SmartSync Architecture Evolution

30 April 2017

Jack Meyer

Nathan Karasch

Charlie Steenhagen

Trevor Henderson

Front-end Code Repository:
https://github.com/KrashLeviathan/SmartSync-Dashboard

Server Code Repository:
https://github.com/jackcmeyer/SmartSync

Demo Video:
https://www.youtube.com/watch?v=OpqfT2fiN9k&feature=youtu.be

Table of Contents

Table of Contents 2

Project Scope 3

Software Features 4

- Login Authentication with Google 4
- User Management 5
- User and Household Settings 6
- Dashboard View 7
- Service Management 8
- Other Features 9

Quality Attributes 10

- Architectural Changes 10
- Acknowledge Patterns 10

Architecture Evolution 11

- Phase 3 Use Case Diagram 11
- Updated Use Case Diagram 12
- Phase 3 Component Diagram 13
- Updated Component Diagram 13
- Phase 3 Deployment Diagram 14
- Updated Deployment Diagram 15
- Updated Class Diagrams 16
  - Server Side 16
  - Client Side 18
- Discussion 19
  - Add, Delete, View Services 19
  - Secure Endpoints and Data 20
  - Invite, Join, Create a Household 20

Architecture Recovery 21

Conclusion 22

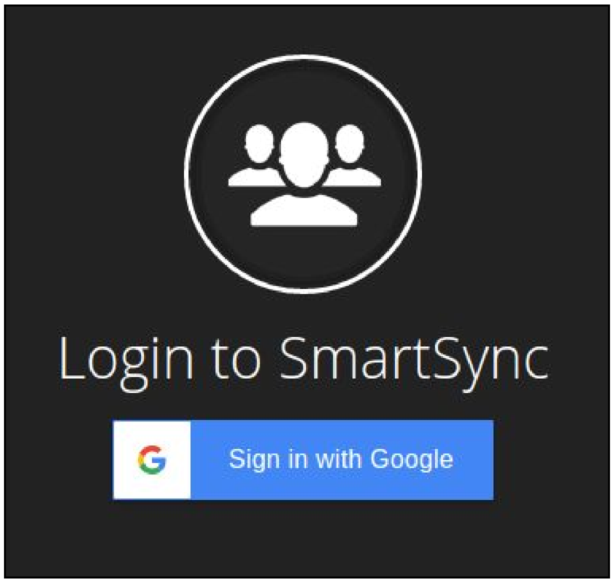
Project Scope

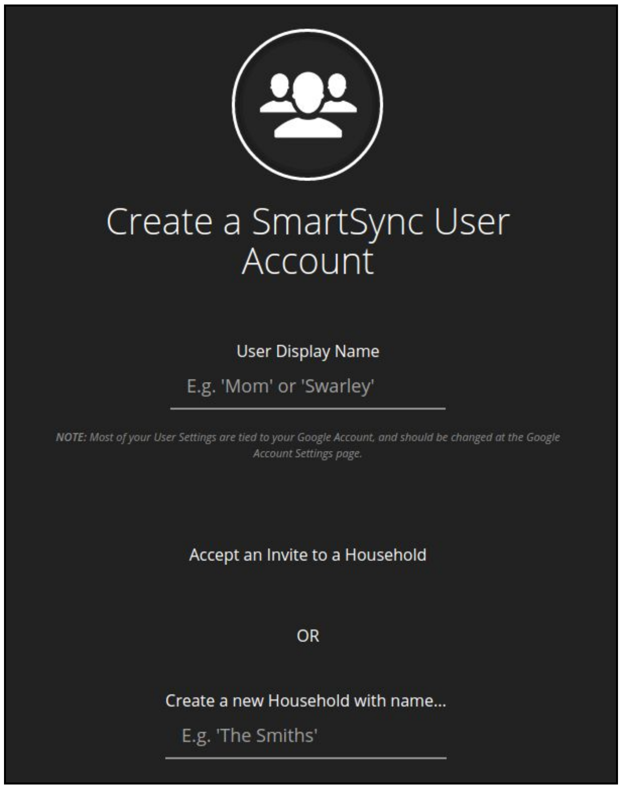
SmartSync is a home management system for synchronizing Internet of Things (IoT) devices, family members' online accounts, and other user applications. It runs as a web application, interfacing with various services to provide a more efficient home for the modern family.

The Minimal Viable Product for Smart Sync will consist of the following:

1. Provide a platform that allows the user to synchronize and manage Internet of Things devices and services.
2. Integrate at least one device or service from each of the following categories
    a. Internet of Things Devices
    b. Online Account Services
    c. Internet Services
3. Provide a "Dashboard" view that displays the status of multiple devices/services simultaneously
4. Provide household account registration with a household admin user and one or more standard users
5. Implement basic security measures for the system (password encryption, input validation, etc.)

Software Features



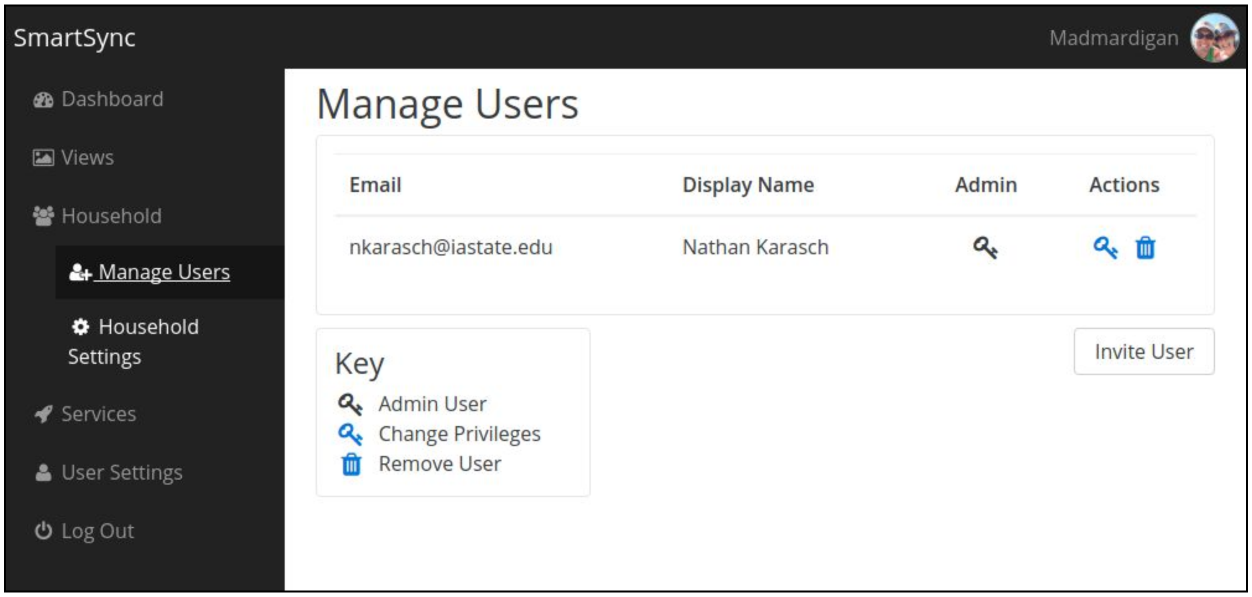

Create a SmartSync User Account
User Display Name
E.g. 'Mom' or 'Swarley'

NOTE: Most of your User Settings are tied to your Google Account, and should be changed at the Google Account Settings page.

Accept an Invite to a Household
OR
Create a new Household with name...
E.g. 'The Smiths'

Login Authentication with Google

We chose Google's OAuth2 API to authenticate our application because it is almost never a good idea to tackle authentication on your own. Google's API is well-documented, but we ran into problems due to the fact that we were using Angular2 for a single-page application. On successful authentication, Google's API reloads the page or redirects the page. This didn't work completely well with our system since the entire application is supposed to be navigated without leaving the page. In retrospect, the architecture would have benefitted from having the login and registration pages as static HTML instead of being part of the main Angular2 application. We could have avoided the page redirect issues, and the architecture would have been simplified.



Manage Users

Email	Display Name	Admin	Actions
nkarasch@iastate.edu	Nathan Karasch		

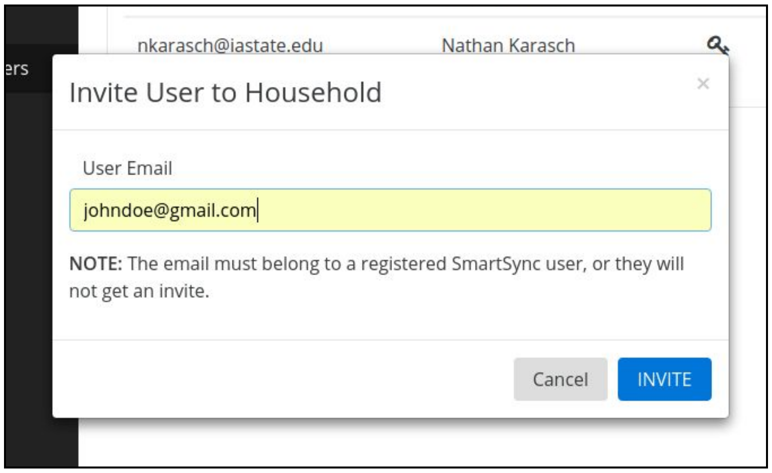
Key
Admin User
Change Privileges
Remove User

User Management

If a user creates a new household, he is automatically the admin user of that household. Admin users have the ability to invite other users to the household, remove them from the household, and give or revoke admin privileges to other users.

The admin user view looks slightly different from the regular user view. The “Manage Users” and “Household Settings” pages become available, and in the “Services” page there are additional actions that can be performed by the admin, such as adding a new service to the household, configuring services, etc.

The difference between admin user privileges and regular user privileges was something we had to address in our architecture, and it took most of the semester to figure out. The authentication service is talked about later, so it won’t be elaborated on here.

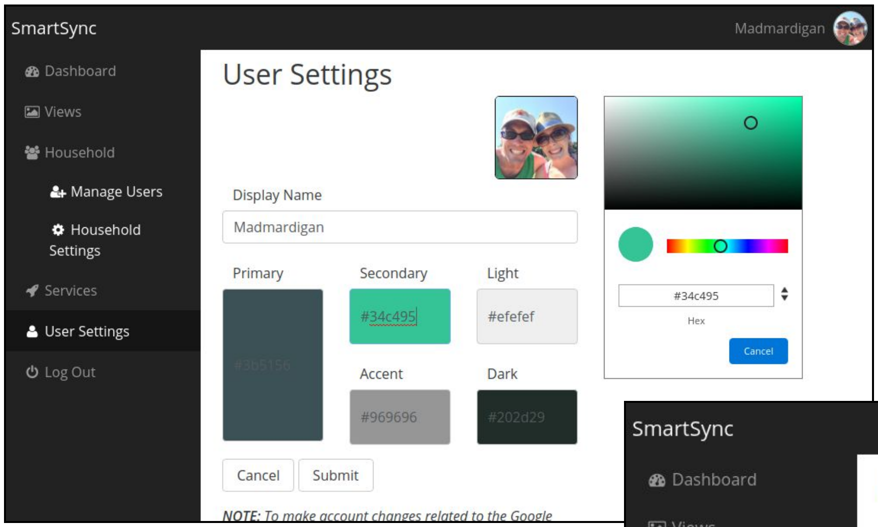


Invite User to Household

User Email

johndoe@gmail.com

NOTE: The email must belong to a registered SmartSync user, or they will not get an invite.



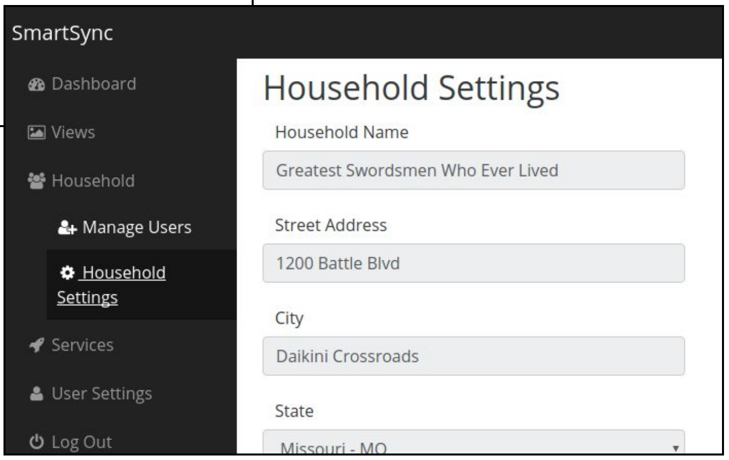
User Settings

Display Name: Madmardigan

Primary: #3b5156
Secondary: #34c495
Light: #efefef
Accent: #969696
Dark: #202d29

#34c495

NOTE: To make account changes related to the Google



Household Settings

Household Name: Greatest Swordsmen Who Ever Lived

Street Address: 1200 Battle Blvd

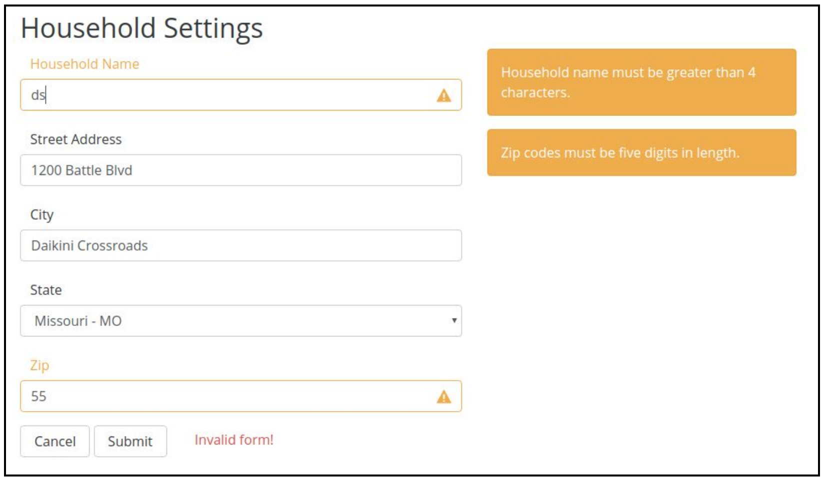
City: Daikini Crossroads

State: Missouri - MO

User and Household Settings

Every user is allowed to make limited customizations to their profile. Their picture, email, name, etc are tied to their Google account, so any changes pertaining to that should be done at their Google account screen. The “User Settings” page features a nice color-selector component for changing the theme of an individual user’s dashboard. The choice of Angular2 made the addition of this component to the architecture fairly simple, since the entire front end is using npm for dependency management. It was a matter of installing the component, inserting it into the UI, and hooking up the form fields to the component. However, there wasn’t enough time to implement the theme colors in the application. They’re simply stored with the user’s preferences on the server.

With the security requirements came the need for input validation. Input validation is done on the front and back end, with Angular2 and Bootstrap elements helping notify the user of invalid form data before sending it to the server.



Household Settings

Household Name: ds

Street Address: 1200 Battle Blvd

City: Daikini Crossroads

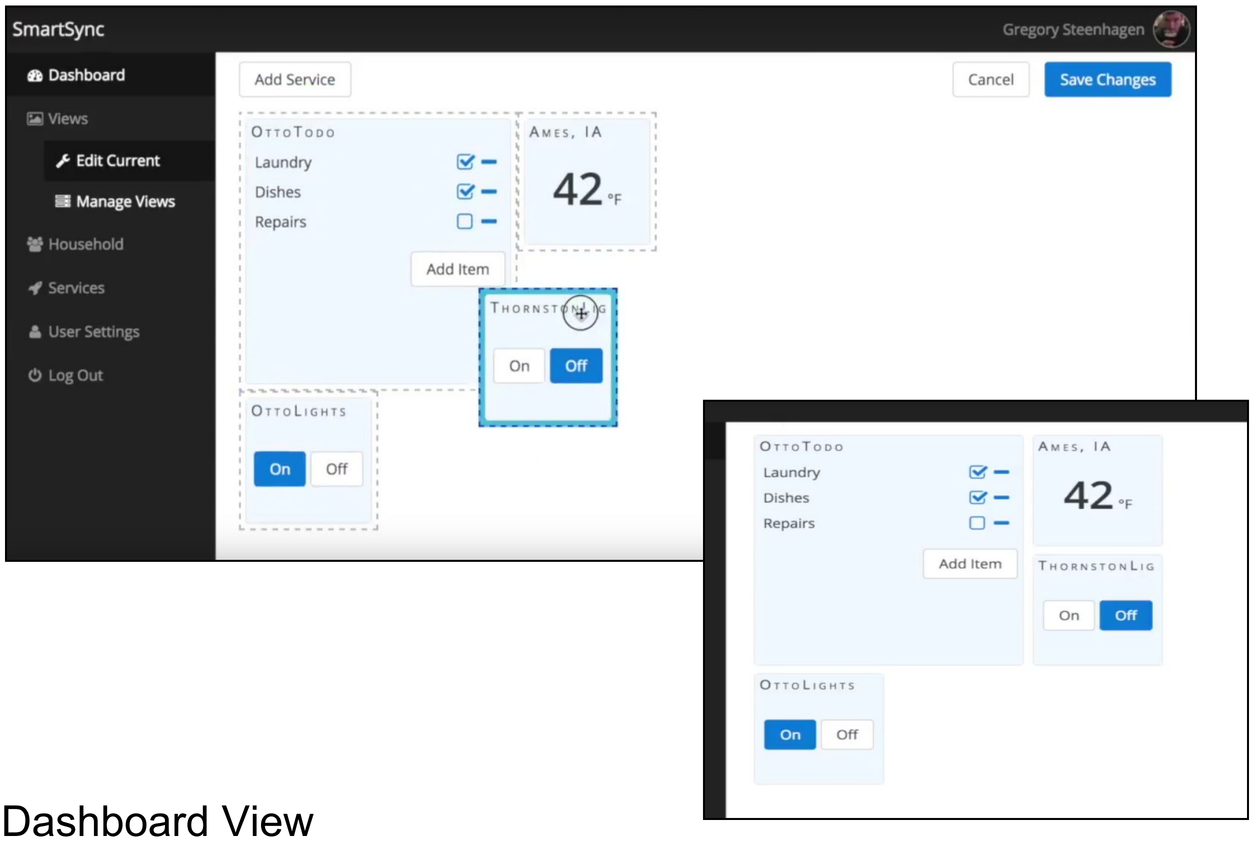
State: Missouri - MO

Zip: 55

Household name must be greater than 4 characters.

Zip codes must be five digits in length.

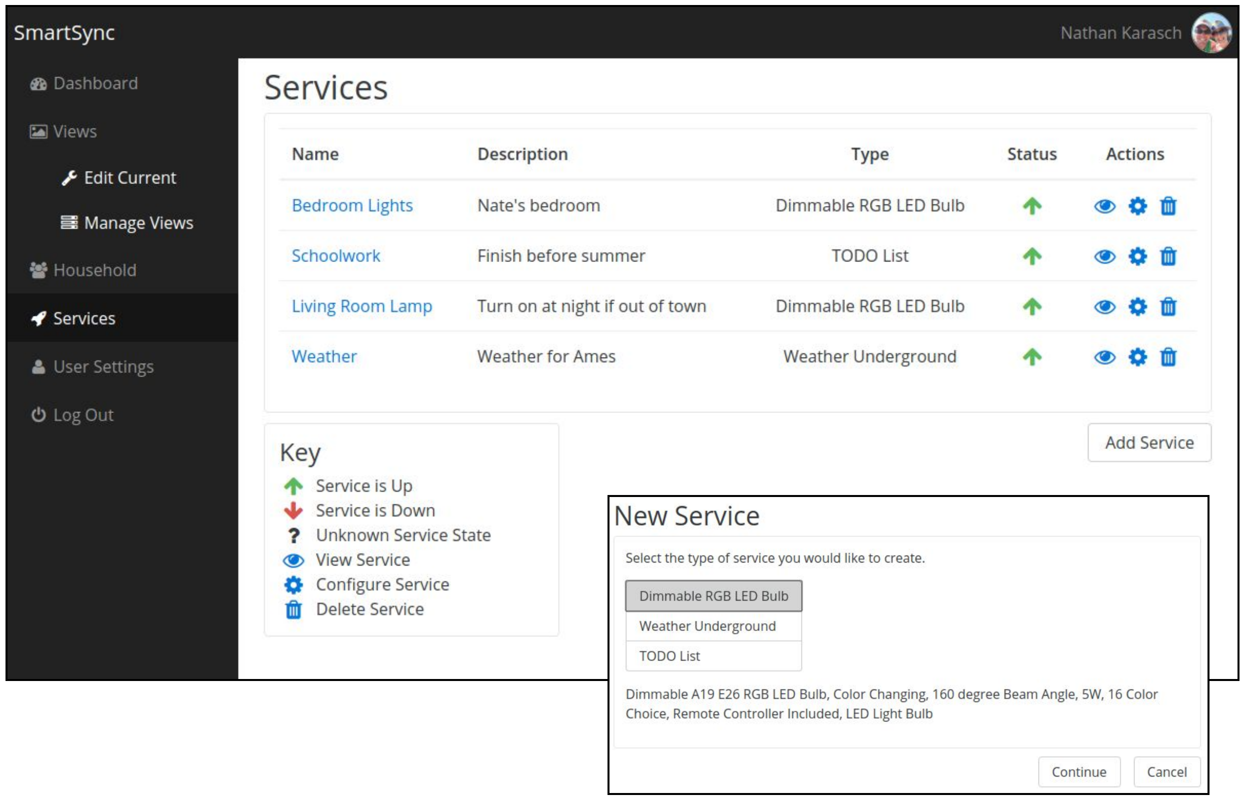
Invalid form!



Dashboard View

The main user interface is the Dashboard View, which lets the user view and interact with the different services in the household. For customization, the user can add more services and rearrange the tiles to fit his or her own preferences. The dashboard settings are linked to the user's account, so that each time they open SmartSync, they see the same dashboard that they customized before.

The JavaScript library **packery** was used for tile packing, and the **draggabilly** library was used for dragging/dropping the tiles to rearrange them.



Services

Name	Description	Type	Status	Actions
Bedroom Lights	Nate's bedroom	Dimmable RGB LED Bulb	↑	👁 ⚙ 🗑
Schoolwork	Finish before summer	TODO List	↑	👁 ⚙ 🗑
Living Room Lamp	Turn on at night if out of town	Dimmable RGB LED Bulb	↑	👁 ⚙ 🗑
Weather	Weather for Ames	Weather Underground	↑	👁 ⚙ 🗑

Key

↑	Service is Up
↓	Service is Down
?	Unknown Service State
👁	View Service
⚙	Configure Service
🗑	Delete Service

New Service

Select the type of service you would like to create.

- Dimmable RGB LED Bulb
- Weather Underground
- TODO List

Dimmable A19 E26 RGB LED Bulb, Color Changing, 160 degree Beam Angle, 5W, 16 Color Choice, Remote Controller Included, LED Light Bulb

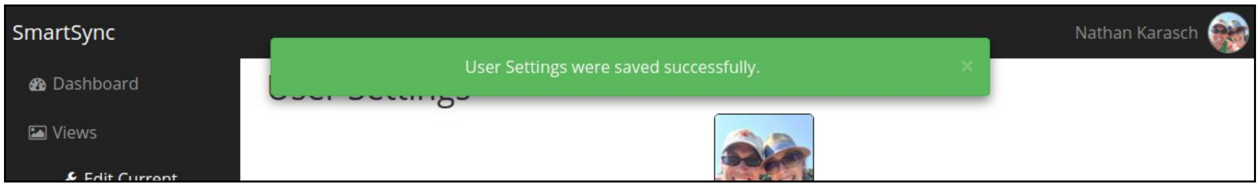
Service Management

Every user has the ability to view the services in the household, but the admin user has the privilege to add, remove, and configure services. The services presented the biggest architectural problem that we had to solve, since a system like this needs to be able to handle a multitude of service types, each with their own unique qualities and attributes.

On the front end, Angular2 uses TypeScript, which allows classes and class extension for your components. There is a component for the generic service, which handles UI features common to all services (the bordered tile box, the functionality to move around the dashboard, etc.). The concrete service instances then have their own view and controller code specific to the service type. So, for example, a lightbulb service has an On/Off switch, a TODO list has an editable list of TODO items, and so on down the line for each new service.

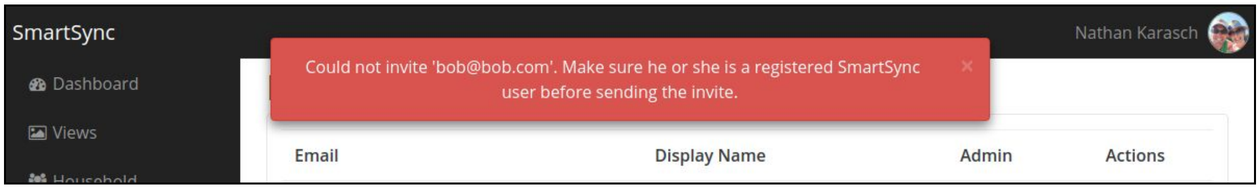
The back end then handles each service by spinning up a new server within the microservice architecture. This will be discussed later in the report.

Other Features


SmartSync

Nathan Karasch

User Settings were saved successfully.

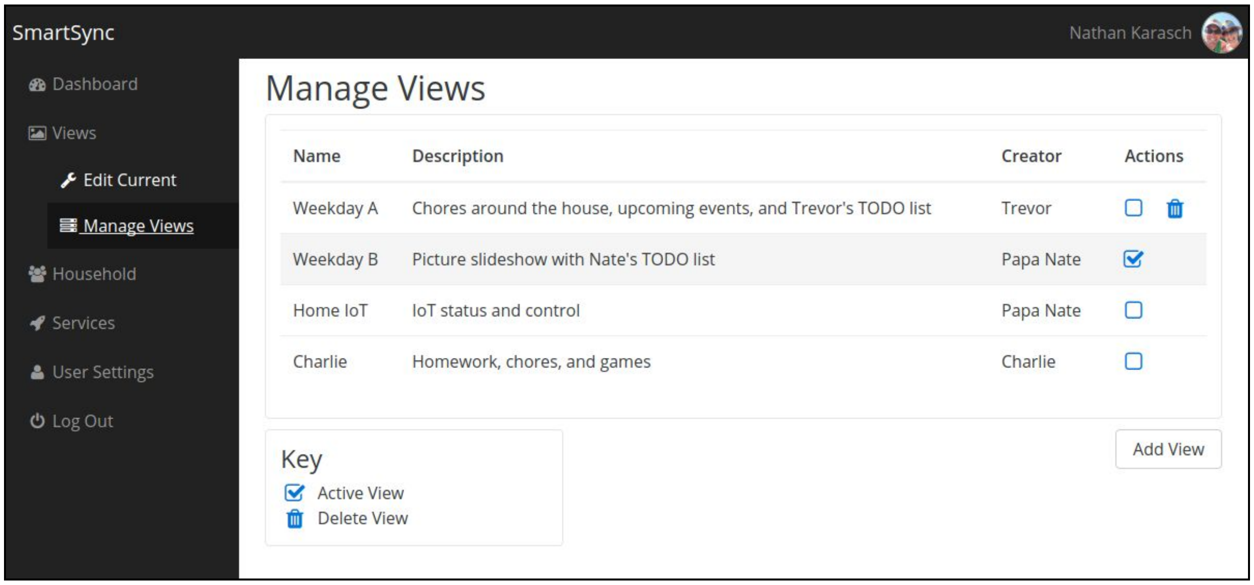

SmartSync

Nathan Karasch

Could not invite 'bob@bob.com'. Make sure he or she is a registered SmartSync user before sending the invite.

Email	Display Name	Admin	Actions

Since SmartSync operates as a single-page application using lots of AJAX calls, it was important to have a method of informing the user when operations succeeded or failed. A small alert API was written to make it easy for any part of the application to post an alert to the top of the page. The alerts disappear after a certain timeout, or the user can click the X in the corner to close them. When more than one alert is displayed, they are stacked vertically, with the older alerts pushed down to make room for newer alerts at the top.


SmartSync

Nathan Karasch

Manage Views

Name	Description	Creator	Actions
Weekday A	Chores around the house, upcoming events, and Trevor's TODO list	Trevor	☐ 🗑
Weekday B	Picture slideshow with Nate's TODO list	Papa Nate	✓
Home IoT	IoT status and control	Papa Nate	☐
Charlie	Homework, chores, and games	Charlie	☐

Key
✓ Active View
🗑 Delete View

We had originally planned to let each user create and manage multiple dashboard views. The UI was written to handle the view management; however, we didn't have time to fully implement and integrate the backend aspects of this feature.

Quality Attributes

Architectural Changes

Since we used a microservice architecture for the backend of SmartSync, there wasn't a need for large architectural changes during the implementation of the security requirements. The security implementation was accomplished by simply creating an authentication service and adding it to the communication utility library so that it could communicate with the services that were exposed to the outside world. Thus the only architectural change needed for the implementation of the security requirements was the addition of a service component and the modification of the service components that are used to access user information.

In order to meet the response times we initially had hoped for we would have needed more powerful hardware than what was at our disposal. However, by adjusting the amount of memory that the JAR files used, we were able to allow for faster communication from client to server and subsequently reach a response time closer to our goal. This adjustment landed us on the seven second mark as listed in our quality attributes and thus changed our deployment process slightly.

Acknowledge Patterns

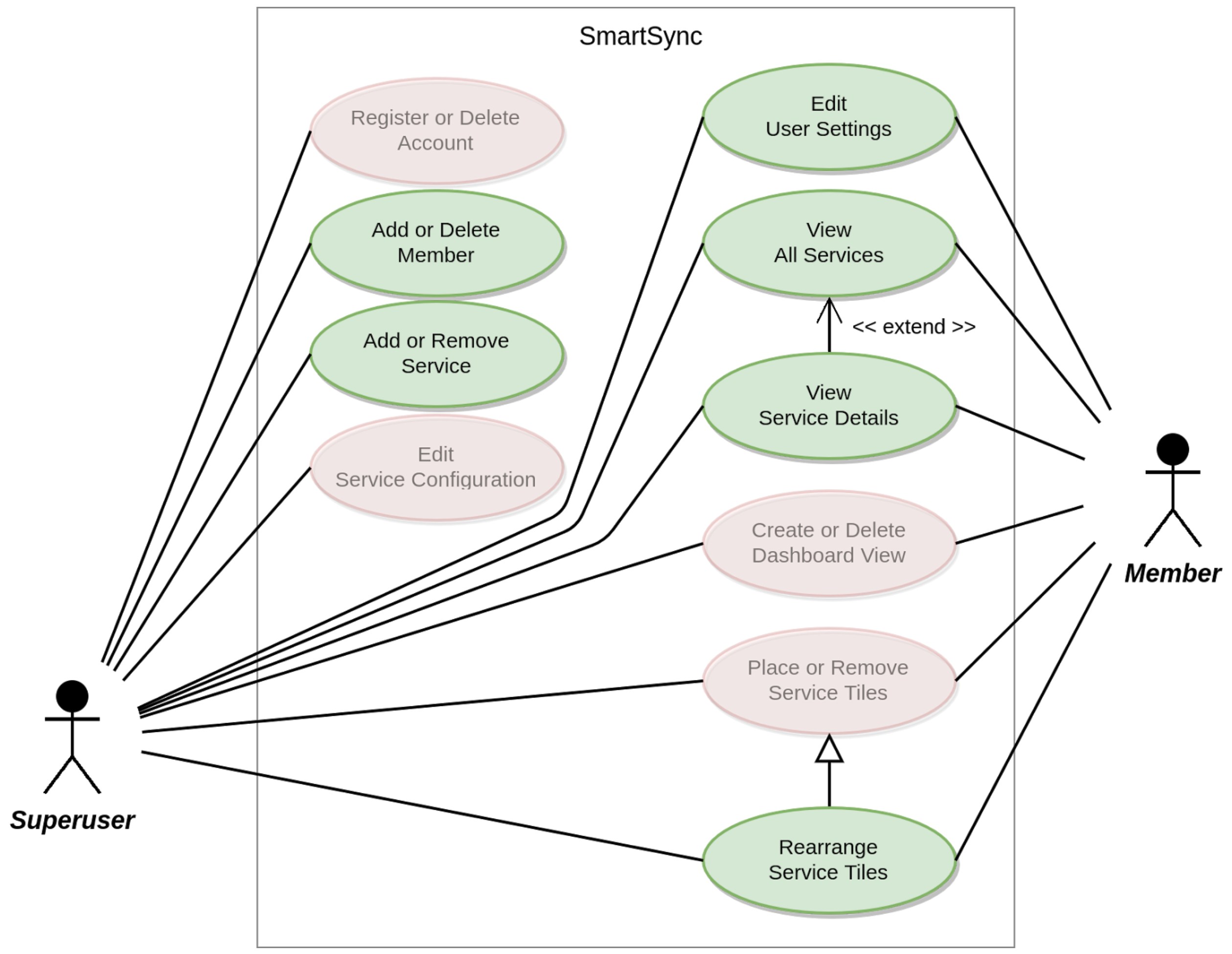
A pattern that was prevalent with our project, along with a microservice architecture in general, was the way that new functionalities are added to the project. When a new functionality is needed such as Authentication or(in the case of our project) a new household smart device, you must create an entirely new service on the backend. This has numerous positive aspects, for example, when a someone needed to add a new service on the backend it wasn't necessary to dig through someone else's code and try to understand it. The notable negative aspect of this emergent pattern came from the lack of resources at our disposal. Since we only had one server and limited memory to host all of our services, it took a long time for the server to start up as well as limited the overall responsiveness of the application.

As we began to deploy the application we noticed that the microservice architecture was very resource intensive. This became particularly apparent when we deployed to the server for the first time and the response times were so slow that some of the HTTP requests would time out. To mitigate this we changed the amount of memory that each service was using and the HTTP requests stopped timing out. This brought us to the conclusion that the ideal deployment environment for the application would have involved an individual server for each service. This would grant optimal response times for client to server requests as well as ensure that each service would have sufficient sufficient resources to function.

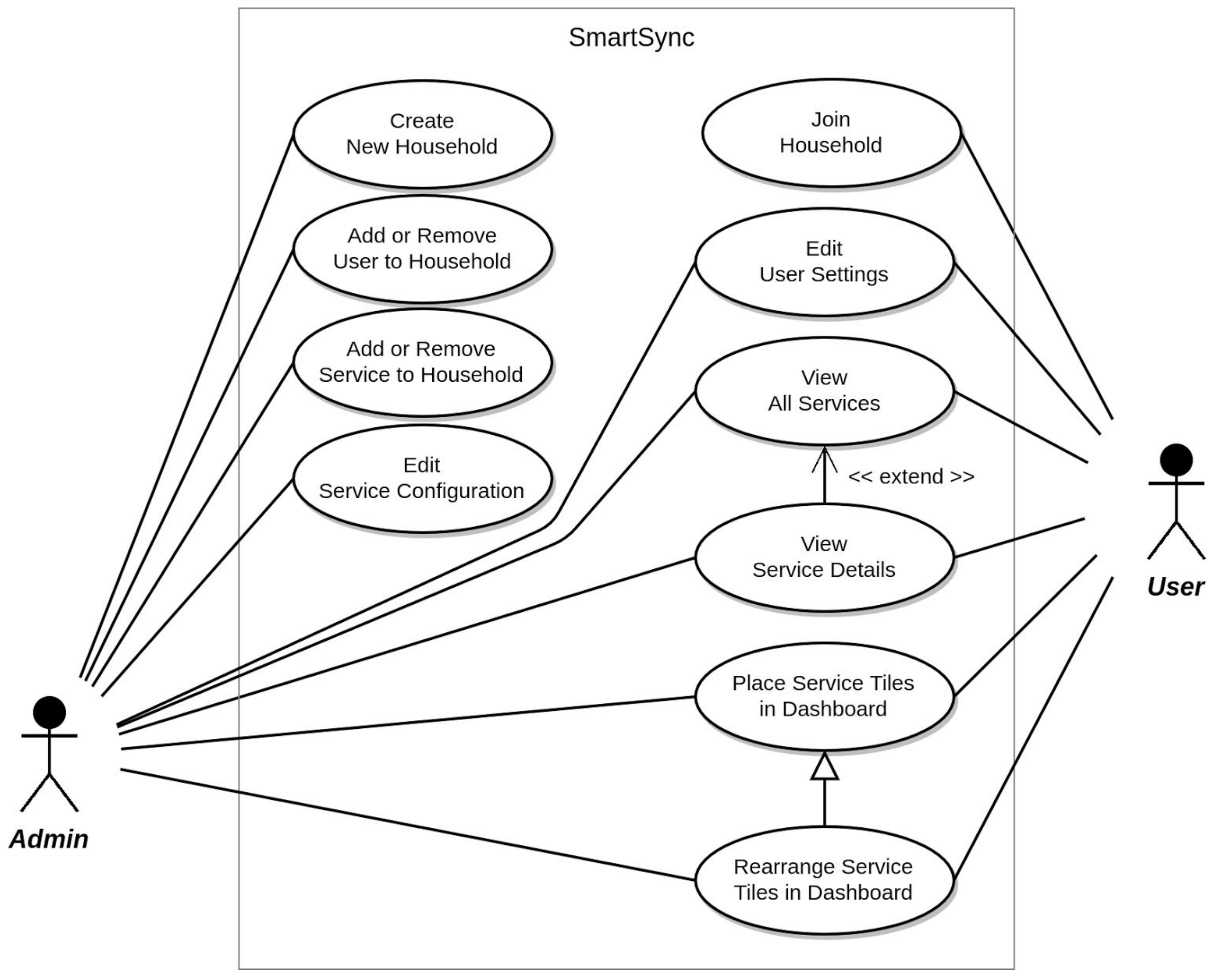
Architecture Evolution

In this phase we finished all of the security requirements that we presented in phase 3; including securing the API endpoints from unauthorized users and ensuring that users can only access their information; the ability to add, delete, and view all services; the ability to interact with a basic service (lightbulb); and the ability to join a household. We also fixed small bugs that had to do with communicating between microservices and redirection issues from login to the main application. Below, there is the updated use case, component, deployment, and class diagrams. After, there is a discussion talking about why we made certain changes, how these changes impacted our architecture, and how they impacted security and performance of the application.

Phase 3 Use Case Diagram



Updated Use Case Diagram

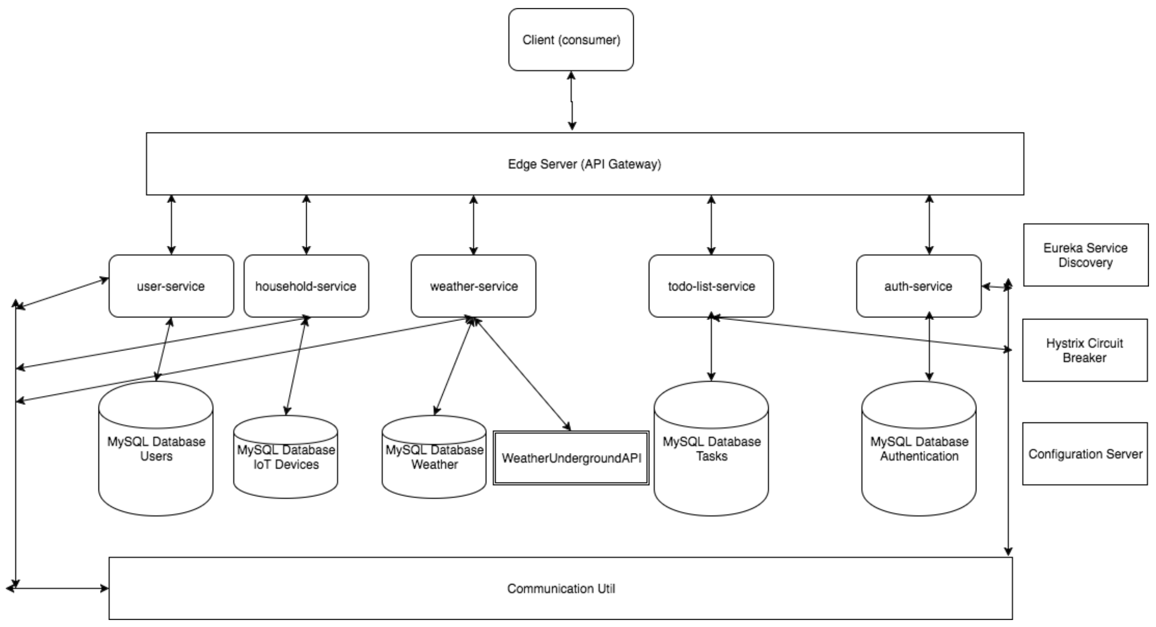

SmartSync
Create New Household
Add or Remove User to Household
Add or Remove Service to Household
Edit Service Configuration
Join Household
Edit User Settings
View All Services
<< extend >>
View Service Details
Place Service Tiles in Dashboard
Rearrange Service Tiles in Dashboard
Admin
User

The main differences in the use case diagram from Phase 3 to Phase 4 are as follows:

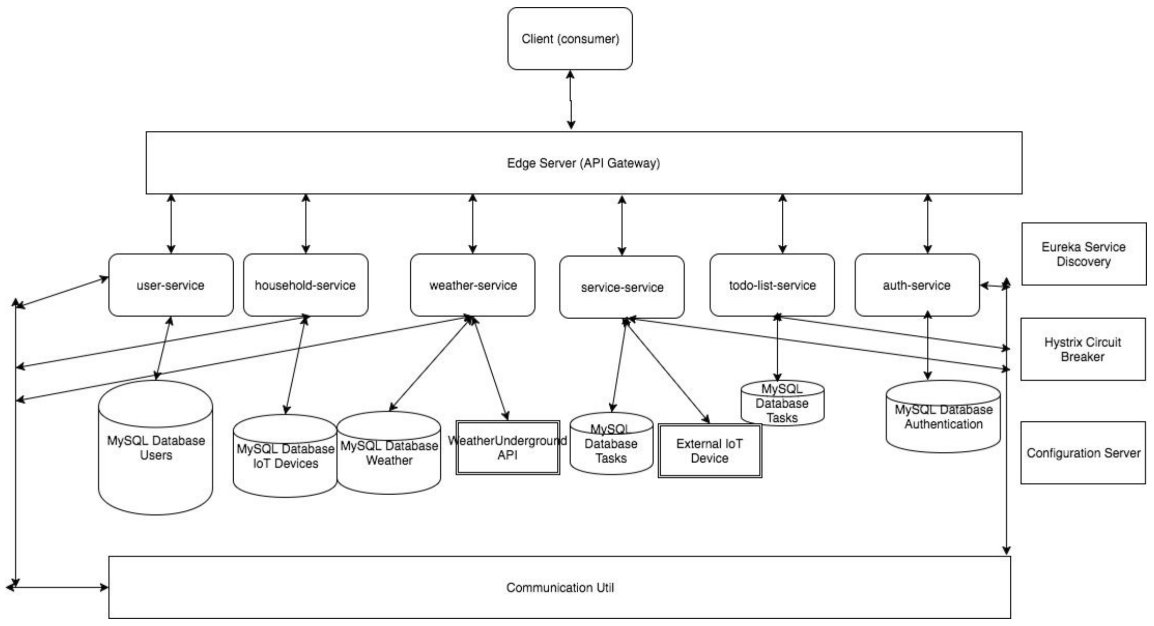
- Removed:
    - Delete Account (Household)
    - Create or Delete Dashboard View
- Added:
    - Join Household

The changes did not dramatically impact the architecture. The removed use cases are ones we didn't have time to implement and weren't critical to the overall project. If this project were to have continued longer, most of the structures are already in place. Implementing or leaving them out doesn't change the architecture, performance, or security mechanisms we implemented. The "Join Household" use case that was added to the diagram had been part of our implicit architecture already, but wasn't fully documented, so its impact was minimal.

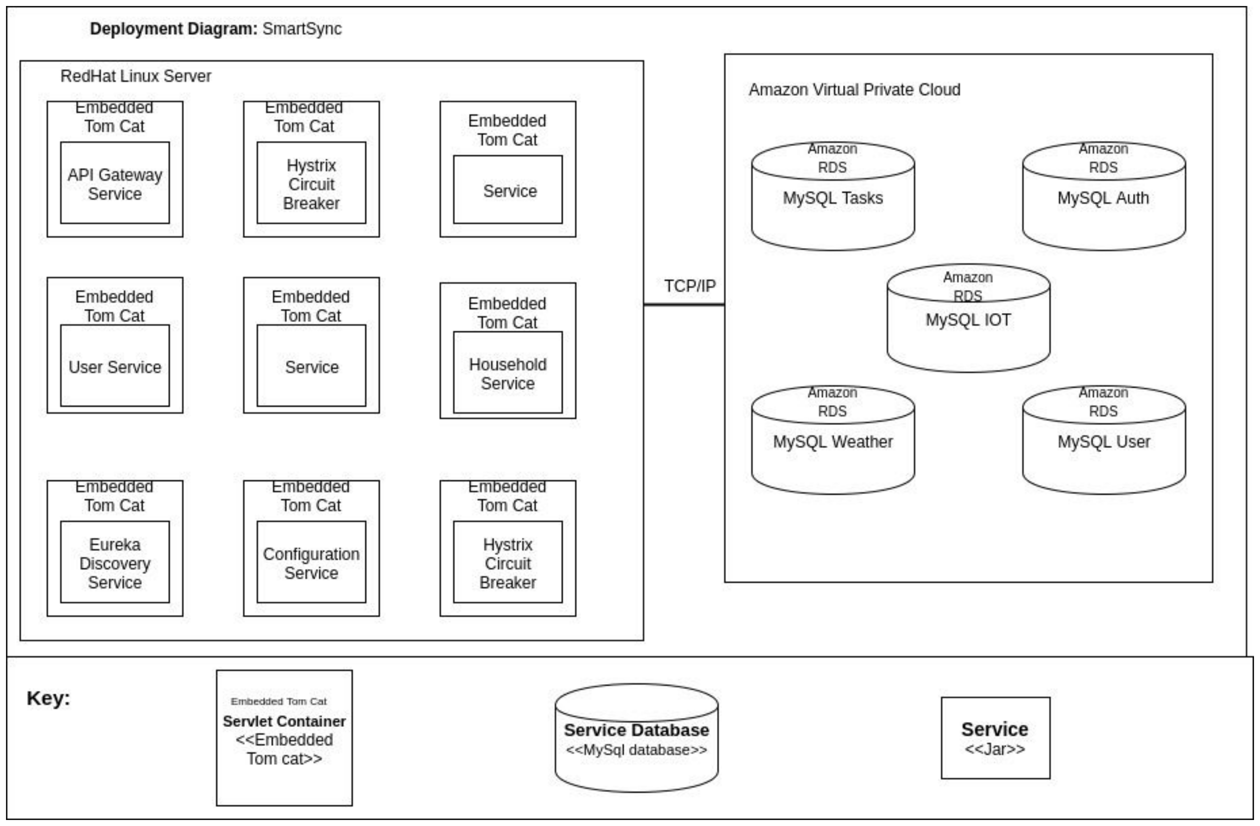
Phase 3 Component Diagram



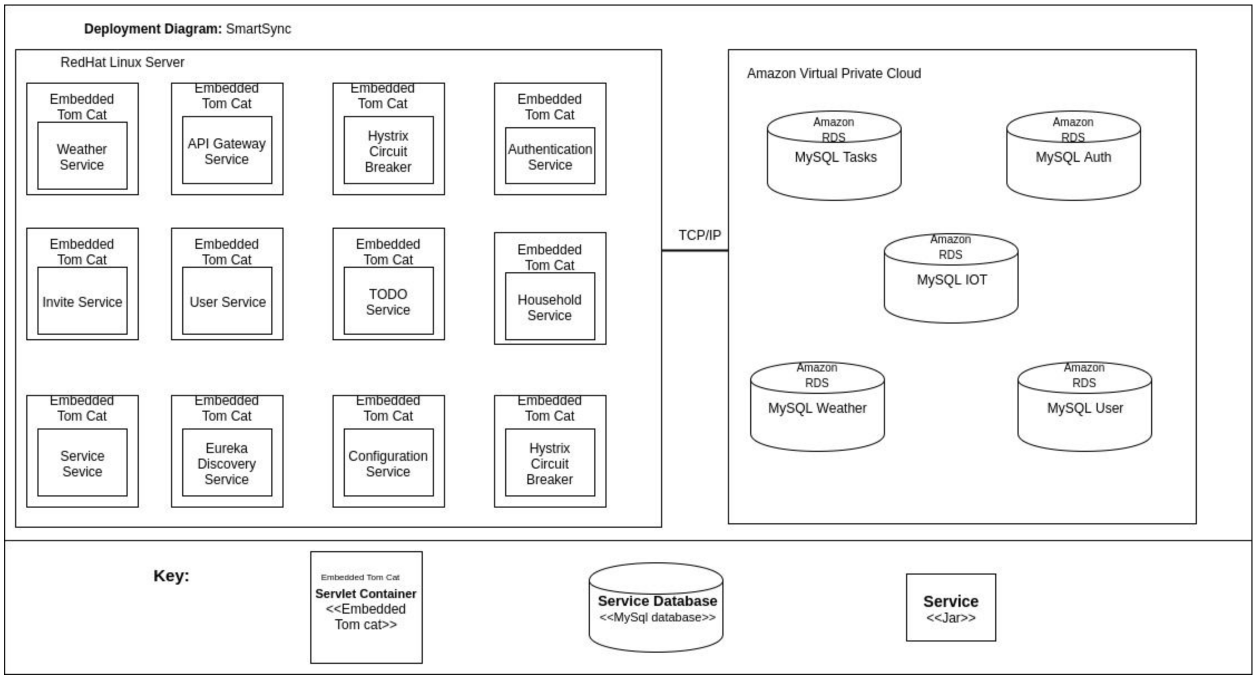
Updated Component Diagram



Phase 3 Deployment Diagram

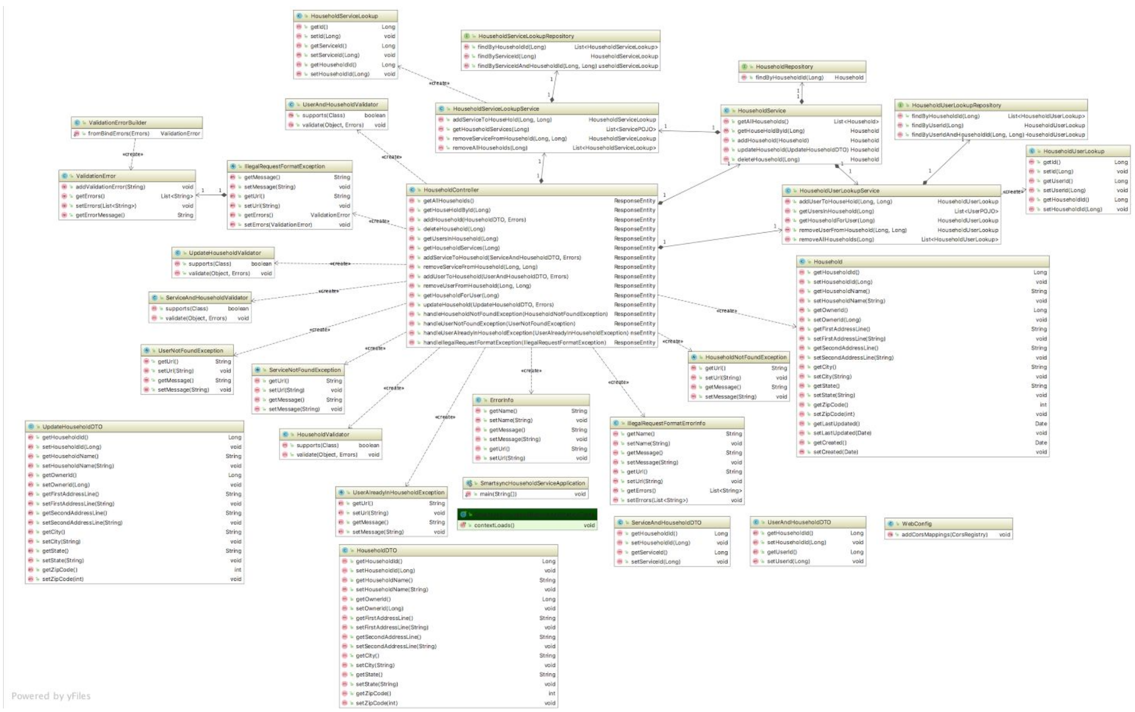


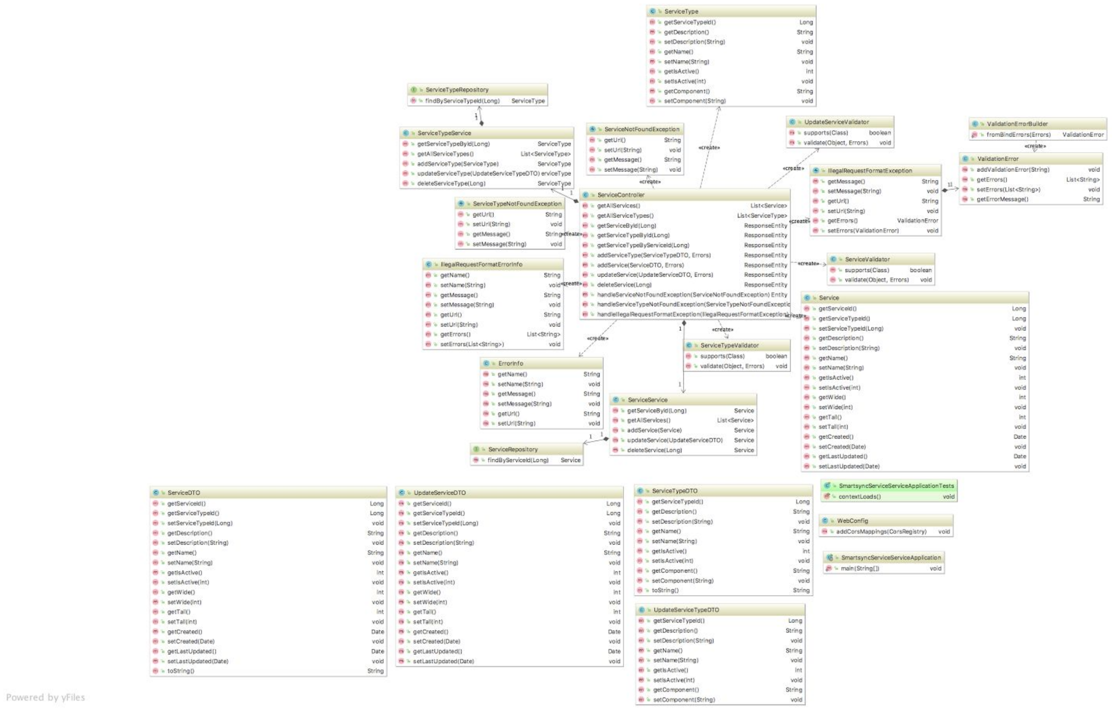
Updated Deployment Diagram

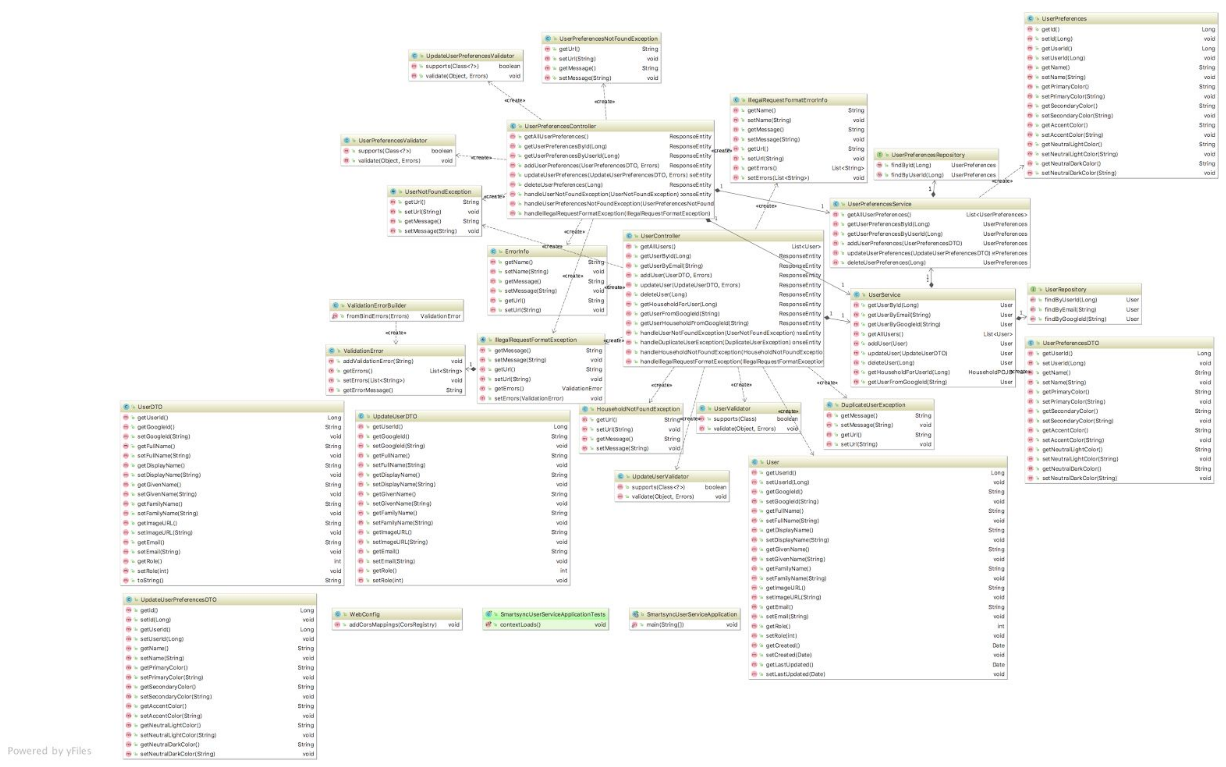


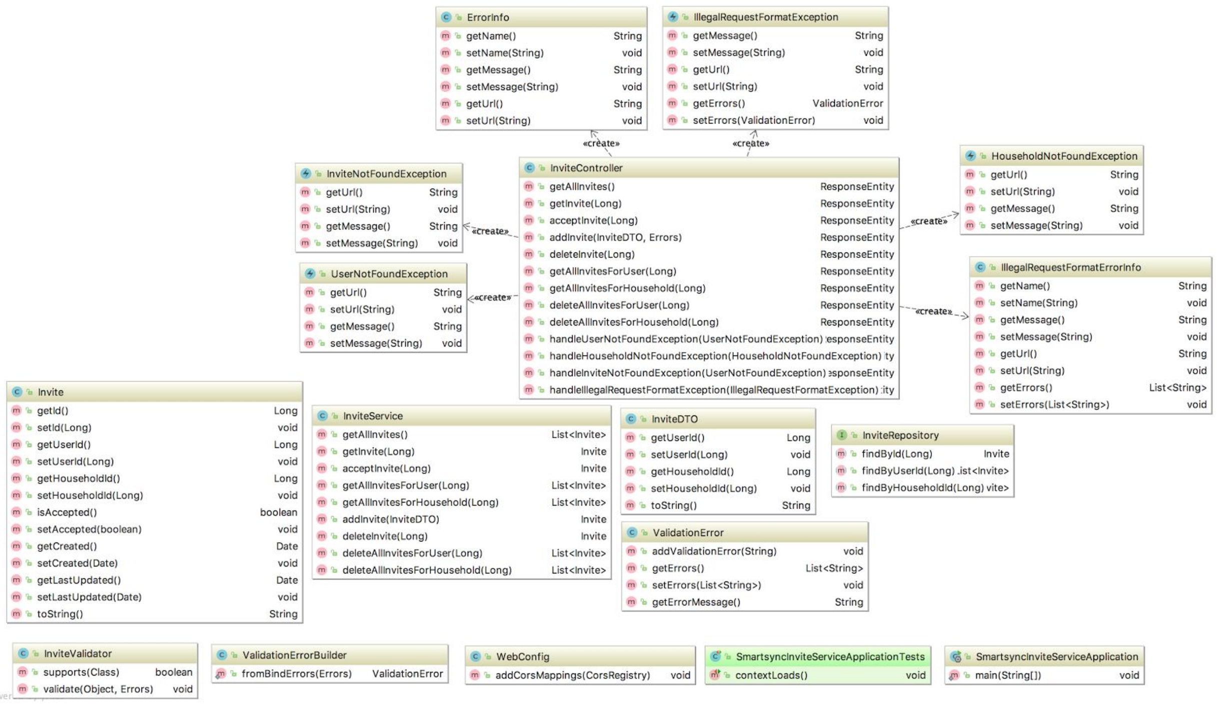
Updated Class Diagrams

Server Side

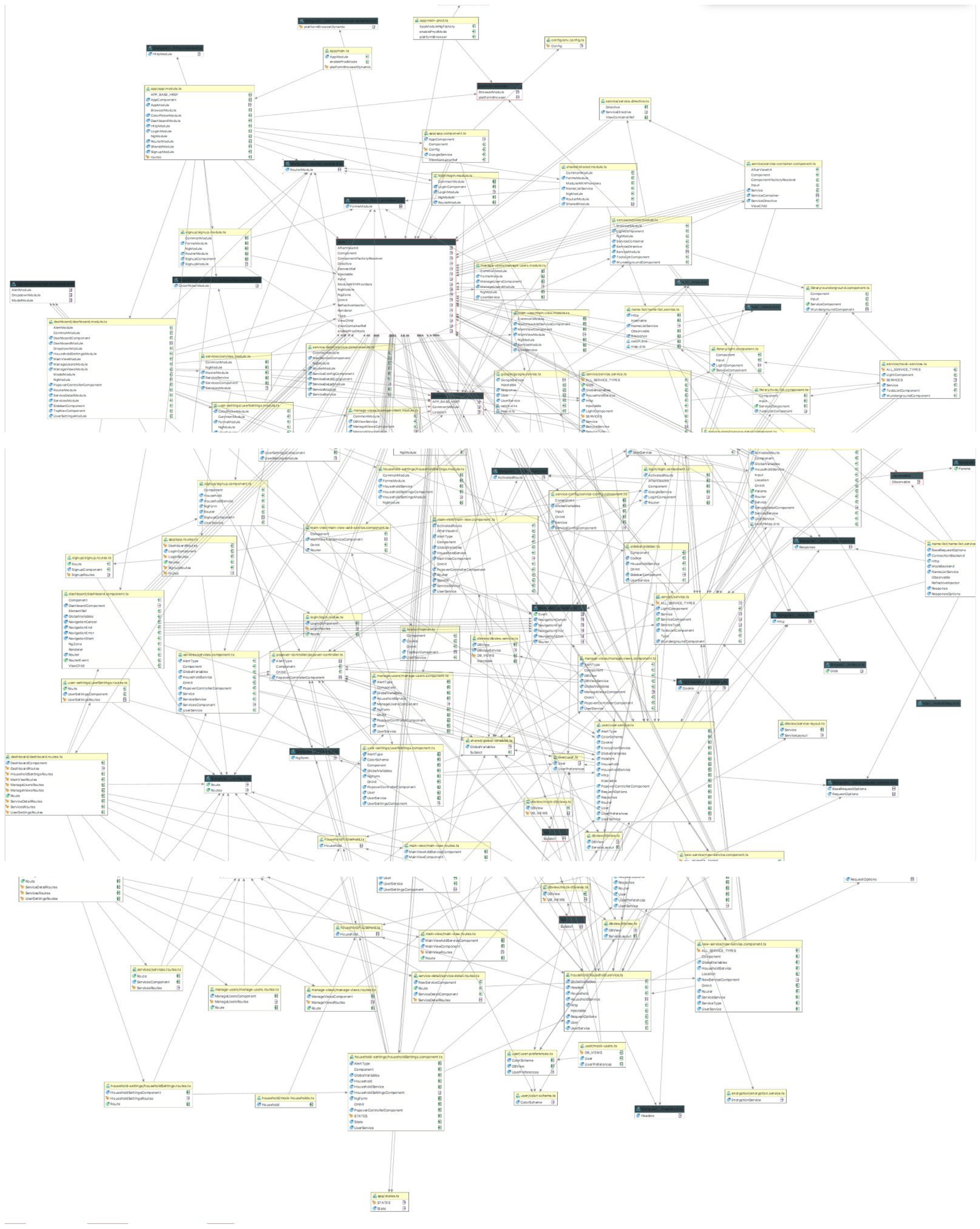





UserPreferences
getId() Long
setId(Long) void
getUserId() Long
setUserId(Long) void
getName() String
setName(String) void
getPrimaryColor() String
setPrimaryColor(String) void
getSecondaryColor() String
setSecondaryColor(String) void
getAccentColor() String
setAccentColor(String) void
getNeutralLightColor() String
setNeutralLightColor(String) void
getNeutralDarkColor() String
setNeutralDarkColor(String) void
UserPreferencesNotFoundException
getUrl() String
setUrl(String) void
getMessage() String
setMessage(String) void
UpdateUserPreferencesValidator
supports(Class<?>) boolean
validate(Object, Errors) void
IllegalRequestFormatErrorInfo
getName() String
setName(String) void
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
getErrors() List<String>
setErrors(List<String>) void
UserPreferencesValidator
supports(Class<?>) boolean
validate(Object, Errors) void
UserPreferencesController
getAllUserPreferences() ResponseEntity
getUserPreferencesById(Long) ResponseEntity
getUserPreferencesByUserId(Long) ResponseEntity
addUserPreferences(UserPreferencesDTO, Errors) ResponseEntity
updateUserPreferences(UpdateUserPreferencesDTO, Errors) ResponseEntity
deleteUserPreferences(Long) ResponseEntity
handleUserNotFoundException(UserNotFoundException) ResponseEntity
handleUserPreferencesNotFoundException(UserPreferencesNotFound
handleIllegalRequestFormatException(IllegalRequestFormatException)
UserPreferencesRepository
findById(Long) UserPreferences
findByUserId(Long) UserPreferences
UserNotFoundException
getUrl() String
setUrl(String) void
getMessage() String
setMessage(String) void
UserPreferencesService
getAllUserPreferences() List<UserPreferences>
getUserPreferencesById(Long) UserPreferences
getUserPreferencesByUserId(Long) UserPreferences
addUserPreferences(UserPreferencesDTO) UserPreferences
updateUserPreferences(UpdateUserPreferencesDTO) UserPreferences
deleteUserPreferences(Long) UserPreferences
ErrorInfo
getName() String
setName(String) void
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
UserController
getAllUsers() List<User>
getUserById(Long) ResponseEntity
getUserByEmail(String) ResponseEntity
addUser(UserDTO, Errors) ResponseEntity
updateUser(UpdateUserDTO, Errors) ResponseEntity
deleteUser(Long) ResponseEntity
getHouseholdForUser(Long) ResponseEntity
getUserFromGoogleId(String) ResponseEntity
getUserHouseholdFromGoogleId(String) ResponseEntity
handleUserNotFoundException(UserNotFoundException) ResponseEntity
handleDuplicateUserException(DuplicateUserException) ResponseEntity
handleHouseholdNotFoundException(HouseholdNotFoundException)
handleIllegalRequestFormatException(IllegalRequestFormatException)
UserService
getUserById(Long) User
getUserByEmail(String) User
getUserByGoogleId(String) User
getAllUsers() List<User>
addUser(User) User
updateUser(UpdateUserDTO) User
deleteUser(Long) User
getHouseholdForUserId(Long) HouseholdPOJO
getUserFromGoogleId(String) User
UserRepository
findByUserId(Long) User
findByEmail(String) User
findByGoogleId(String) User
ValidationErrorBuilder
fromBindErrors(Errors) ValidationError
ValidationError
addValidationError(String) void
getErrors() List<String>
setErrors(List<String>) void
getErrorMessage() String
IllegalRequestFormatException
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
getErrors() ValidationError
setErrors(ValidationError) void
UserPreferencesDTO
getUserId() Long
setUserId(Long) void
getName() String
setName(String) void
getPrimaryColor() String
setPrimaryColor(String) void
getSecondaryColor() String
setSecondaryColor(String) void
getAccentColor() String
setAccentColor(String) void
getNeutralLightColor() String
setNeutralLightColor(String) void
getNeutralDarkColor() String
setNeutralDarkColor(String) void
HouseholdNotFoundException
getUrl() String
setUrl(String) void
getMessage() String
setMessage(String) void
UserValidator
supports(Class) boolean
validate(Object, Errors) void
DuplicateUserException
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
UserDTO
getUserId() Long
getGoogleId() String
setGoogleId(String) void
getFullName() String
setFullName(String) void
getDisplayName() String
setDisplayName(String) void
getGivenName() String
setGivenName(String) void
getFamilyName() String
setFamilyName(String) void
getImageURL() String
setImageURL(String) void
getEmail() String
setEmail(String) void
getRole() int
setRole(int) void
toString() String
UpdateUserDTO
getUserId() Long
getGoogleId() String
setGoogleId(String) void
getFullName() String
setFullName(String) void
getDisplayName() String
setDisplayName(String) void
getGivenName() String
setGivenName(String) void
getFamilyName() String
setFamilyName(String) void
getImageURL() String
setImageURL(String) void
getEmail() String
setEmail(String) void
getRole() int
setRole(int) void
UpdateUserValidator
supports(Class<?>) boolean
validate(Object, Errors) void
User
getUserId() Long
setUserId(Long) void
getGoogleId() String
setGoogleId(String) void
getFullName() String
setFullName(String) void
getDisplayName() String
setDisplayName(String) void
getGivenName() String
setGivenName(String) void
getFamilyName() String
setFamilyName(String) void
getImageURL() String
setImageURL(String) void
getEmail() String
setEmail(String) void
getRole() int
setRole(int) void
getCreated() Date
setCreated(Date) void
getLastUpdated() Date
setLastUpdated(Date) void
UpdateUserPreferencesDTO
getId() Long
setId(Long) void
getUserId() Long
setUserId(Long) void
getName() String
setName(String) void
getPrimaryColor() String
setPrimaryColor(String) void
getSecondaryColor() String
setSecondaryColor(String) void
getAccentColor() String
setAccentColor(String) void
getNeutralLightColor() String
setNeutralLightColor(String) void
getNeutralDarkColor() String
setNeutralDarkColor(String) void
WebConfig
addCorsMappings(CorsRegistry) void
SmartsyncUserServiceApplicationTests
contextLoads() void
SmartsyncUserServiceApplication
main(String[]) void
«create»
Powered by yFiles


ErrorInfo
getName() String
setName(String) void
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
IllegalRequestFormatException
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
getErrors() ValidationError
setErrors(ValidationError) void
«create»
«create»
InviteNotFoundException
getUrl() String
setUrl(String) void
getMessage() String
setMessage(String) void
InviteController
getAllInvites() ResponseEntity
getInvite(Long) ResponseEntity
acceptInvite(Long) ResponseEntity
addInvite(InviteDTO, Errors) ResponseEntity
deleteInvite(Long) ResponseEntity
getAllInvitesForUser(Long) ResponseEntity
getAllInvitesForHousehold(Long) ResponseEntity
deleteAllInvitesForUser(Long) ResponseEntity
deleteAllInvitesForHousehold(Long) ResponseEntity
handleUserNotFoundException(UserNotFoundException) ResponseEntity
handleHouseholdNotFoundException(HouseholdNotFoundException) ResponseEntity
handleInviteNotFoundException(InviteNotFoundException) ResponseEntity
handleIllegalRequestFormatException(IllegalRequestFormatException) ResponseEntity
HouseholdNotFoundException
getUrl() String
setUrl(String) void
getMessage() String
setMessage(String) void
«create»
«create»
«create»
«create»
UserNotFoundException
getUrl() String
setUrl(String) void
getMessage() String
setMessage(String) void
IllegalRequestFormatErrorInfo
getName() String
setName(String) void
getMessage() String
setMessage(String) void
getUrl() String
setUrl(String) void
getErrors() List<String>
setErrors(List<String>) void
Invite
getId() Long
setId(Long) void
getUserId() Long
setUserId(Long) void
getHouseholdId() Long
setHouseholdId(Long) void
isAccepted() boolean
setAccepted(boolean) void
getCreated() Date
setCreated(Date) void
getLastUpdated() Date
setLastUpdated(Date) void
toString() String
InviteService
getAllInvites() List<Invite>
getInvite(Long) Invite
acceptInvite(Long) Invite
getAllInvitesForUser(Long) List<Invite>
getAllInvitesForHousehold(Long) List<Invite>
addInvite(InviteDTO) Invite
deleteInvite(Long) Invite
deleteAllInvitesForUser(Long) List<Invite>
deleteAllInvitesForHousehold(Long) List<Invite>
InviteDTO
getUserId() Long
setUserId(Long) void
getHouseholdId() Long
setHouseholdId(Long) void
toString() String
InviteRepository
findById(Long) Invite
findByUserId(Long) .ist<Invite>
findByHouseholdId(Long) vite>
ValidationError
addValidationError(String) void
getErrors() List<String>
setErrors(List<String>) void
getErrorMessage() String
InviteValidator
supports(Class) boolean
validate(Object, Errors) void
ValidationErrorBuilder
fromBindErrors(Errors) ValidationError
WebConfig
addCorsMappings(CorsRegistry) void
SmartsyncInviteServiceApplicationTests
contextLoads() void
SmartsyncInviteServiceApplication
main(String[]) void
Powered by yFiles

Client Side



Discussion

This section will be a discussion about the architecturally impacting changes in relation to the new use cases we implemented. For each use case, we will attempt to answer the following questions:

- What are the impacts of these use cases on the architecture?
- Did these changes impact the performance and security mechanisms that you implemented

The sections below are listed to in order from largest architecture change, to smallest architecture change

Add, Delete, View Services

In phase 4 of this project, we implemented the ability to add, delete, edit, and view different services. There is a few different use cases in play in this section. As an application, we can support different types of services, for the scope of this project we chose to do a weather service, to-do-list service, and a lightbulb service (that simulates a smart lightbulb). The weather service used the Weather Underground API, so we made a wrapper class that makes calls to their service. The to-do-list service was written by us, and can include to-do-lists for a household and for specific users. Finally, the lightbulb service is just a python script that simulates a smart lightbulb. The communication to this lightbulb is done over http, as the light bulb simulator runs a very simple web server. The to-do-list service and weather service were already completed, so it did not impact the architecture for phase 4. The new part for phase 4 was the addition of the service services, which included the connection to the light bulb. In order to accomplish this, we added a new service which would interact with any smart device that we told it to. This new service can be seen in the component diagram as “Service Service”. As the diagram says, we needed to add new service, new database table, and then right the interactions between the our server and the smart services we connected to.

The addition of the service service required some new security constraints as well. Just like the other services, we needed to ensure that users couldn’t access data that wasn’t theirs. So we verified which data they could see by their session token. We’ll talk more about our security process in the next section. Also, in a production level application, securing the IoT device would have been a priority. It would be very important to create a strong firewall between the device and our server to make sure that if the device is compromised, it does not also compromise our server. We believed that this was outside the scope of the class, so this was not implemented in our project. With the addition of the new service, we again slowed our server down by adding another process that needs to be run. In the real world, however, this would not be an issue, as we would have more resources at our disposal. The only other performance concern was getting data quickly from the smart device. Our server will get the data as fast as the HTTP request will allow us to.

Secure Endpoints and Data

In this phase, we also secured all of our endpoints and data. This required some major architecture changes. We wanted to make sure that only authenticated users could have access to our server and that only users with the correct privileges could see the data that they asked for. In order to accomplish this, we needed to add, yet again, another service to our application. This service is called the auth-service. This service is responsible for keeping track of the session tokens. There are three endpoints in this service. The login endpoint, which exchanges a google id for a session token. The authenticated endpoint, which checks if the session token is valid and if it is associated with a user. Finally, the logout endpoint, which kills the session by deleting the session token. In every other server request to our services, we make a call to the authentication service to make sure the user is authenticated. For a request which the user is making a request to get some data, we make sure that the session token is associated with the user that owns the data. For example, Jim cannot access Bob's data because Jim's session token is associated with his user id, not Bob's.

The addition of the new authentication service had some performance issues associated with it. Other than our server having to handle yet another service, each request to our server required a call to the authentication service. This added a lot of overhead to each request to our server, making it much slower. Instead of one HTTP request per request to our server, now we have two HTTP requests. This had an impact on the users as it slowed down our application. This is an important tradeoff to consider, speed vs. security. Security is a very important aspect to any web application, so it's a required trade off.

Our original plan was to do everything security related in the API gateway. In this situation, the API gateway would almost act like a firewall to our server. If the user was not authenticated for the request, they would never get to our microservices. However, we lacked the knowledge and time to accomplish this. Instead, we took a very naive approach to the security of our application. This was our reasoning to taking the naive solution, since we lacked knowledge and time, we choose a solution that worked, not necessarily the best one.

Invite, Join, Create a Household

These use cases did not tremendously impact the architecture, since we had planned on them from the beginning of the project. It was simply a matter of completing their integration and implementation.

Architecture Recovery

To accomplish the architecture recovery of our project, we did the following steps:

1. Downloaded the java-callgraph project and built the jar file for that.
2. For each one of our microservices, build the most recent jar file, and passed that built jar file into the java-callgraph program.
3. After making the java-callgraph, cleaned up the file by removing unnecessary graph edges. These edges Spring Boot Jar files and maven build stuff. This was not necessary to look that the architecture of our project.
4. Downloaded the Bunch tool and built that jar for that project
5. For each one of our microservices, take the java-callgraph output and put it into the Bunch tool and exported the output from that tool as a .dot file.
6. For each .dot file, we removed the "rotate" line and increased the size of the digraph to 100x100. Then the following command was executed in the command line to create the SVG files: **`fdp -Tsvg <filename>.dot -O`** (requires the graphviz package to be installed)

After doing the architecture recovery on our project, it became clear that our intended architecture was indeed the architecture that was made. Of course, it is difficult to tell if we actually built a microservice architecture, but the fact that we built several different applications, and they can all communicate in unison, tells me that we did. I found it interesting how the clustering of each individual project worked out. It is interesting that you can see each individual use case in the clusters. For example, if the household service recovery, cluster 2, it is very clearly our update households use case. This has methods which updates households and has all of the error validation in it. It is also interesting to see all of the technologies that are used in a cluster.

It is also very interesting to see all of the edges between the different parts of our application. The graph would be much more interesting if all our services were in was massive jar file, so the call graph would be one big call graph. But since we did the microservices architecture, each one of our services have their own call graph. But, still, it is cool to see the edges between the different parts and how each cluster of the application interacts with one another. For instance, in the communication util recovery, we can really see how the application interacts as a whole. In each of the clusters it is very clear what it is doing. For example, cluster one represents the communication between the households and the users and cluster two is the communication between the households, users, and services.

Through the architecture recovery process, we've learned that it's very difficult to do this task. The call graph produces a lot of "noise" which is includes information about the inner workings of the libraries that are being used in the project. While looking at the architecture of your own

project, some of these inner working are unimportant and only bloats the final graph output. One of the challenges is going through and eliminating some of this noise. To view the architecture recovery images, view the following link. The graphs are too small to put into a report and still have them be readable.
https://github.com/jackcmeyer/SmartSync/tree/master/documentation/architecture_recovery

Conclusion

In conclusion, SmartSync quickly became more than just another school project, and in the end it provided us with an incredibly rewarding learning experience. Aside from the experience we gained from learning Angular 2 and Spring Boot, SmartSync taught us how to focus on our architecture and taught us the tools and methods needed to reason about it. By keeping the architecture at the front of our concerns, we may not have reached all of our functionality goals, but we did develop an architecture that we are proud of. Coming into this project, none of us had focussed on architecture much, besides looking into different trends. By the end we all have developed knowledge and opinions of our own about different software architectures that will allow us to reason about it in a professional manner.