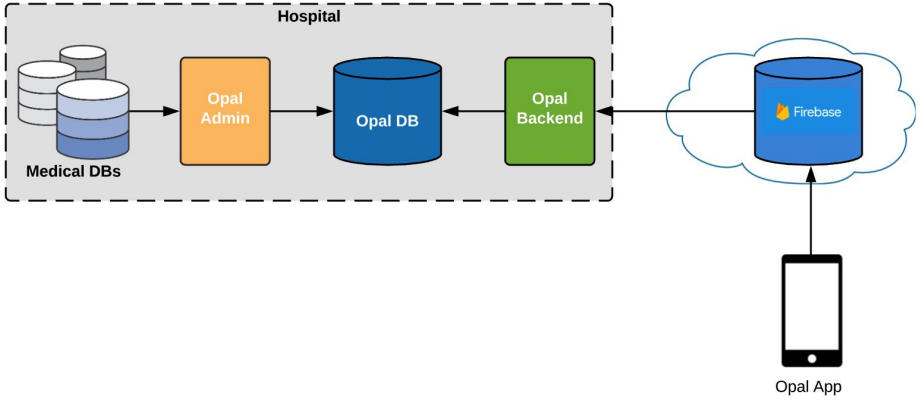
Opal Backend

David Herrera

Table of contents

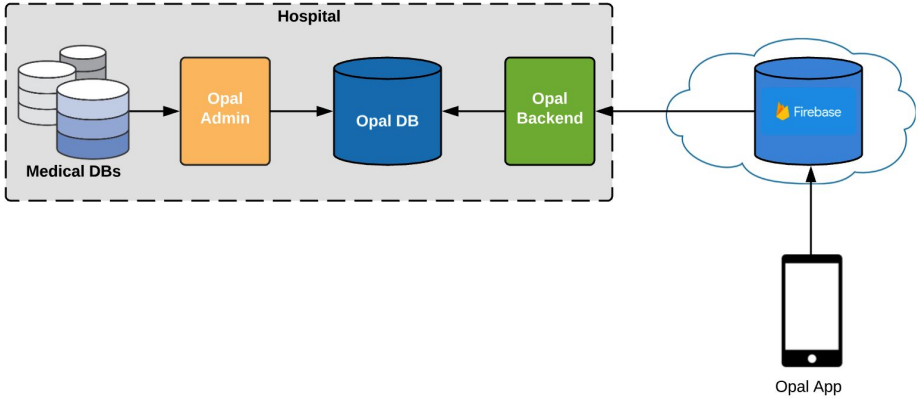
What is the Opal Back-end?

- Back-end for Opal mobile app
- Sits inside the hospital servers
- Three versions:
  - Prod
  - Pre-Prod
  - Sandbox
- Ran via a daemon manager called pm2
- Connects directly with OpalDB
- Connects with other services such as ORMS for patient check-in



Types of request

- Two types of Opal requests:
  - Unauthenticated request
  - Authenticated request
- **Security module** handles unauthenticated requests, including:
  - Password reset
  - Security Question
- **Main Module** handles authenticated requests
  - Any user request once authenticated



Authenticated request flow

Send request to Opal backend

- Call **sendRequestWithResponse()**
- Wait for response.
- Handle failure in terms of the view.
- Who can tell me what's wrong with the promise in this code?

```
/**
 * @ngdoc method
 * @name downloadDocumentFromServer
 * @methodOf MUHCApp.service:Documents
 * @param {String} serNum DocumentSerNum
 * @returns {Promise} Promise successful upon correct arrival of document,
 *                    rejected if error in server, or request timeout
 * @description Downloads the document that matches that DocumentSerNum
 *              parameter from the server
 **/
downloadDocumentFromServer:function(serNum)
{
    var _this = this;
    var r = $q.defer();
    RequestToServer.sendRequestWithResponse('DocumentContent',[serNum])
        .then(function(response)
    {
        if(response.Code === '3' && response.Data !== 'DocumentNotFound')
        {
            var doc = _this.getDocumentBySerNum(serNum);
            doc.Content = response.Data.Content;
            r.resolve("success");
        }else{
            r.reject(response);
        }

    }).catch(function(error)
    {
        r.reject(error);
    });
    return r.promise;
},
```

RequestToServer

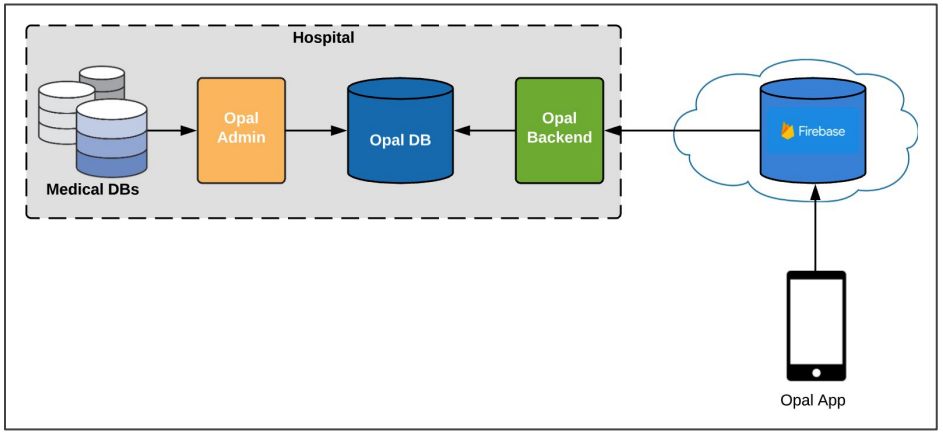
- The app sends a request using **RequestToServer** service.

```
function sendRequestWithReponse(requestType, requestParameters,
                                reference) {
    return new Promise((resolve, reject) => {
        //Sends request and gets random key for request
        sendRequest(typeOfRequest, parameters, encryptionKey,
                    referenceField)
            .then(key => {
                //Waits to obtain the request data.
                refRequestResponse.once('value', snapshot => {
                    if (snapshot.exists()) {
                        let data = snapshot.val();
                        data = ResponseValidator.validate(data, encryptionKey,
                            timeOut);
                        if (data.success) {
                            resolve(data.success)
                        } else {
                            reject(data.error)
                        }
                    }
                }, error => {
                    reject(error);
                });
            });
    });
}
```

```
function sendRequest(requestType, requestParameters)
{
    let request_object = {
        'Request' : requestType,
        'DeviceId': UUID.getUUID(),
        'Token':UserAuthorizationInfo.getToken(),
        'UserID': UserAuthorizationInfo.getUsername(),
        'Parameters':requestParameters,
        'Timestamp':firebase.database.ServerValue.TIMESTAMP,
        'UserEmail': UserAuthorizationInfo.getEmail(),
        'AppVersion': version
    };
    let reference = referenceField || 'requests';
    let pushID =  firebase_url.child(reference).push(request_object);
    return pushID.key;
}
```

Request In Firebase

- Firebase verifies the request has the right parameters based in the Firebase rules.
- The app listens for response for the given random key.
- The backend listens to requests.

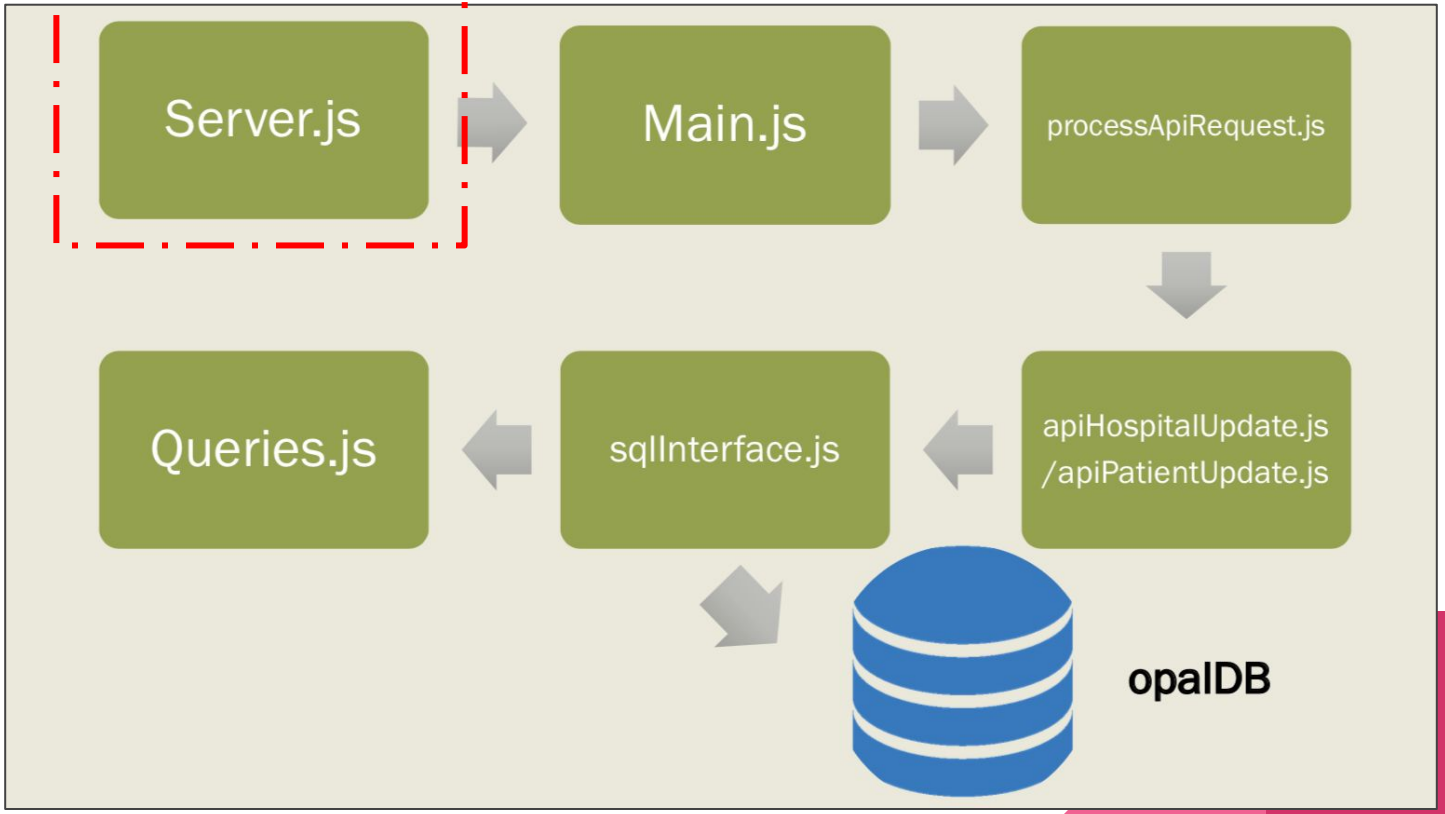



```
requests
  -LCZAwElcSC2qcRYZczd
    AppVersion: "0.9.108"
    DeviceId: "B633C67E-4957-45AD-9714-487DA66D4560"
    Parameters
      0: "Mi/X6oX0V9sOnqInwV1ng2UnBahv5Pr2q89GR5HK5eNRMiZ..."
    Request: "LRtatmP6NvmLzt5DzWJf0QdSjrFYX5m3v39w/eb0VY66+XC..."
    Timestamp: 1526394242210
    Token: "eyJhbGciOiJSUzI1NiIsImtpZCI6IjhiZjA2YWU3MGJhMjV..."
    UserEmail: "muhc.app.mobile@gmail.com"
    UserID: "ac6eaeaa-f725-4b07-bdc0-72faef725985"
```

Firebase Rules

```
"requests": {
  "$request_id": {
    ".write": "auth.uid !== null",
    ".read": "auth.uid !== null",
    ".validate": "newData.hasChildren(['Request', 'DeviceId', 'Token', 'UserID'])
    "&& newData.child('UserID').val() === auth.uid"
  }
},
```

Request In the Backend



Server.js - Responsibilities

- Spawning the server
- Instantiating listeners
- Determining if its an authenticated, unauthenticated request
- Sending response back to Firebase
- Logs incoming/outgoing request, Firebase errors

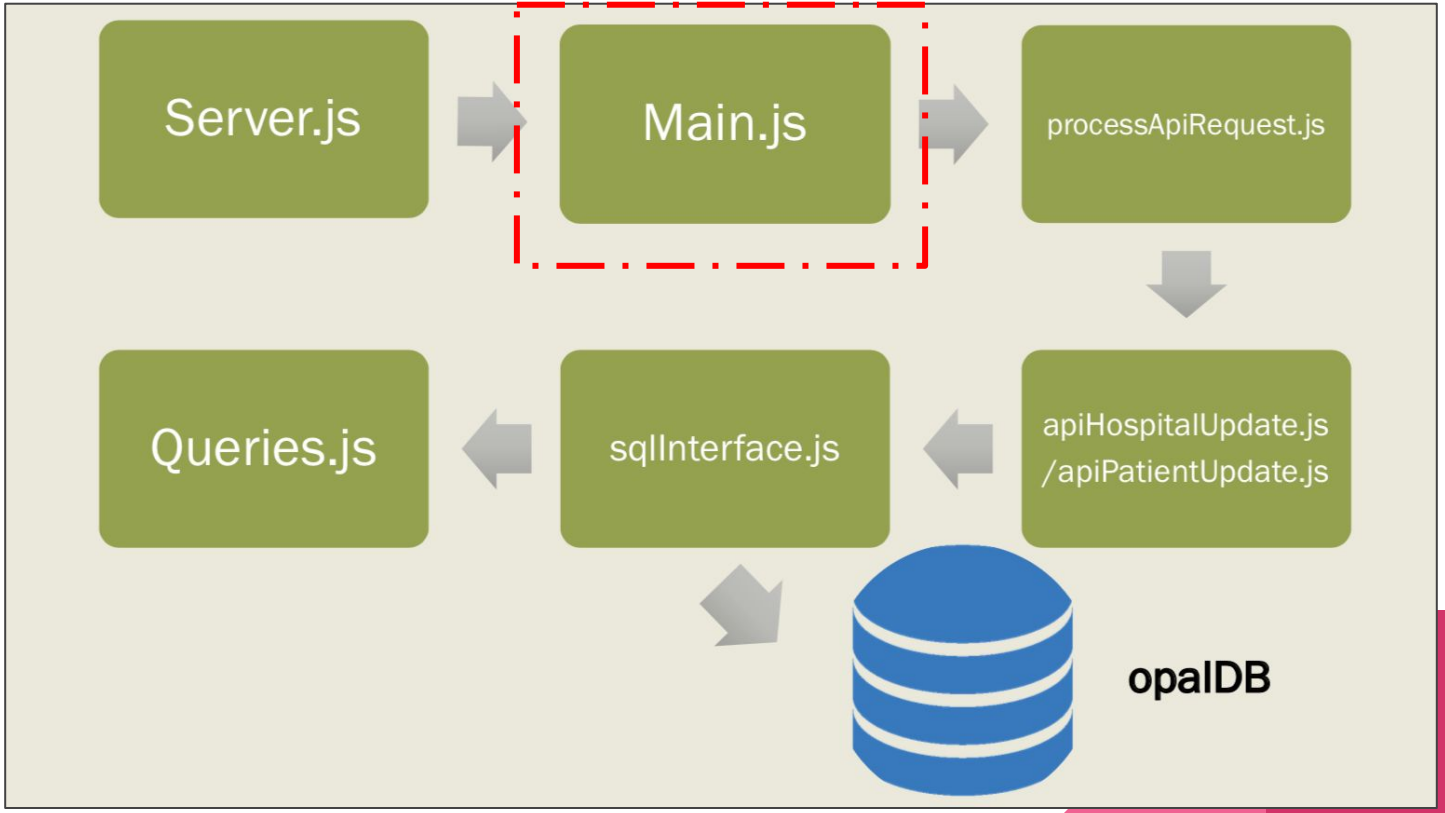
```
/**
 * listenForRequest
 * @param requestType
 * @desc Listen for firebase changes and send responses for requests
 */
function listenForRequest(requestType){
    logger.log('info','Starting '+ requestType+' listener.');

    //disconnect any existing listeners..
    ref.child(requestType).off();

    ref.child(requestType).on('child_added',
        function(snapshot){
            logger.log('debug', 'Received request from Firebase: ', JSON.stringify(snapshot.val()));
            logger.log('info', 'Received request from Firebase: ', snapshot.val().Request);

            if(snapshot.val().Request === 'HeartBeat'){
                logger.log('debug', 'Handling heartbeat request');
                handleHeartBeat(snapshot.val())
            } else {
                handleRequest(requestType,snapshot);
            }
        },
        function(error){
            logError(error);
        });
}
```

Request In the Backend - Main.js



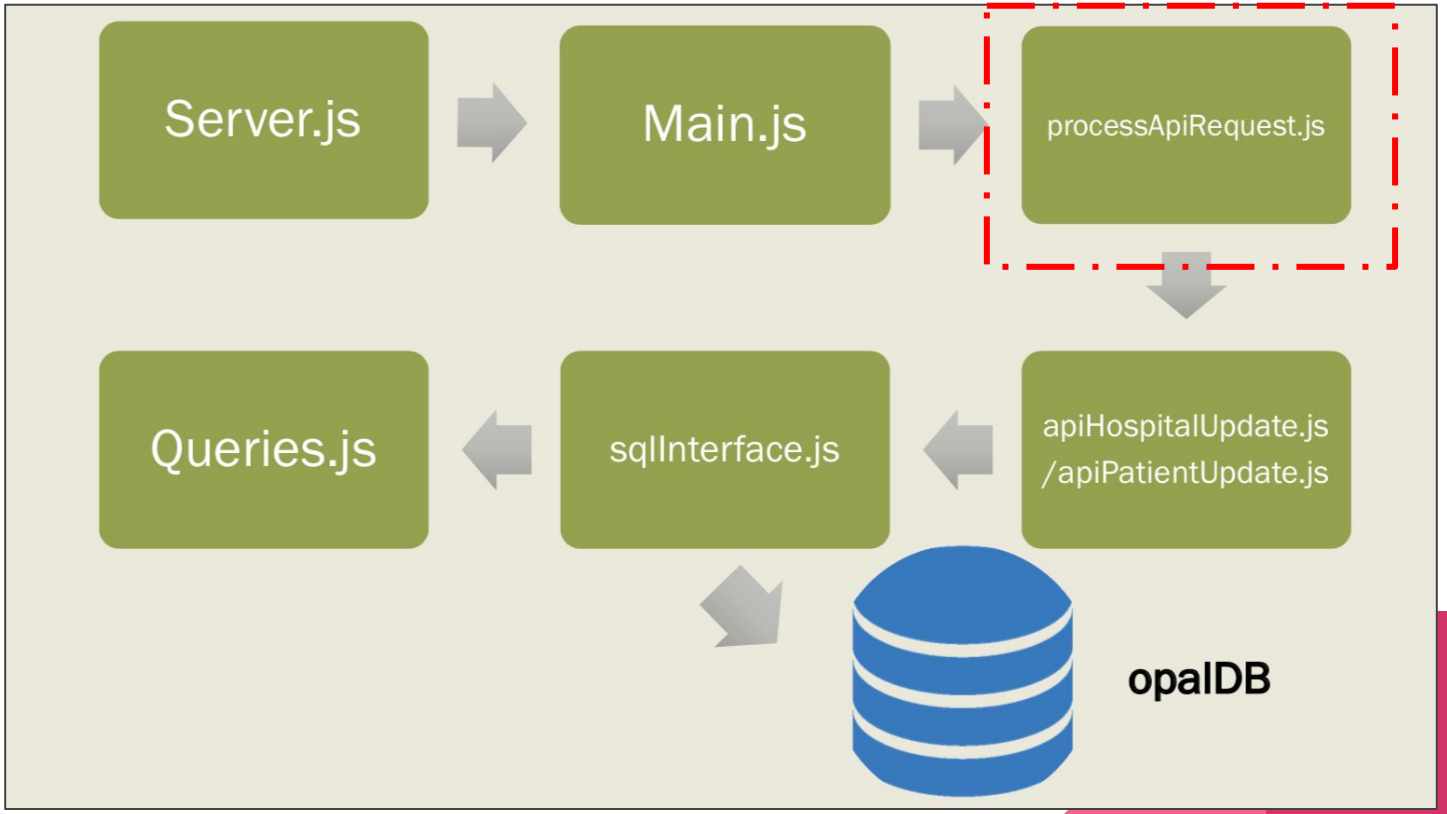
Main.js - Responsibilities

- Validates request credentials.
  - Makes sure the user is who he claims to be
  - Makes sure all the authentication parameters are defined
- Encrypts/Decrypts the request
- Calls the API handler
- Upon error, returns error response.
- Call to ***toLegacy()*** is an adapter design pattern for the requests. Work still needs to be done here.

```
/**
 * requestFormatter
 * @description handles the api requests by formatting
 *              the response obtained from the API
 * @param {key, request}
 * @returns {Promise}
 */
function requestFormatter({key,request}) {
    return RequestValidator.validate(key, request)
        .then( opalReq => { //opalReq of type, OpalRequest
            return processApiRequest.processRequest(opalReq.toLegacy())
                    .then((data)=>
            {
                logger.log('debug', 'Successfully processed request: ' + data);
                logger.log('info', 'Successfully processed request');

                let response = new OpalResponseSuccess(data, opalReq);
                return response.toLegacy();
            }).catch((err)=>{
                logger.log('error', 'Error processing request', err);
                let response = new OpalResponseError( 2, opalReq, err);
                return response.toLegacy();
            });
        }).catch( err => {
            logger.log('error', 'Error validating request', err);
            return err.toLegacy();
        });
}
```

Request In the Backend - processApiRequest



processApiRequest.js - Responsibilities

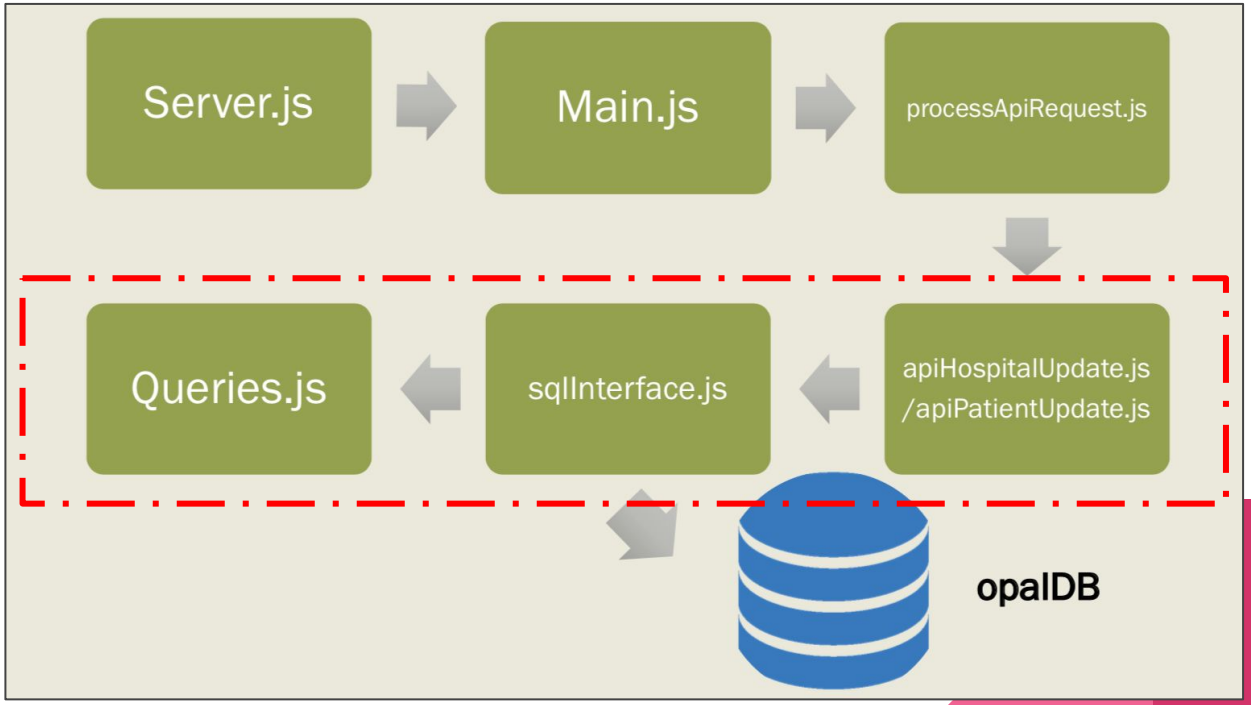
- Sends the request to the appropriate handler.
- Offers the API interface for the entire application.

```javascr
/**
 * processRequest
 * @desc Maps the incoming requestObject to the correct
 *       API function to handle it
 * @param requestObject
 * @return {Promise}
 */
exports.processRequest=function(requestObject) {
    const r = Q.defer();
    const type = requestObject.Request;
    if (API.hasOwnProperty(type)) {
        logger.log('debug', 'Processing request of type: ' + type);
        return  API[type](requestObject);
    }else{
        logger.log('error', 'Invalid request type: ' + type);
        r.reject('Invalid request type');
    }
    return r.promise;
};
```

```javascr
const API = {
    'DeviceIdentifier': apiHospitalUpdate.updateDeviceIdentifier,
    'Log': apiPatientUpdate.logActivity,
    'Login': apiPatientUpdate.login,
    'Logout': apiHospitalUpdate.logout,
    'Resume': apiPatientUpdate.resume,
    'Refresh': apiPatientUpdate.refresh,
    'AccountChange': apiHospitalUpdate.accountChange,
    'CheckCheckin': apiPatientUpdate.checkCheckin,
    'Checkin': apiHospitalUpdate.checkIn,
    'CheckinUpdate': apiPatientUpdate.checkinUpdate,
    'DocumentContent': apiPatientUpdate.getDocumentsContent,
    'Feedback': apiHospitalUpdate.inputFeedback,
    'LabResults': apiPatientUpdate.getLabResults,
    'MapLocation': apiPatientUpdate.getMapLocation,
    'Message': apiHospitalUpdate.sendMessage,
    'NotificationsAll': apiHospitalUpdate.getAllNotifications,
    'Questionnaires': apiPatientUpdate.getQuestionnaires,
    'QuestionnaireRating': apiHospitalUpdate.inputEducationalMaterialRating,
    'QuestionnaireAnswers': apiHospitalUpdate.inputQuestionnaireAnswers,
    'Read': apiHospitalUpdate.updateReadStatus,
    'PFPMembers': apiPatientUpdate.getPatientsForPatientsMembers
};
```

```javascr
/**
 * API HANDLERS FOR SECURITY SPECIFIC REQUESTS
 * @type {{PasswordReset: *, SecurityQuestion: *,
 *     SetNewPassword: *, VerifyAnswer: *}}
 */
exports.securityAPI = {
    'PasswordReset': security.resetPasswordRequest,
    'SecurityQuestion': security.securityQuestion,
    'SetNewPassword': security.resetPasswordRequest,
    'VerifyAnswer': security.resetPasswordRequest
};
```

Request In the Backend - apiPatientUpdate.js/ apiHospitalUpdate.js



apiPatientUpdate.js / apiHospitalUpdate.js - Responsabilities

- Call handler, return response
  - Call the respective handler
  - Handler validates parameters
  - Handles requests by calling the respective query.
  - Prepares the query results for the application
  - Returns response or error back up the chain.

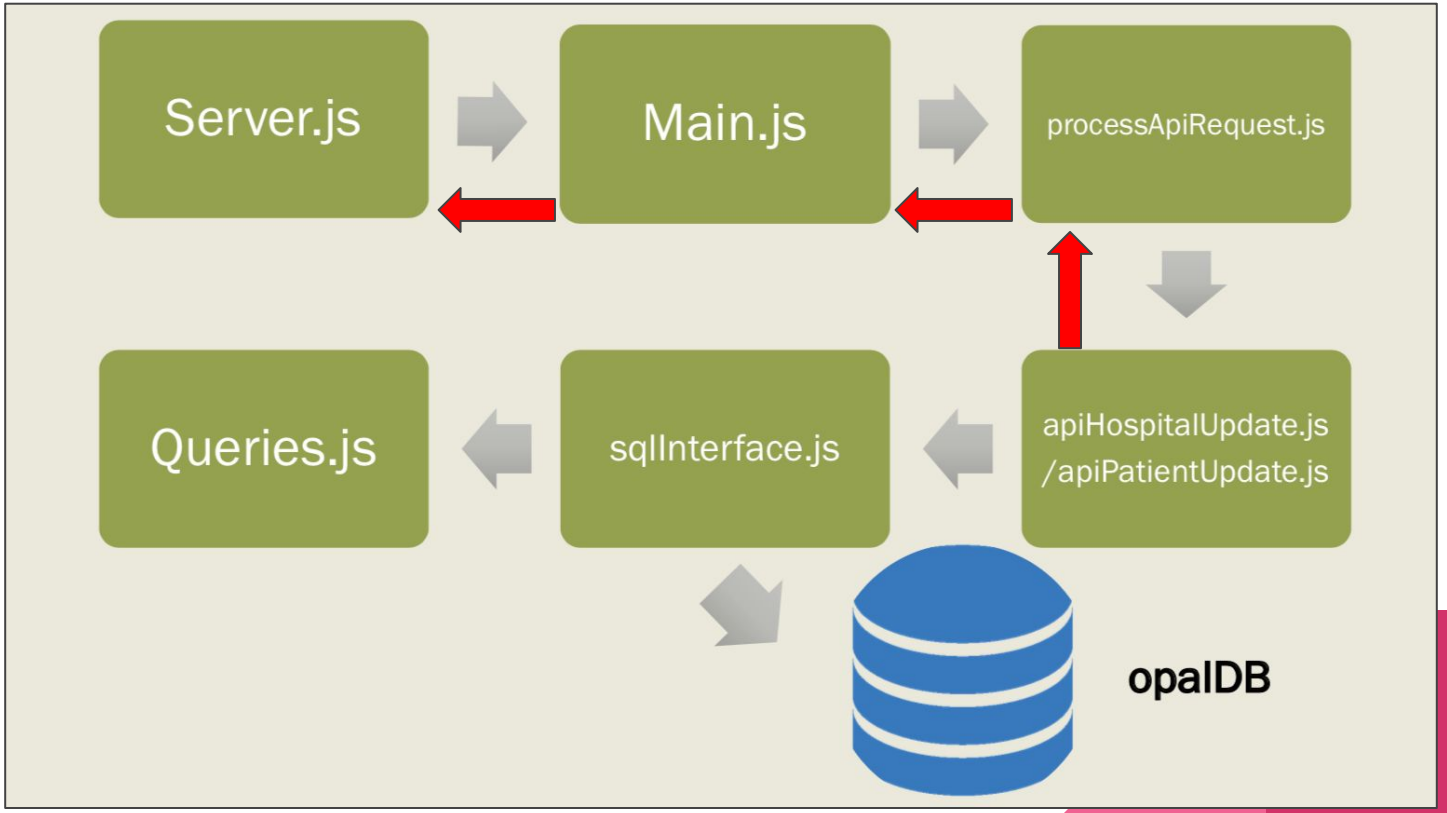
Example - Document handler

- **Validates** makes sure the parameter is an array
- **Queries,** does sql query where we grab the doc info.
- **Pre-processing,** in this case none.
- **Post-processing,** based on those paths it grabs all the documents
- **Validates,** Query return
- **Returns,** success or failure

```
exports.getDocumentsContent = function(requestObject) {

    let r = Q.defer();
    let documents = requestObject.Parameters;
    let userID = requestObject.UserID;
    if(!(typeof documents.constructor !=='undefined'
            &&documents.constructor=== Array)){
        r.reject({Response:'error',Reason:'Not an array'});
    }else{
        exports.runSqlQuery(queries.getDocumentsContentQuery(),
            [[documents],userID]).then((rows)=>{
            if(rows.length === 0) {
                r.resolve({Response:'success',Data:'DocumentNotFound'});
            } else {
                LoadDocuments(rows).then(function(documents) {
                    if(documents.length === 1)
                        r.resolve({Response:'success',Data:documents[0]});
                    else r.resolve({Response:'success',Data:documents});
                }).catch(function (err) {
                    r.reject({Response:'error', Reason:err});
                });
            }
        }).catch((err)=>{
            r.reject({Response:'error',Reason:err});
        });
    }
    return r.promise;
};
```

Request In the Backend - **Return promise chain!**



Return promise chain! - Example - Document Handler

- **Document handler** processes requests, returns to **apiPatientUpdate**
- From **apiPatientUpdate** return to **processApiRequest**
- From **processApiRequest** return to **main.js**
- From **main.js,** encrypt request, wrap it around an opal response object.
  - If Error, log errors of programming type and such, replace by a generic error message
- Send to **server.js**
- **Server.js** calls **uploadToFirebase** logs response.

Response in Firebase


```
-LCZi-9WexgOMBcV1c34
    Code: 3
    Data
        Content: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5Fnp6S93yWjLhk68f..."
        DocumentSerNum: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5FrHWUEi43GDaZcxy..."
        DocumentType: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5Fufc00BFF70LR8Y+..."
        FinalFileName: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5FqoomssrU8qp2lpb..."
    Headers
    Response: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5FBG3YQGkHC6eT093..."
    Timestamp: 1526403171552
```

Response in Firebase

- **Code**, success or failure codes. Ressembles the HTTP codes. I.e. 404, 200, 403.
- **Headers:** RequestKey
- **Response**: sucess/failure headers
- **Timestamp:** Time of request
- **Data:** Encrypted response data.


```
-LCZi-9WexgOMBcV1c34
    Code: 3
    Data
        Content: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5Fnp6S93yWjLhk68f..."
        DocumentSerNum: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5FrHWUEi43GDaZcxy..."
        DocumentType: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5Fufc00BFF70LR8Y+..."
        FinalFileName: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5FqoomssrU8qp2lpb..."
    Headers
    Response: "/IbLtKPFZFy3m0ioGYlwQv4ZXjDhVI5FBG3YQGkHC6eT093..."
    Timestamp: 1526403171552
```

Future work

Problem - Request Handler

- Request handling seems complicated
- How do we ensure this returns the right things statically?
- How do we know the request is being validated?
- Every requests should follow this pipeline
  - Validation
  - Query preparation
  - Query handling
  - Response preparation

```
exports.getDocumentsContent = function(requestObject) {

    let r = Q.defer();
    let documents = requestObject.Parameters;
    let userID = requestObject.UserID;
    if(!(typeof documents.constructor !=='undefined'
            &&documents.constructor=== Array)){
        r.reject({Response:'error',Reason:'Not an array'});
    }else{
        exports.runSqlQuery(queries.getDocumentsContentQuery(),
            [[documents],userID]).then((rows)=>{
            if(rows.length === 0) {
                r.resolve({Response:'success',Data:'DocumentNotFound'});
            } else {
                LoadDocuments(rows).then(function(documents) {
                    if(documents.length === 1)
                        r.resolve({Response:'success',Data:documents[0]});
                    else r.resolve({Response:'success',Data:documents});
                }).catch(function (err) {
                    r.reject({Response:'error', Reason:err});
                });
            }
        }).catch((err)=>{
            r.reject({Response:'error',Reason:err});
        });
    }
    return r.promise;
};
```

Solution - RequestHandler

Why don’t we enforce this pipeline with a programming paradigm?

Why?

- Avoids programmer mistakes
- Makes sure that every request is validated and makes sure our objects have the right types, everything done statically!

Solution - RequestHandler

- **Factory** should grab a request, instantiate the right handler and return the results of the handler and request.
- A **processor**, should take that handler and run it through the pipeline
- For every request we implement the **OpalRequestHandler**
- An **OpalHandler** should:
  1. Validate requests parameters
  2. Prepare parameters for query
  3. Call a query handler
  4. Validate query response
  5. Resolve/reject with error data or success data up the chain

Solution - RequestHandler

What do we need?

- A **Factory** which will just produce **OpalData** objects by instantiating the right **OpalRequestHandler.**
- A **processor**, this will take the **OpalRequestHandler,** and run it through the pipeline.
- Template class **OpalRequestHandler** will declare abstract functions that need to be implemented by every handler.
- Request handlers will have to extend and implement **OpalRequestHandler**
- **OpalDataSuccess, OpalDataError** classes which will serve as a base class for all the response objects and will be returned by the chain

Solution - Factory

```
let apiHandler = {
    "DocumentContent":Documents.DocumentContentHandler
};
function opalHandlerFactory(requestObject)
{
    if(apiHandler.hasOwnProperty(requestObject.RequestType))
    {
        let handlerClass = apiHandler[requestObject.RequestType];
        let opalRequestHandler = new handlerClass(requestObject);
        return apiHospital(requestObject, opalRequestHandler);
    }else{
        return Promise.reject(
            new OpalDataError("Invalid request type", requestObject));
    }
}
```

Solution - processor

- Gets the **opalRequestHandler** and runs it through pipeline
- Calls the different functions implemented by each handler.
- Pipeline:
  - Validate
  - Prepare queries
  - Queries
  - Post processing response

```
function apiHospital(requestObject, opalRequestHandler)
{
    let {userId, parameters} = requestObject;
    try{
        parameters = opalRequestHandler.validator(parameters);
    }catch(err)
    {
        return Promise.reject(err);
    }
    let queryParameters = opalRequestHandler.prepareParameters(parameters);

    return opalRequestHandler.handleRequestQueries()
        .then((queryResponse)=>{
            return opalRequestHandler.prepareOpalResponse(queryResponse);
        });
}
```

Future work - OpalRequestHandler

- Follows a **template design pattern**
- Implements all the functions to handler request

```
class OpalRequestHandler {

    constructor(){}
    validator(){}
    prepareParameters(){}
    prepareOpalResponse(){}
    handleRequestQueries(){}

}
class DocumentContentHandler extends OpalRequestHandler {
    constructor(objectRequest){}
    validator(){}
    prepareParameters(){}
    prepareOpalResponse(){}
    handleRequestQueries(){}
}
```

The end