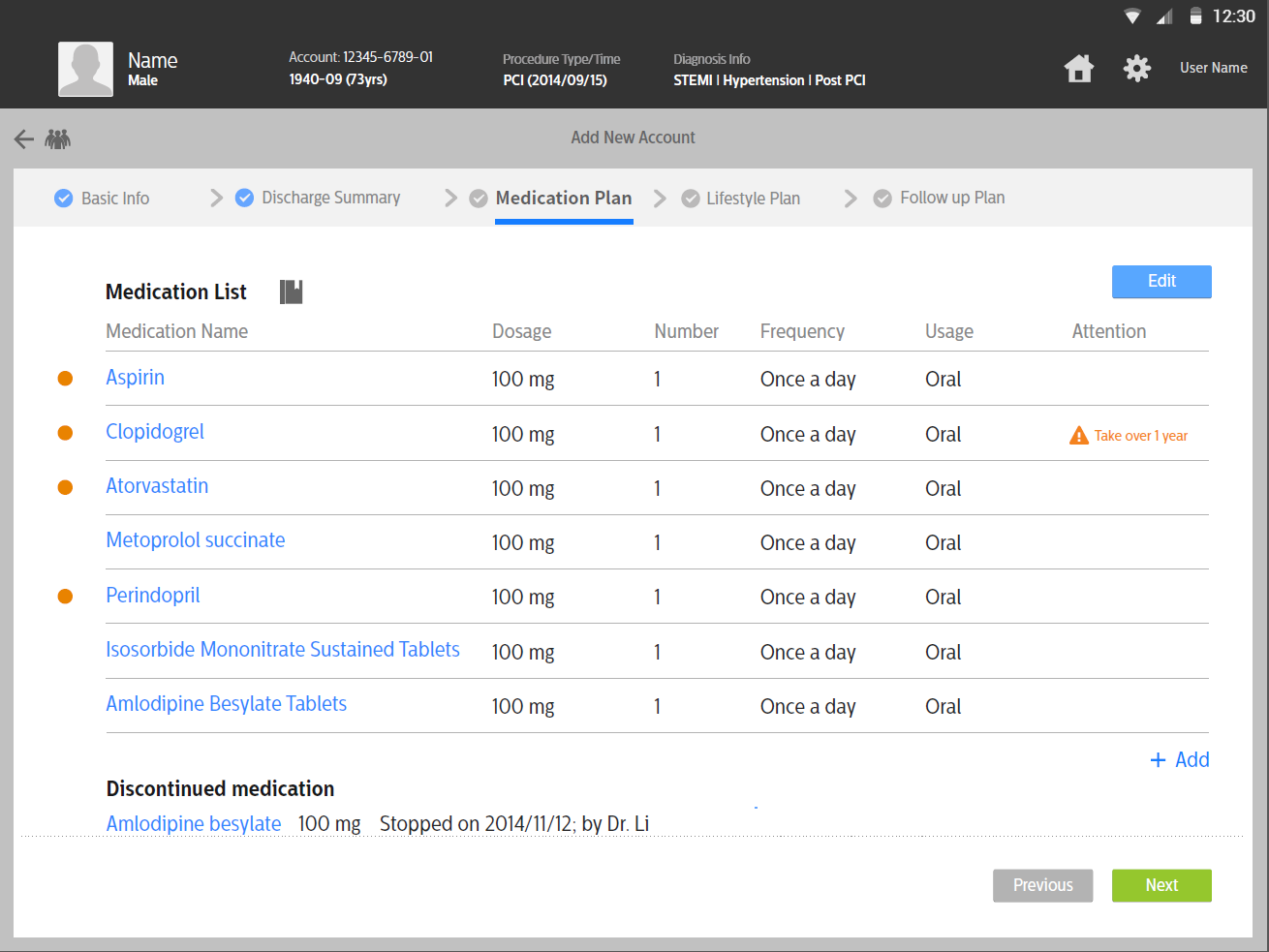
Page 6. Medication plan – physician create medication plan for patient

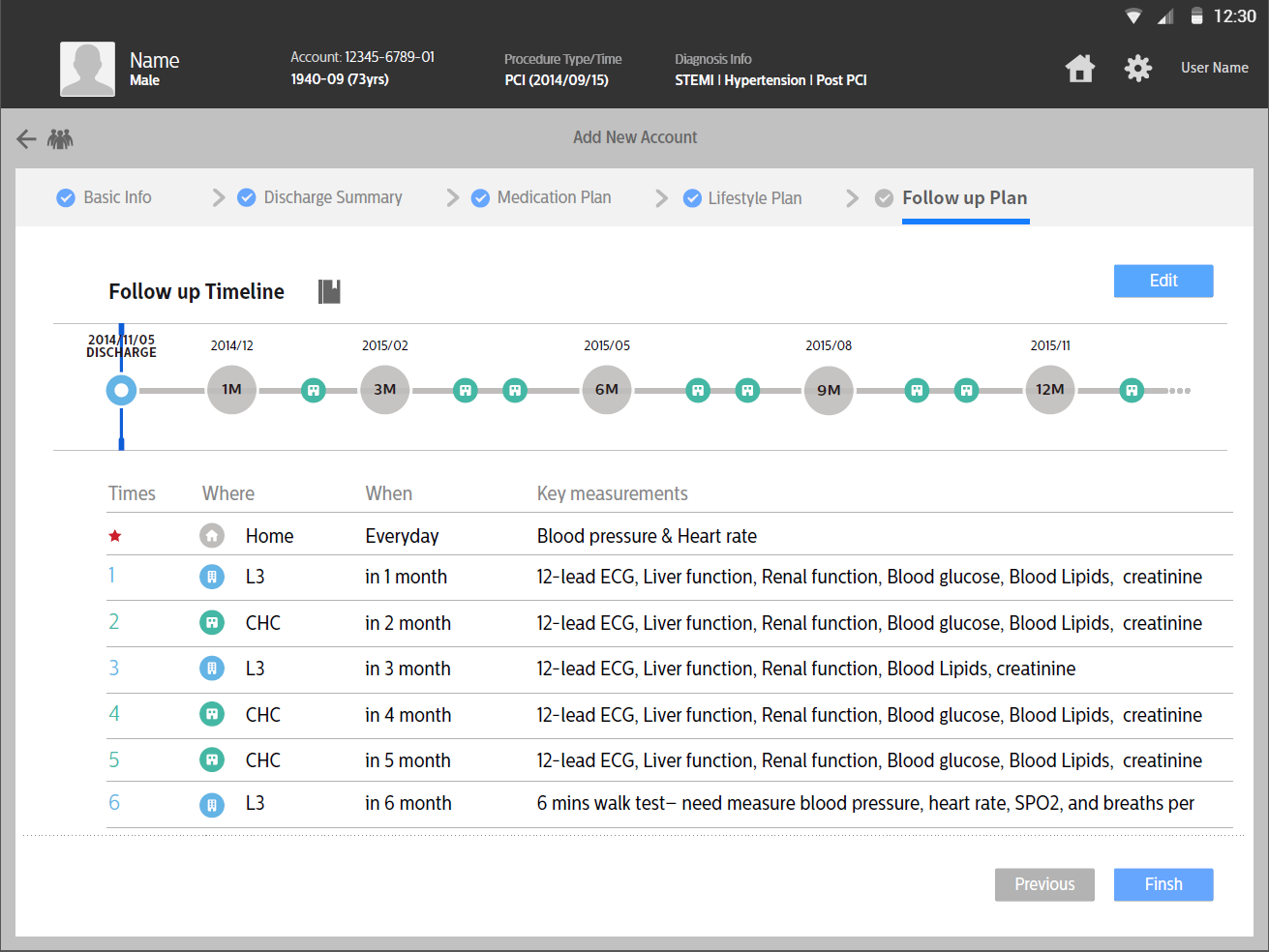
Page 7. Lifestyle plan – physician create lifestyle plan for patient

Page 8. Followup plan – physician create followup plan for patient

### Patient portal

Page 1. Login

Page 2. Main home – task view, medication taken view, score view…

Page 3.1 – 3.4 Medication list, and CRUD of medication

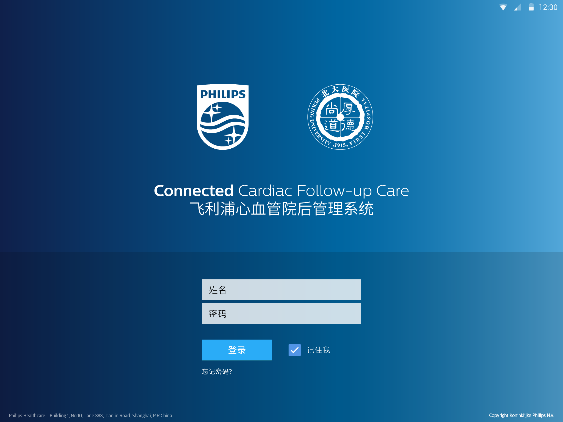
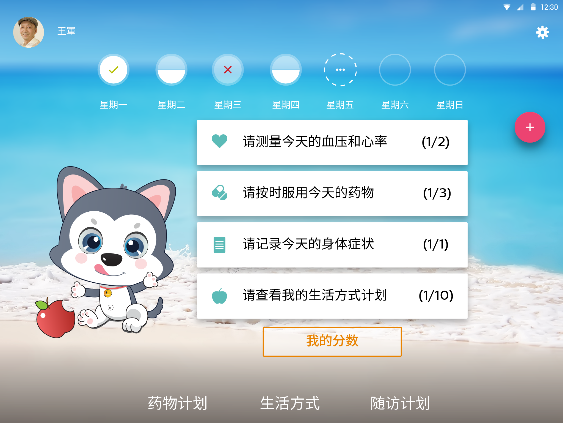
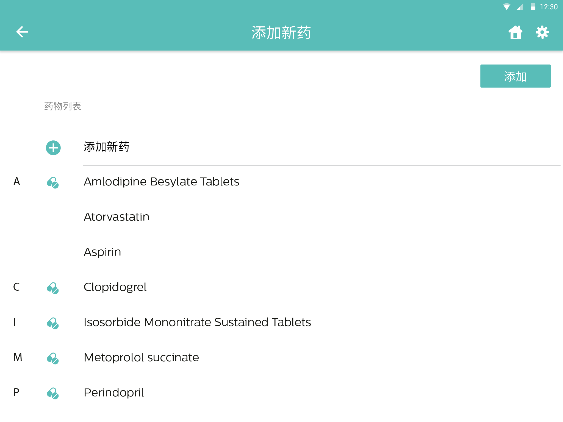
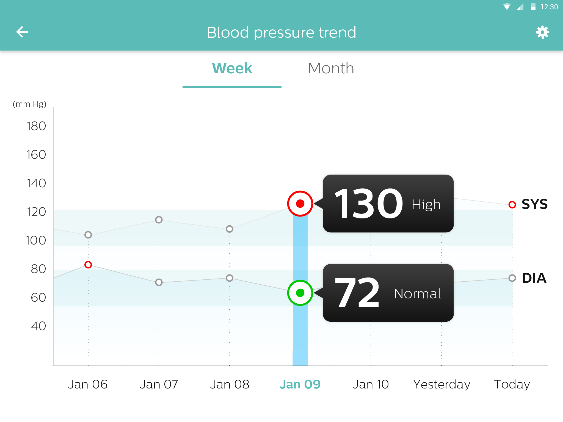
Page 4. Lifestyle

Page 5. Followup

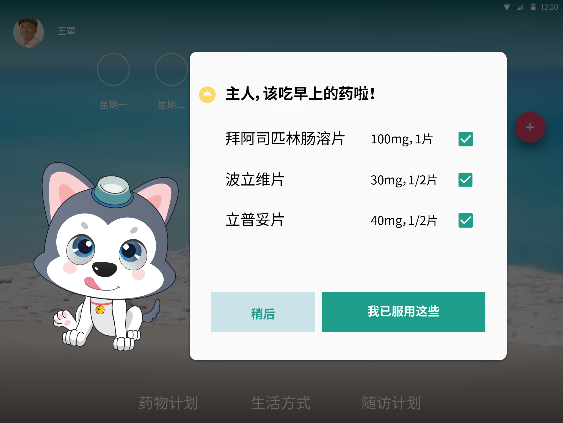
Page 6.1 - 6.3 Measurement view and input

Page 7.1 – 7.2 Annotation including symptom view and input

Page 8. Medication taken reminder



## Data Model

Based on data from CDR server and UI wireframe, the data model is designed as figure 7 and figure 8 including patient, measurement, medication taken record and annotation

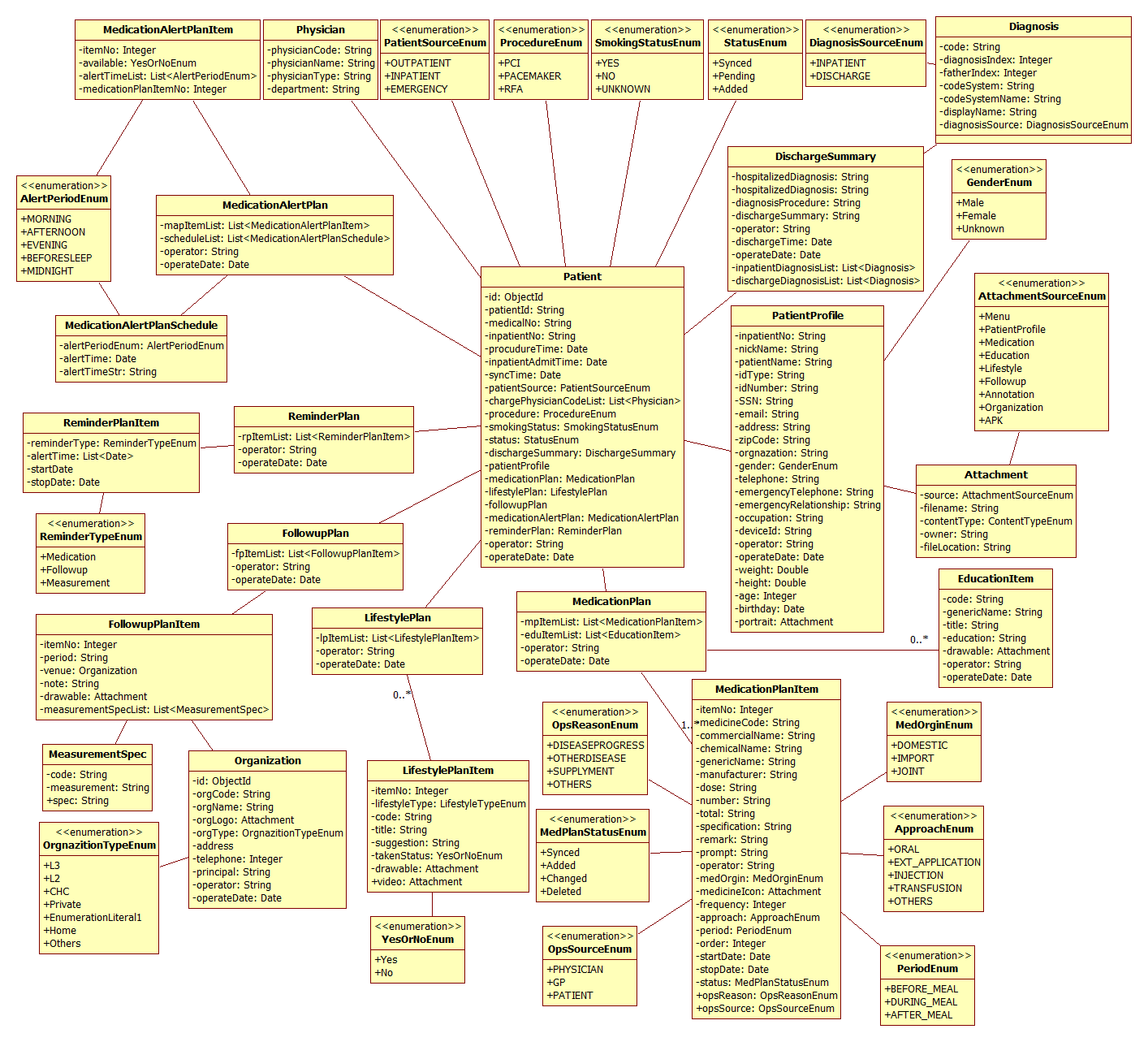


Figure 7. the class diagram of patient

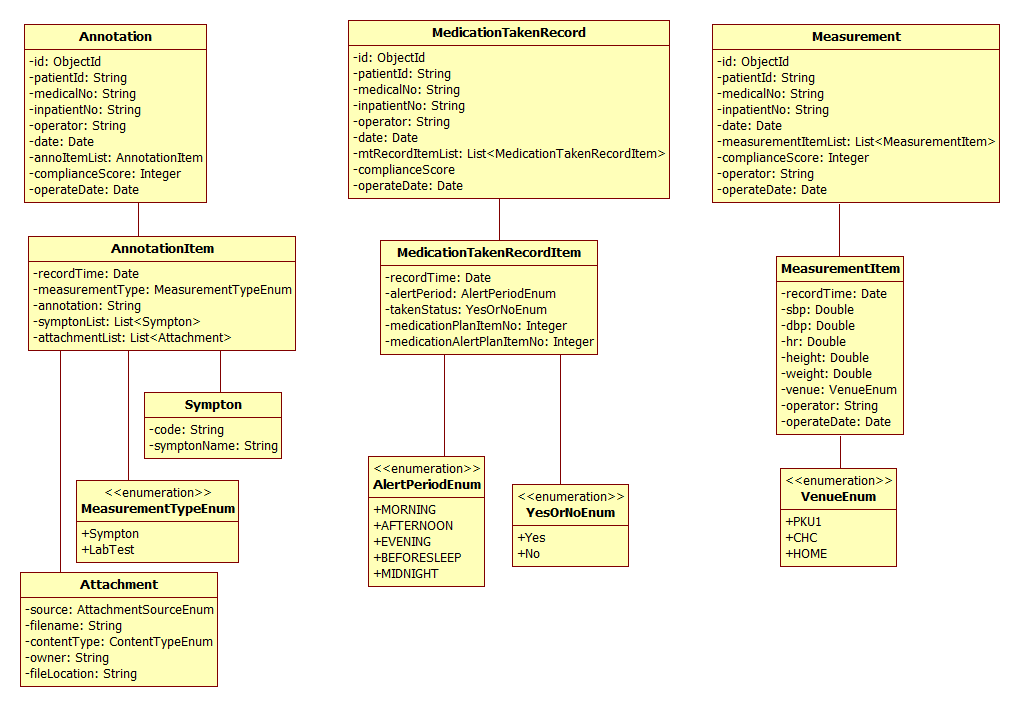


Figure 8. the class diagram of measurement, medication taken recor and annotation

## Activity Diagram of Patient portal

Figure 9 shows the activity diagram of patient portal(because physician portal is developed by india team), generally one page of android native app is mapped to one Activity, tab inside page is mapped to one fragment inside activity. Most activity works during app running except two system level service, one is when medication taken time is up, MedicineReminderRecordActivity is called to remind patient to take medication, second is periodically check back-server there is new app version or not. The transfer of activity state is total euqal to UI wireframe.

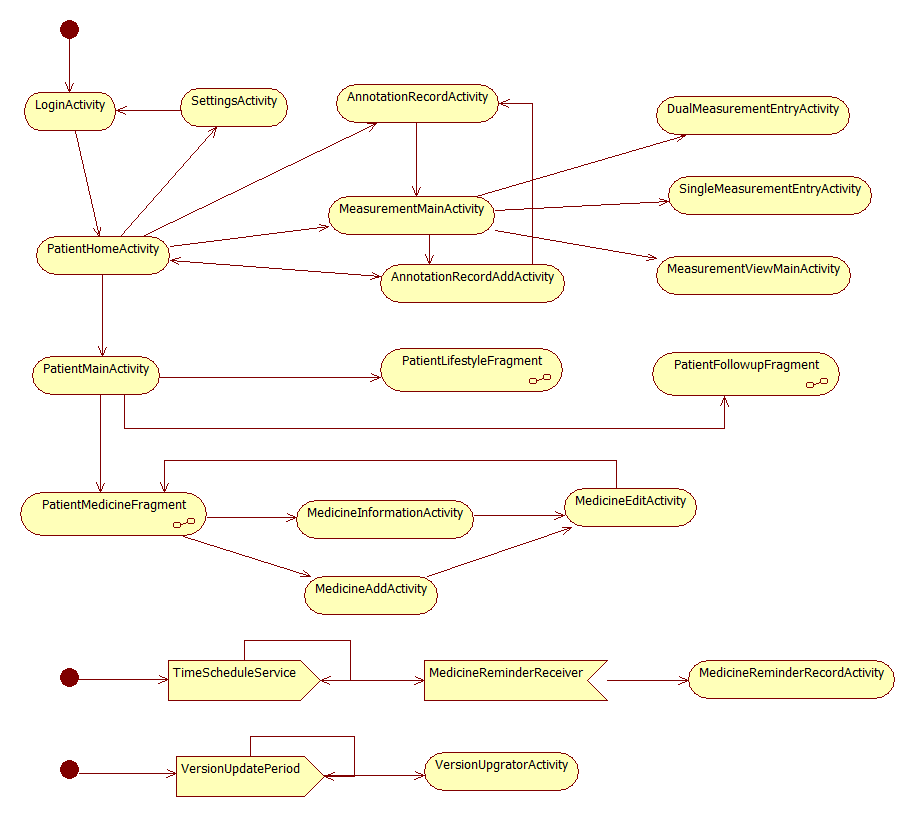


Figure 9. Activity Diagram of Patient portal

## Offline support

Figure 10 shows all related class combination for offline support. The local database is “content://com.philips.pscs.ccs.data/ccs.db”, Table includes MeasurementData, PatientData, MedicationTakenRecrod, AnnotationData. Database Schema is set via DbSchema, CCSContentProvider implements the CRUD operation of local database generally. Then the concrect CRUD operation of table like MeasurementProvider is via context.getContentResolver().query/insert/delete/update.

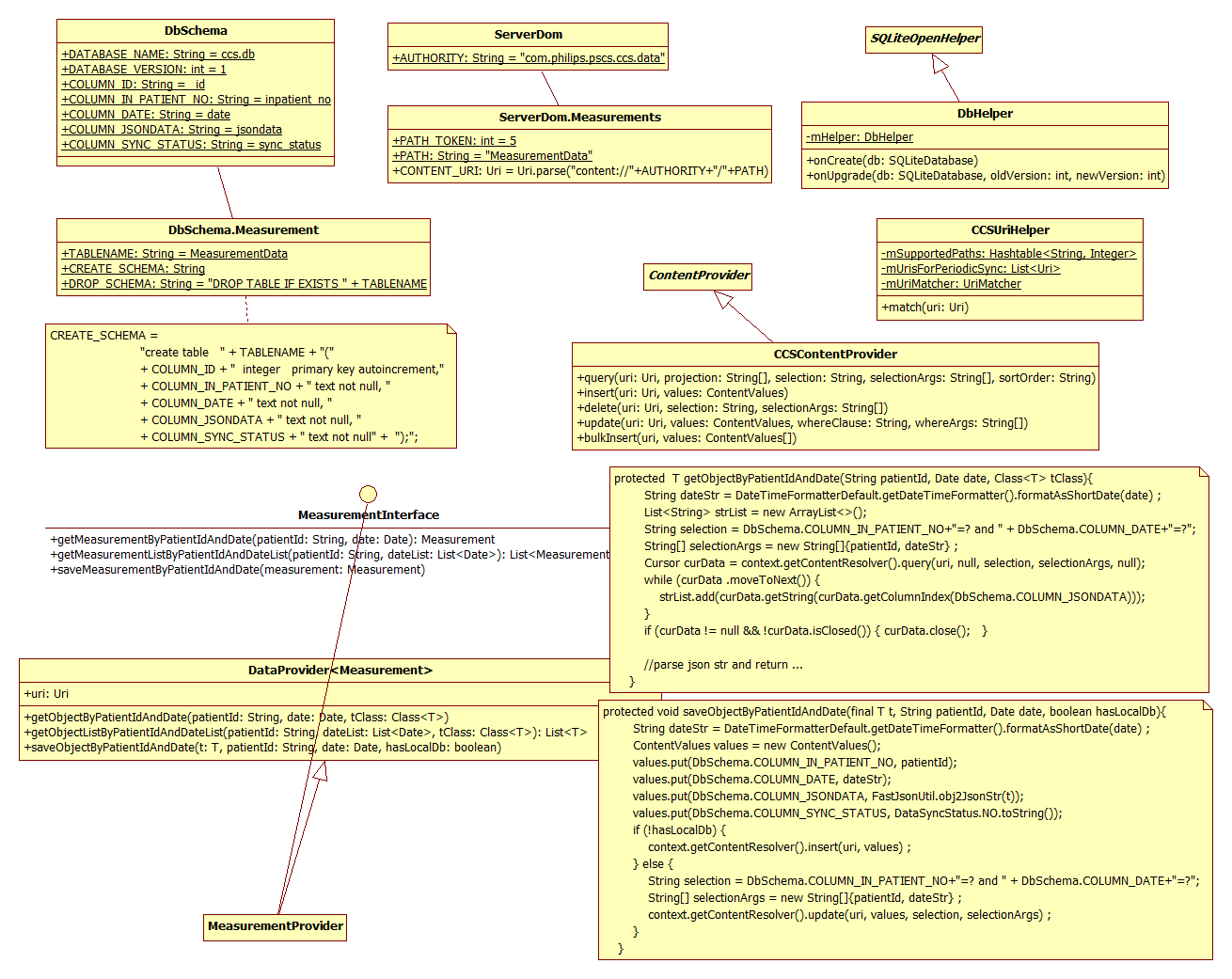


Figure 10. the diagram of offline support

## Data sync

Data sync between front-end and back-end is implemented via DataSyncManager, which is the front-end API to access back-end end-point remotely. Of three frameworks Spring RestTemplate, Retrofit and Volley, Retrofit, with better performance and clear API, is applied. Example code of how to use these three frameworks is referred to Appendix III

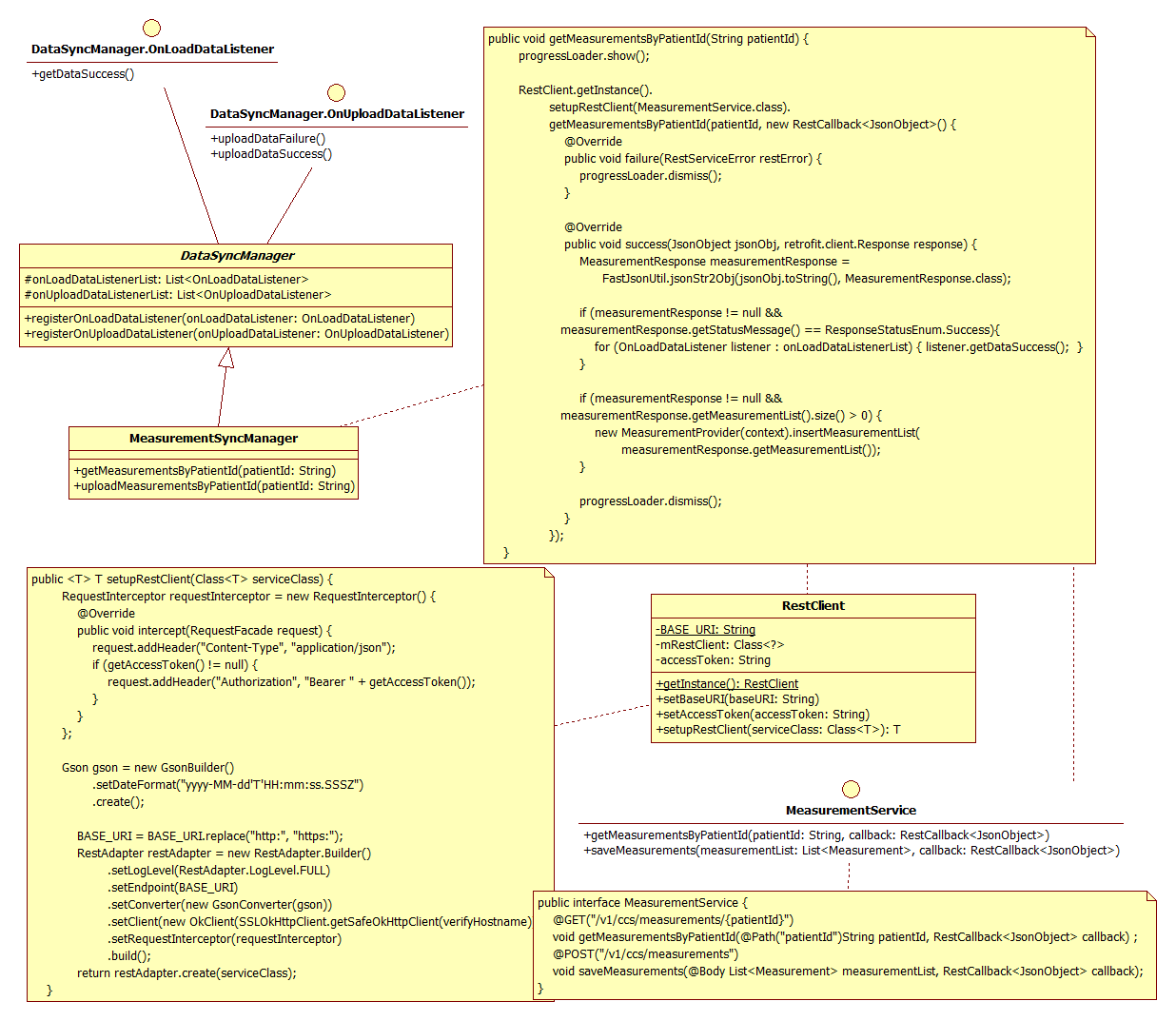


Figure 11. the diagram of Data sync between front-end and back-end

# Acknowledgements

We would like to thank the following colleagues for the co-creation of the research product and their valuable insights from the market and product:

UI team from Design at China: Zhang, Michelle Xl

PINS at China: LI, Eric, ZHU, Erin and Song, Kevin Tx

Market team at China: Emily Cao

# Appendix I - Micro service

## Why micro-service compared to monolith application?

Server-side enterprise application often supports a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response.

Generally the application has either a layered or hexagonal architecture[4] and consists of different types of components:

* Presentation components - responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIS);
* Business logic - the application’s business logic;
* Database access logic - data access objects responsible for access the database;
* Application integration logic - messaging layer, e.g. based on Spring integration.

Traditionally, the application is deployed as a monolith. For example, if you were using Java then the application would consist of a single WAR file running on a web container such as Tomcat. The monolith works well for relatively small applications. However, the monolithic architecture becomes unwieldy for complex applications. For example

* A large monolithic application can be difficult for developers to understand and maintain.
* It is also an obstacle to frequent deployments. To deploy changes to one application component you have to build and deploy the entire monolith,
* A monolithic architecture also makes it difficult to trial and adopt new technologies. It’s difficult, for example, to try out a new infrastructure framework without rewriting the entire application.
* Scaling the monolith application can be difficult - a monolithic architecture is that it can only scale in X-axis dimension via duplication the application as shown . the monolith application can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic.

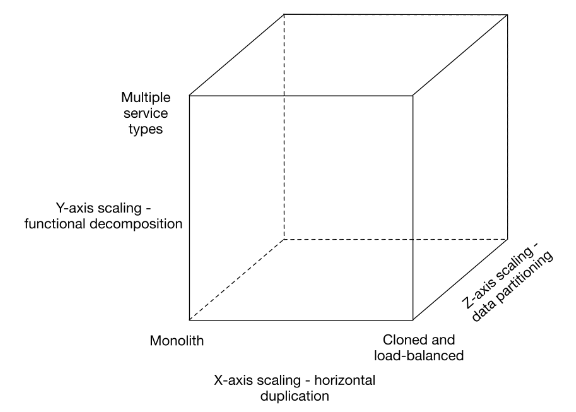


Figure 12. the scale cube: the dimension scalability model[]

As shown in figure 12, That’s a great way of improving the capacity and the availability of an application.Y-axis scaling is called functional decomposition at the application tier, Y-axis scaling splits a monolithic application into a set of services. Each service implements a set of related functionality, which is called micro-service. This solution has a number of benefits:

* + - Each microservice is relatively small, which means it easier for a developer to understand, the IDE is faster making developers more productive and the web container starts faster.
    - Each service can be deployed independently of other services - easier to deploy new versions of services frequently
    - Easier to scale development. It enables you to organize the development effort around multiple teams, each (two pizza) team is responsible a single service, each team can develop, deploy and scale their service independently of all of the other teams.
    - Improved fault isolation

For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system; Each service can be developed and deployed independently; Eliminates any long-term commitment to a technology stack

## How to implement micro-service?

Services typically need to call one another. In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically as shown in Figure 13.

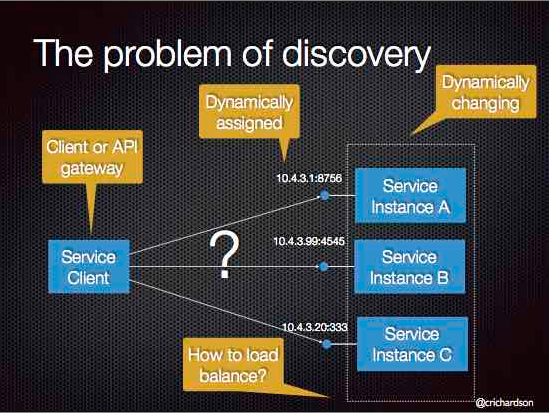
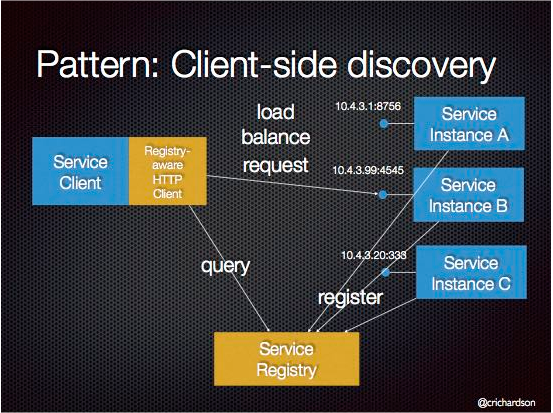
 

Figure 13 service discovery

Consequently, you must implement a mechanism for that enables the clients of service to make requests to a dynamically changing set of ephemeral service instances.

The solution is When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances. Eureka is the Netflix Service Discovery Server and Client. Each service find each other via server

Service Discovery: Eureka Server

The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others. Because Eureka server has no storage function, every service that registers to Eureka service, should send heart beat per 30 seconds, via serviceUrl in application.yml to indicate his life.

Example of eureka server

ServiceRegistryServer.java

@SpringBootApplication

@EnableEurekaServer

@EnableDiscoveryClient

public class ServiceRegistryServer {

public static void main(String[] args) {

SpringApplication.run(ServiceRegistryServer.class, args);

}

}

application.yml

server:

port: 8761

eureka:

client:

// by default, each eureka server is eureka client, also access peer via serviceUrl

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

instance:

hostname: localhost

//as standalone eureka server

client:

registerWithEureka: false

fetechRegistry: false

build.gradle

buildscript {

dependencies {

classpath "io.spring.gradle:dependency-management-plugin:0.4.0.RELEASE"

}

}

apply plugin: "io.spring.dependency-management"

dependencyManagement {

imports {

mavenBom 'org.springframework.cloud:spring-cloud-starter-parent:1.0.0.RELEASE'

}

}

The server has a home page with a UI, and HTTP API endpoints per the normal Eureka functionality under /eureka/\*. Eureka server uses spring-cloud-starter-eureka-server to set up the classpath

Service Discovery: Eureka Clients

When a client registers with Eureka, it provides meta-data about itself such as host and port, health indicator URL, home page etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

Example of eureka client:

@SpringBootApplication

@EnableEurekaClient

public class CcsApplication {

public static void main(String[] args) {

SpringApplication.run(CcsApplication.class, args);

}

}

Service will automatically register to Eureka server, the basic information will be recorded into server registry, in this way, each service could find each other

application.yml

eureka:

client:

serviceUrl:

defaultZone: <http://localhost:8761/eureka/>

instance:

hostname: localhost

nonSecurePort: 80

metadataMap:

instanceId: ${spring.application.name}:${spring.application.instance\_id:${random.value}}

Where "defaultZone" is a magic string fallback value that provides the service URL for any client that doesn’t express a preference (i.e. it’s a useful default).

The default application name (service ID), virtual host and non-secure port, taken from the Environment, are ${spring.application.name}, ${spring.application.name} and ${server.port} respectively.

@EnableEurekaClient makes the app into both a Eureka "instance" (i.e. it registers itself) and a "client" (i.e. it can query the registry to locate other services). The instance behaviour is driven by eureka.instance.\* configuration keys, but the defaults will be fine if you ensure that your application has a spring.application.name (this is the default for the Eureka service ID, or VIP).

# Appendix II – Security

## Authentification and Authroization Service

Firstly four roles are defined:

Resource owner - An entity capable of granting access to a protedted resource. When the resource owner is a person, it is referred to as an end-user

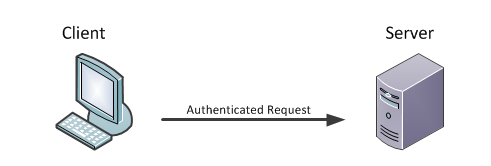
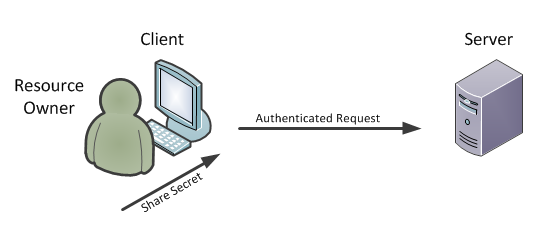
Resource server - the server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens

Client - an application making protected resource requests on behalf of the resource owner and with its authorization. The term “client” does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices.

Authorization server - the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

A protected resource like data (photos, documents, contacts), services (posting blog item, transferring funds), or any resource requiring access restrictions, is a resource stored on (or provided by) the resource server which requires authentication in order to access it. Protected resources are owned or controlled by the resource owner. Anyone requesting access to a protected resource must be authorized to do so by the resource owner (enforced by the resource server).

In the traditional client-server authentication model as shown in figure14, the client uses its credentials to access its resources hosted on the server. As far as the server is concerned, the shared secret used by the client belongs to the client, The server doesn’t really care where it came from or if the client is acting on behalf of some other entity as shown in figure. As long as the shared secret matches the server’s expectation, the request is processed.

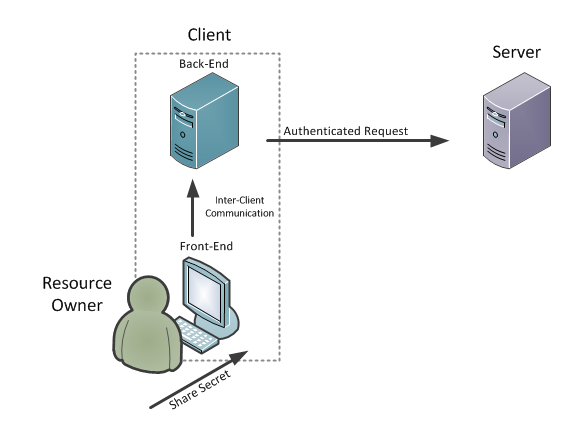


Figure 14 a. traditional client-server authentication model b,c. the client on behalf of another entity

There are many times when the client is acting on behalf of another entity as shown in figure 14. That entity can be another machine or person. It is enough that entity requires resource owner's credentials to make requests – pretending to be the resource owner. because User credentials typically include a username or screen-name and a password. There are serveral problems and limitations:

* The client are required to store the resource owner’s credentials for future use, typically a password in clear-text
* The client gain overly broad access to the resource owner’s protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.

Instead of using the resource owner’s credentials to access protected resources, the client obtains an access token – a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to the clients by an authorization server with the approval of the resource owner. The client uses the token credentials to access the protected resource without having to know the resource owner’s password.

The token[5] may denote an identifier used to retrieve the authorization information or may self-contain the authorization information, the access token provides an abstraction layer, replacing different authorization constructs (e.g., username and password) with a single token understood by the resource server and removing the resource server’s need to understand a wide range of authentication methods

The Protocol Flow of authentification is shown in figure 15.

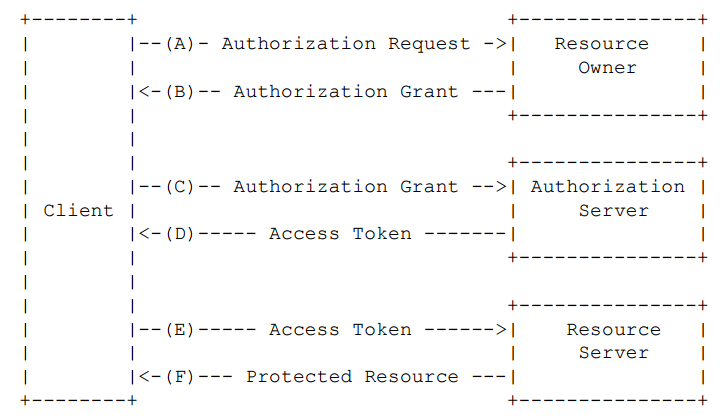


Figure 15. Protocol flow

(A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.

(B) The client receives an authorization grant, which is a credential representing the resource owner’s authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

(C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.

(E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F) The resource server validates the access token, and if valid, serves the request.

There are many server libraries for authentication and access-control. Spring Security provides comprehensive security services for Java EE-based enterprise software application. As you probably know two major areas of application security are “authentication” and “authorization”(or “access-control”). These are the two main areas that spring security targets. “Authentification” is the process of establishing a principal is who they claim to be (a “principal” generally means a user, device or some other system which can perform an action in your application). “Authorization” refers to the process of deciding whether a principal is allowed to perform an action within your application.

Resource Server (like api-gateway)

To use the access token you need a Resource Server (which can be the same as the Authorization Server). Creating a Resource Server is easy, just add @EnableResourceServer and provide some configuration to allow the server to decode access tokens. If your application is also an Authorization Server it already knows how to decode tokens, so there is nothing else to do

com.philips.pscs.cpc.gateway.security. CustomWebSecurityConfigurerAdapter.java

@Configuration

**public** **class** CustomWebSecurityConfigurerAdapter **extends**

WebSecurityConfigurerAdapter {

@Autowired **private** SecurityConfig securityConfig;

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.httpBasic().disable();

http.csrf().disable();

}

@Bean

**public** ResourceServerTokenServices tokenService() {

RemoteTokenServices tokenServices = **new** RemoteTokenServices();

tokenServices.setClientId(securityConfig.getClientId());

tokenServices.setClientSecret(securityConfig.getClientSecret());

tokenServices.setCheckTokenEndpointUrl(securityConfig.getAuthServer()

+ "oauth/check\_token");

**return** tokenServices;

}

@Override

**public** AuthenticationManager authenticationManagerBean() **throws** Exception {

OAuth2AuthenticationManager authenticationManager = **new** OAuth2AuthenticationManager();

authenticationManager.setTokenServices(tokenService());

**return** authenticationManager;

}

@Configuration

@EnableResourceServer

**protected** **static** **class** ResourceServerConfiguration **extends**

ResourceServerConfigurerAdapter {

@Autowired **private** SecurityConfig securityConfig;

@Override

**public** **void** configure(ResourceServerSecurityConfigurer resources) {

resources.resourceId(securityConfig.getResourceId());

}

@Override

**public** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests().antMatchers("/v1/ccs/\*\*").authenticated();

}

}

}

application.yml

authSecurity:

authServer: http://localhost:9005/

clientId: clientapp

clientSecret: 123456

resourceId: restservice

Authorization Server (like authorization-service)

To create an Authorization Server and grant access tokens you need to use @EnableAuthorizationServer and provide security.oauth2.client.client-id and security.oauth2.client.client-secret] properties. The client will be registered for you in an in-memory repository. Having done that you will be able to use the client credentials to create an access token, for example:

cmd> curl client:secret@localhost:8080/oauth/token -d grant\_type=password -d username=user -d password=pwd

The basic auth credentials for the /token endpoint are the client-id and client-secret. The user credentials are the normal Spring Security user details (which default in Spring Boot to “user” and a random password).

com.philips.pscs.cpc.authorization.security.CustomAuthenticationServer

@Configuration

@EnableAuthorizationServer

**public** **class** CustomAuthenticationServer **extends** AuthorizationServerConfigurerAdapter {

**private** TokenStore tokenStore = **new** InMemoryTokenStore();

@Autowired **private** SecurityConfig securityConfig;

@Autowired @Qualifier("authenticationManagerBean")

**private** AuthenticationManager authenticationManager;

//config user information store (database)

@Autowired **private** CustomUserDetailsService userDetailsService;

@Override

**public** **void** configure(AuthorizationServerEndpointsConfigurer endpoints)

**throws** Exception {

endpoints

.tokenStore(**this**.tokenStore)

.authenticationManager(**this**.authenticationManager)

.userDetailsService(userDetailsService)

.accessTokenConverter(accessTokenConverter())

.tokenEnhancer(tokenEnhancer());

}

//customize token

@Bean **public** TokenEnhancer tokenEnhancer() {

**return** **new** CustomTokenEnhancer();

}

@Bean **public** DefaultAccessTokenConverter accessTokenConverter() {

**return** **new** DefaultAccessTokenConverter();

}

@Override

**public** **void** configure(ClientDetailsServiceConfigurer clients) **throws** Exception {

clients

.inMemory()

.withClient(securityConfig.getClientId()) .authorizedGrantTypes(SecurityConstants.***GrantType\_Password***, SecurityConstants.***GrantType\_RefreshToken***) .scopes(SecurityConstants.***Scope\_Read***, SecurityConstants.***Scope\_Write***)

.resourceIds(securityConfig.getResourceId())

.secret(securityConfig.getClientSecret());

}

@Override

**public** **void** configure(AuthorizationServerSecurityConfigurer oauthServer) **throws** Exception {

oauthServer.checkTokenAccess("permitAll()");

}

@Bean @Primary

**public** DefaultTokenServices tokenServices() {

DefaultTokenServices tokenServices = **new** DefaultTokenServices();

tokenServices.setTokenEnhancer(tokenEnhancer());

tokenServices.setSupportRefreshToken(**true**);

tokenServices.setTokenStore(**this**.tokenStore);

**return** tokenServices;

}

}

CustomUserDetailsService.java

@Service

**public** **class** CustomUserDetailsService **implements** UserDetailsService {

**private** UserDao userRepository;

@Autowired **public** CustomUserDetailsService(UserDao userRepository) {

**this**.userRepository = userRepository;

}

@Override

**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {

User user = userRepository.getUser(username);

**if** (user == **null**) {

**throw** **new** UsernameNotFoundException(String.*format*("User %s does not exist!", username));

}

user.setPassword(decryptPassword(user.getPassword()));

**return** **new** UserRepositoryUserDetails(user);

}

**private** String decryptPassword(String password) {

**byte**[] passwordBytes = password.getBytes();

**byte**[] decodedBytes = Base64.*decodeBase64*(passwordBytes);

String decodedPassword = **new** String(decodedBytes);

**return** decodedPassword;

}

**private** **final** **static** **class** UserRepositoryUserDetails **extends** User **implements** UserDetails {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** UserRepositoryUserDetails(User user) {**super**(user); }

@Override

**public** Collection<? **extends** GrantedAuthority> getAuthorities() {

**return** getRoles();

}

@Override **public** String getUsername() {**return** getLoginId();}

@Override **public** **boolean** isAccountNonExpired() {**return** **true**;}

@Override **public** **boolean** isAccountNonLocked() {**return** **true**; }

@Override **public** **boolean** isCredentialsNonExpired() {**return** **true**;}

@Override **public** **boolean** isEnabled() {**return** **true**;}

}

}

CustomTokenEnhancer.java

@Configuration

**public** **class** CustomTokenEnhancer **implements** TokenEnhancer {

@Override **public** OAuth2AccessToken enhance(OAuth2AccessToken accessToken, OAuth2Authentication authentication) {

DefaultOAuth2AccessToken result = **new** DefaultOAuth2AccessToken(accessToken);

User user = (User) authentication.getPrincipal(); **final** Map<String, Object> additionalInformation = **new** HashMap<>();

additionalInformation.put("firstName", user.getFirstName());

additionalInformation.put("lastName", user.getLastName());

additionalInformation.put("loginId", user.getLoginId());

additionalInformation.put("roles", AuthorityUtils.*authorityListToSet*(authentication.getAuthorities()));

result.setAdditionalInformation(additionalInformation);

**return** result;

}

}

application.yml

authSecurity:

clientId: clientapp

clientSecret: 123456

resourceId: restservice

## HTTPS

HTTPS (HTTP over SSL or HTTP Secure) is the use of Secure Socket Layer (SSL) or Transport Layer Security (TLS) as a sublayer under regular HTTP application layering. HTTPS encrypts and decrypts user page requests as well as the pages that are returned by the Web server.

If you are using Spring Boot and want to enable SSL (https) for your application on the embedded Tomcat there a few short steps you will need to take.

Step1. Get yourself a SSL certificate

generate a self-signed certifcate or get one from a Certificate Authority, Every Java Runtime Environment (JRE) comes bundled with a certificate management utility, keytool which is located as %JAVA\_HOME%/bin/Keytool.exe. This can be used to generate our self-signed certificate

// generate java keystore and digital certificate

cmd> keytool -genkey -alias tomcat -keyalg RSA -keystore d:\mykeystore.p12 -dname "CN=localhost, OU=localhost, O=localhost, L=SH, ST=SH, C=CN" -keypass changeit -storepass -validity 180

// different formats like p12, pfx, bks, jks could be translated

p12 -> pfx: because both files have same binary format, just change the extention

p12 -> jks:

cmd> keytool -importkeystore -srckeystore [MY\_FILE.p12] -srcstoretype pkcs12

-srcalias [ALIAS\_SRC] -destkeystore [MY\_KEYSTORE.jks]

-deststoretype jks -deststorepass [PASSWORD\_JKS] -destalias [ALIAS\_DEST]

p12 -> bks

cmd> keytool –export –alias tomcat –keystore keystore.p12 –file keystore.crt

cmd> keytool –import –alias tomcat –file keystore.crt –keystore keystore.bks –storetype BKS –provider org.bouncycastle.jce.provider.BouncyCastleProvider

Step2. Enable HTTPS in Spring Boot

application.yml

server:

port: 9002

ssl:

key-store: classpath:keystore.p12

key-store-password: changeit

keyAlias: tomcat

key-password: changeit

Step 3. Access HTTS

web explorer:

<https://localhost:9002>

postman: first please install keystore.p12 (double click->…)

<https://localhost:9002> GET

Java client:

SSLConnectionSocketFactory socketFactory = **new** SSLConnectionSocketFactory(

**new** SSLContextBuilder() .loadTrustMaterial(**null**, **new** TrustSelfSignedStrategy()).build());

HttpClient httpClient = HttpClients.*custom*().setSSLSocketFactory(socketFactory)

.build();

TestRestTemplate testRestTemplate = **new** TestRestTemplate();

((HttpComponentsClientHttpRequestFactory) testRestTemplate.getRequestFactory()).setHttpClient(httpClient);

ResponseEntity<String> entity = testRestTemplate .getForEntity("https://localhost:" + **this**.port, String.**class**);

*assertEquals*(HttpStatus.***OK***, entity.getStatusCode());

Android client:

keystore.bks <- keystore.p12

RestAdapter restAdapter = new RestAdapter.Builder()  
 .setEndpoint(url)  
 .setClient(new OkClient(SSLOkHttpClient.*getSafeOkHttpClient*(false)))  
 .build() ;  
GreetingService service = restAdapter.create(GreetingService.class) ;  
Greeting greeting = service.greeting("QiZhong Lin") ;  
return greeting ;

public class SSLOkHttpClient {  
  
 private static final String *SERVER\_IP* = "161.92.141.144";  
 private static final int *SERVER\_PORT* = 9002;  
 private static final String *CLIENT\_TRUST\_MANAGER* = "X509";  
 private static final String *CLIENT\_TRUST\_PASSWORD* = "changeit";  
 private static final String *CLIENT\_AGREEMENT* = "TLS";  
 private static final String *CLIENT\_TRUST\_KEYSTORE* = "BKS";  
 private static final String *CLIENT\_CACERT\_BKS* = "res/raw/keystore.bks";  
  
 public static OkHttpClient getSafeOkHttpClient(final boolean verifyHostname) {  
 try {  
 TrustManagerFactory trustManager = TrustManagerFactory.*getInstance*(*CLIENT\_TRUST\_MANAGER*);  
 KeyStore tks = KeyStore.*getInstance*(*CLIENT\_TRUST\_KEYSTORE*);  
 InputStream is = SSLOkHttpClient.class.getClassLoader().getResourceAsStream(*CLIENT\_CACERT\_BKS*);  
 tks.load(is, *CLIENT\_TRUST\_PASSWORD*.toCharArray());  
 trustManager.init(tks);  
  
 final SSLContext sslContext = SSLContext.*getInstance*(*CLIENT\_AGREEMENT*);  
 sslContext.init(null, trustManager.getTrustManagers(), new java.security.SecureRandom());  
 final SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();  
  
 OkHttpClient okHttpClient = new OkHttpClient();  
 okHttpClient.setSslSocketFactory(sslSocketFactory);  
 okHttpClient.setHostnameVerifier(new HostnameVerifier() {  
 @Override  
 public boolean verify(String hostname, SSLSession session) {  
 if(verifyHostname){  
 X509Certificate[] peerCerts;  
 try {  
 peerCerts = session.getPeerCertificateChain();  
 } catch (SSLPeerUnverifiedException e) {  
 return false;  
 }  
 for (X509Certificate peerCert : peerCerts) {  
 Principal x500s = peerCert.getSubjectDN();  
 try {  
 peerCert.checkValidity();  
 } catch (Exception e) {  
 return false;  
 }  
 }  
 return true;  
 } else {  
 return true;  
 }  
 }  
 });  
 return okHttpClient;  
 } catch (Exception e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

# Appendix III - RESTful request of android app

Method1. Spring RestTemplate

build.gradle

dependencies {

compile 'org.springframework.android:spring-android-rest-template:2.0.0.M1'

compile 'com.fasterxml.jackson.core:jackson-databind:2.3.2'

}

// work thread

private class HttpRequestTask extends AsyncTask<Void, Void, Greeting> {

@Override protected Greeting doInBackground(Void... params) {

try {

final String url = "http://161.92.141.144:8080/greeting";

return fetchDataByRestTemplate(String url);

} catch (Exception e) {

Log.e("MainActivity", e.getMessage(), e);

}

return null;

}

private **Greeting** fetchDataByRestTemplate(String url){

RestTemplate restTemplate = new RestTemplate();

restTemplate.getMessageConverters().add(new MappingJackson2HttpMessageConverter());

Greeting greeting = restTemplate.getForObject(url, Greeting.class);

return greeting;

}

//the result is sent to main thread

@Override protected void onPostExecute(**Greeting greeting**) {

TextView greetingIdText = (TextView) findViewById(R.id.id\_value);

TextView greetingContentText = (TextView) findViewById(R.id.content\_value);

greetingIdText.setText(greeting.getId());

greetingContentText.setText(greeting.getContent());

}

}

// Main thread

new HttpRequestTask().execute();

Method 2. Retrofit

build.gradle

dependencies {

compile 'com.squareup.retrofit:retrofit:1.9.0'

}

GreetingService.java

public interface GreetingService {

@GET("/")

Greeting greeting(@Query("name")String name) ;

@GET("/")

void greeting(@Query("name")String name, Callback<Greeting> callback) ;

}

RestAdapter restAdapter = new RestAdapter.Builder().setEndpoint(url).build() ;

GreetingService service = restAdapter.create(GreetingService.class) ;

service.greeting("QiZhong Lin", new Callback<Greeting>() {

@Override

public void success(Greeting greeting, Response response) {

updateActivity(greeting);

}

@Override

public void failure(RetrofitError error) {

}

});

Method 3. Volley

build.gradle

dependencies {

compile 'com.mcxiaoke.volley:library:1.0.+'

}

RequestQueue requestQueue = Volley.newRequestQueue(context) ;

JsonObjectRequest request = new JsonObjectRequest(url, null,

new Response.Listener<JSONObject>() {

@Override public void onResponse(JSONObject response) {

Greeting greeting = new Gson().fromJson(response.toString(), Greeting.class) ;

updateActivity(greeting);

}

},

new Response.ErrorListener(){

@Override public void onErrorResponse(VolleyError error) {

Log.e("TAG", error.getMessage(), error);

}

}) ;

requestQueue.add(request) ;

References

1. <https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>.
2. MongoDB: The Definitive Guide,Kristina Chodorow.
3. Spring in action(fourth edition), Criag Walls. Pages: 327 – 340
4. <http://c2.com/cgi/wiki?HexagonalArchitecture>
5. <http://oauth.net/>