**DATA SMART数据智能分析系统1.0**

**源程序**

**目 录**

[第1章 后台主程序 2](#_Toc502564825)

[第2章 前台主程序 72](#_Toc502564826)

# 后台主程序

|  |
| --- |
| */\*  \* Copyright (c) 2017. 联思智云（北京）科技有限公司. All rights reserved.  \*/* **package** com.smartsuites.socket;  **import** java.io.IOException; **import** java.net.URISyntaxException; **import** java.net.UnknownHostException; **import** java.text.ParseException; **import** java.text.SimpleDateFormat; **import** java.util.\*; **import** java.util.concurrent.ConcurrentHashMap; **import** java.util.concurrent.ConcurrentLinkedQueue; **import** java.util.regex.Matcher; **import** java.util.regex.Pattern;  **import** javax.servlet.http.HttpServletRequest;  **import** com.google.common.base.Strings; **import** com.google.common.collect.Sets; **import** com.smartsuites.utils.SecurityUtils; **import** org.apache.commons.lang.StringUtils; **import** org.apache.commons.vfs2.FileSystemException; **import** com.smartsuites.conf.ZeppelinConfiguration; **import** com.smartsuites.conf.ZeppelinConfiguration.ConfVars; **import** com.smartsuites.display.AngularObject; **import** com.smartsuites.display.AngularObjectRegistry; **import** com.smartsuites.display.AngularObjectRegistryListener; **import** com.smartsuites.display.Input; **import** com.smartsuites.helium.ApplicationEventListener; **import** com.smartsuites.helium.HeliumPackage; **import** com.smartsuites.interpreter.\*; **import** com.smartsuites.interpreter.remote.RemoteAngularObjectRegistry; **import** com.smartsuites.interpreter.remote.RemoteInterpreterProcessListener; **import** com.smartsuites.interpreter.thrift.InterpreterCompletion; **import** com.smartsuites.notebook.JobListenerFactory; **import** com.smartsuites.notebook.Folder; **import** com.smartsuites.notebook.Note; **import** com.smartsuites.notebook.Notebook; **import** com.smartsuites.notebook.NotebookAuthorization; **import** com.smartsuites.notebook.NotebookEventListener; **import** com.smartsuites.notebook.NotebookImportDeserializer; **import** com.smartsuites.notebook.Paragraph; **import** com.smartsuites.notebook.ParagraphJobListener; **import** com.smartsuites.notebook.repo.NotebookRepo.Revision; **import** com.smartsuites.notebook.socket.Message; **import** com.smartsuites.notebook.socket.Message.OP; **import** com.smartsuites.notebook.socket.WatcherMessage; **import** com.smartsuites.rest.exception.ForbiddenException; **import** com.smartsuites.scheduler.Job; **import** com.smartsuites.scheduler.Job.Status; **import** com.smartsuites.server.ZeppelinServer; **import** com.smartsuites.ticket.TicketContainer; **import** com.smartsuites.types.InterpreterSettingsList; **import** com.smartsuites.user.AuthenticationInfo; **import** com.smartsuites.util.WatcherSecurityKey; **import** com.smartsuites.utils.InterpreterBindingUtils; **import** org.eclipse.jetty.websocket.servlet.WebSocketServlet; **import** org.eclipse.jetty.websocket.servlet.WebSocketServletFactory; **import** org.joda.time.DateTime; **import** org.joda.time.format.DateTimeFormat; **import** org.joda.time.format.DateTimeFormatter; **import** org.quartz.SchedulerException; **import** org.slf4j.Logger; **import** org.slf4j.LoggerFactory;  **import** com.google.common.collect.Queues; **import** com.google.gson.Gson; **import** com.google.gson.GsonBuilder; **import** com.google.gson.reflect.TypeToken;  */\*\*  \* Data Smart websocket service.  \* 主业务逻辑  \*/* **public class** NotebookServer **extends** WebSocketServlet  **implements** NotebookSocketListener, JobListenerFactory, AngularObjectRegistryListener,  RemoteInterpreterProcessListener, ApplicationEventListener {   */\*\*  \* 任务管理器服务类型  \*/* **protected enum** JOB\_MANAGER\_SERVICE {  ***JOB\_MANAGER\_PAGE***(**"JOB\_MANAGER\_PAGE"**);  **private** String **serviceTypeKey**;   JOB\_MANAGER\_SERVICE(String serviceType) {  **this**.**serviceTypeKey** = serviceType;  }   String getKey() {  **return this**.**serviceTypeKey**;  }  }    **private static final** Logger ***LOG*** = LoggerFactory.*getLogger*(NotebookServer.**class**);  **private static** Gson *gson* = **new** GsonBuilder()  .setDateFormat(**"yyyy-MM-dd'T'HH:mm:ssZ"**)  .registerTypeAdapter(Date.**class**, **new** NotebookImportDeserializer())  .setPrettyPrinting()  .registerTypeAdapterFactory(Input.***TypeAdapterFactory***).create();   *// 主要保存访问note的客户端* **final** Map<String, List<NotebookSocket>> **noteSocketMap** = **new** HashMap<>();   *// 主要保存所有访问的客户端* **final** Queue<NotebookSocket> **connectedSockets** = **new** ConcurrentLinkedQueue<>();   *// 主要保存当前用户所有的客户端连接* **final** Map<String, Queue<NotebookSocket>> **userConnectedSockets** = **new** ConcurrentHashMap<>(); **final** Queue<NotebookSocket> **watcherSockets** = Queues.*newConcurrentLinkedQueue*();   **private** Notebook notebook() {  **return** DataSmartServer.*notebook*;  }   @Override  **public void** configure(WebSocketServletFactory factory) {  factory.setCreator(**new** NotebookWebSocketCreator(**this**));  }   **public boolean** checkOrigin(HttpServletRequest request, String origin) {  **try** {  **return** SecurityUtils.*isValidOrigin*(origin, DataSmartConfiguration.*create*());  } **catch** (UnknownHostException e) {  ***LOG***.error(e.toString(), e);  } **catch** (URISyntaxException e) {  ***LOG***.error(e.toString(), e);  }  **return false**;  }   **public** NotebookSocket doWebSocketConnect(HttpServletRequest req, String protocol) {  **return new** NotebookSocket(req, protocol, **this**);  }   @Override  **public void** onOpen(NotebookSocket conn) {  ***LOG***.info(**"New connection from {} : {}"**, conn.getRequest().getRemoteAddr(),  conn.getRequest().getRemotePort());  **connectedSockets**.add(conn);  }   @Override  **public void** onMessage(NotebookSocket conn, String msg) {  Notebook notebook = notebook();  **try** {  Message messagereceived = deserializeMessage(msg);  ***LOG***.debug(**"RECEIVE << "** + messagereceived.**op** +  **", RECEIVE PRINCIPAL << "** + messagereceived.**principal** +  **", RECEIVE TICKET << "** + messagereceived.**ticket** +  **", RECEIVE ROLES << "** + messagereceived.**roles** +  **", RECEIVE DATA << "** + messagereceived.**data**);   **if** (***LOG***.isTraceEnabled()) {  ***LOG***.trace(**"RECEIVE MSG = "** + messagereceived);  }   String ticket = TicketContainer.***instance***.getTicket(messagereceived.**principal**);  **if** (ticket != **null** &&  (messagereceived.**ticket** == **null** || !ticket.equals(messagereceived.**ticket**))) {  */\* not to pollute logs, log instead of exception \*/* **if** (StringUtils.*isEmpty*(messagereceived.**ticket**)) {  ***LOG***.debug(**"{} message: invalid ticket {} != {}"**, messagereceived.**op**,  messagereceived.**ticket**, ticket);  } **else** {  **if** (!messagereceived.**op**.equals(OP.***PING***)) {  conn.send(serializeMessage(**new** Message(OP.***SESSION\_LOGOUT***).put(**"info"**,  **"Your ticket is invalid possibly due to server restart. "** + **"Please login again."**)));  }  }  **return**;  }   DataSmartConfiguration conf = DataSmartConfiguration.*create*();  **boolean** allowAnonymous = conf.isAnonymousAllowed();  **if** (!allowAnonymous && messagereceived.**principal**.equals(**"anonymous"**)) {  **throw new** Exception(**"Anonymous access not allowed "**);  }   HashSet<String> userAndRoles = **new** HashSet<>();  userAndRoles.add(messagereceived.**principal**);  **if** (!messagereceived.**roles**.equals(**""**)) {  HashSet<String> roles =  *gson*.fromJson(messagereceived.**roles**, **new** TypeToken<HashSet<String>>() {  }.getType());  **if** (roles != **null**) {  userAndRoles.addAll(roles);  }  }  **if** (StringUtils.*isEmpty*(conn.getUser())) {  addUserConnection(messagereceived.**principal**, conn);  }  AuthenticationInfo subject =  **new** AuthenticationInfo(messagereceived.**principal**, messagereceived.**roles**,  messagereceived.**ticket**);   */\*\* Lets be elegant here \*/* **switch** (messagereceived.**op**) {  *// 列出所有Notes* **case *LIST\_NOTES***:  unicastNoteList(conn, subject, userAndRoles);  **break**;   *// 重新从Repo加载Notes* **case *RELOAD\_NOTES\_FROM\_REPO***:  broadcastReloadedNoteList(subject, userAndRoles);  **break**;   *// 获取首页的Note* **case *GET\_HOME\_NOTE***:  sendHomeNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 获取一个Note* **case *GET\_NOTE***:  sendNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 创建一个Note* **case *NEW\_NOTE***:  createNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 删除一个Note* **case *DEL\_NOTE***:  removeNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 删除一个文件夹* **case *REMOVE\_FOLDER***:  removeFolder(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 将Note放到回收站* **case *MOVE\_NOTE\_TO\_TRASH***:  moveNoteToTrash(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 将文件夹放到回收站* **case *MOVE\_FOLDER\_TO\_TRASH***:  moveFolderToTrash(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 清空回收站* **case *EMPTY\_TRASH***:  emptyTrash(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 从回收站恢复文件夹* **case *RESTORE\_FOLDER***:  restoreFolder(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 从回收站恢复Note* **case *RESTORE\_NOTE***:  restoreNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 从回收站恢复所有文件* **case *RESTORE\_ALL***:  restoreAll(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 复制一个Note* **case *CLONE\_NOTE***:  cloneNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 导入一个Note* **case *IMPORT\_NOTE***:  importNote(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 提交一个片段* **case *COMMIT\_PARAGRAPH***:  updateParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;   *// 运行一个片段* **case *RUN\_PARAGRAPH***:  runParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;    **case *PARAGRAPH\_EXECUTED\_BY\_SPELL***:  broadcastSpellExecution(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *RUN\_ALL\_PARAGRAPHS***:  runAllParagraphs(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *CANCEL\_PARAGRAPH***:  cancelParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *MOVE\_PARAGRAPH***:  moveParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *INSERT\_PARAGRAPH***:  insertParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *COPY\_PARAGRAPH***:  copyParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *PARAGRAPH\_REMOVE***:  removeParagraph(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *PARAGRAPH\_CLEAR\_OUTPUT***:  clearParagraphOutput(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *PARAGRAPH\_CLEAR\_ALL\_OUTPUT***:  clearAllParagraphOutput(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *NOTE\_UPDATE***:  updateNote(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *NOTE\_RENAME***:  renameNote(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *FOLDER\_RENAME***:  renameFolder(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *UPDATE\_PERSONALIZED\_MODE***:  updatePersonalizedMode(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *COMPLETION***:  completion(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *PING***:  **break**; *//do nothing* **case *ANGULAR\_OBJECT\_UPDATED***:  angularObjectUpdated(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *ANGULAR\_OBJECT\_CLIENT\_BIND***:  angularObjectClientBind(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *ANGULAR\_OBJECT\_CLIENT\_UNBIND***:  angularObjectClientUnbind(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *LIST\_CONFIGURATIONS***:  sendAllConfigurations(conn, userAndRoles, notebook);  **break**;  **case *CHECKPOINT\_NOTE***:  checkpointNote(conn, notebook, messagereceived);  **break**;  **case *LIST\_REVISION\_HISTORY***:  listRevisionHistory(conn, notebook, messagereceived);  **break**;  **case *SET\_NOTE\_REVISION***:  setNoteRevision(conn, userAndRoles, notebook, messagereceived);  **break**;  **case *NOTE\_REVISION***:  getNoteByRevision(conn, notebook, messagereceived);  **break**;  **case *LIST\_NOTE\_JOBS***:  unicastNoteJobInfo(conn, messagereceived);  **break**;  **case *UNSUBSCRIBE\_UPDATE\_NOTE\_JOBS***:  unsubscribeNoteJobInfo(conn);  **break**;  **case *GET\_INTERPRETER\_BINDINGS***:  getInterpreterBindings(conn, messagereceived);  **break**;  **case *SAVE\_INTERPRETER\_BINDINGS***:  saveInterpreterBindings(conn, messagereceived);  **break**;  **case *EDITOR\_SETTING***:  getEditorSetting(conn, messagereceived);  **break**;  **case *GET\_INTERPRETER\_SETTINGS***:  getInterpreterSettings(conn, subject);  **break**;  **case *WATCHER***:  switchConnectionToWatcher(conn, messagereceived);  **break**;  **default**:  **break**;  }  } **catch** (Exception e) {  ***LOG***.error(**"Can't handle message: "** + msg, e);  }  }   @Override  **public void** onClose(NotebookSocket conn, **int** code, String reason) {  ***LOG***.info(**"Closed connection to {} : {}. ({}) {}"**, conn.getRequest().getRemoteAddr(),  conn.getRequest().getRemotePort(), code, reason);  removeConnectionFromAllNote(conn);  **connectedSockets**.remove(conn);  removeUserConnection(conn.getUser(), conn);  }   **private void** removeUserConnection(String user, NotebookSocket conn) {  **if** (**userConnectedSockets**.containsKey(user)) {  **userConnectedSockets**.get(user).remove(conn);  } **else** {  ***LOG***.warn(**"Closing connection that is absent in user connections"**);  }  }   **private void** addUserConnection(String user, NotebookSocket conn) {  conn.setUser(user);  **if** (**userConnectedSockets**.containsKey(user)) {  **userConnectedSockets**.get(user).add(conn);  } **else** {  Queue<NotebookSocket> socketQueue = **new** ConcurrentLinkedQueue<>();  socketQueue.add(conn);  **userConnectedSockets**.put(user, socketQueue);  }  }   *// 消息的转换* **protected** Message deserializeMessage(String msg) {  **return** *gson*.fromJson(msg, Message.**class**);  }   **protected** String serializeMessage(Message m) {  **return** *gson*.toJson(m);  }   **private void** addConnectionToNote(String noteId, NotebookSocket socket) {  **synchronized** (**noteSocketMap**) {  removeConnectionFromAllNote(socket); *// make sure a socket relates only a single note.* List<NotebookSocket> socketList = **noteSocketMap**.get(noteId);  **if** (socketList == **null**) {  socketList = **new** LinkedList<>();  **noteSocketMap**.put(noteId, socketList);  }  **if** (!socketList.contains(socket)) {  socketList.add(socket);  }  }  }   *// 防止noteSocketMap在一个Note中有相同的存在* **private void** removeConnectionFromNote(String noteId, NotebookSocket socket) {  **synchronized** (**noteSocketMap**) {  List<NotebookSocket> socketList = **noteSocketMap**.get(noteId);  **if** (socketList != **null**) {  socketList.remove(socket);  }  }  }   **private void** removeNote(String noteId) {  **synchronized** (**noteSocketMap**) {  List<NotebookSocket> socketList = **noteSocketMap**.remove(noteId);  }  }   **private void** removeConnectionFromAllNote(NotebookSocket socket) {  **synchronized** (**noteSocketMap**) {  Set<String> keys = **noteSocketMap**.keySet();  **for** (String noteId : keys) {  removeConnectionFromNote(noteId, socket);  }  }  }   **private** String getOpenNoteId(NotebookSocket socket) {  String id = **null**;  **synchronized** (**noteSocketMap**) {  Set<String> keys = **noteSocketMap**.keySet();  **for** (String noteId : keys) {  List<NotebookSocket> sockets = **noteSocketMap**.get(noteId);  **if** (sockets.contains(socket)) {  id = noteId;  }  }  }   **return** id;  }   **private void** broadcastToNoteBindedInterpreter(String interpreterGroupId, Message m) {  Notebook notebook = notebook();  List<Note> notes = notebook.getAllNotes();  **for** (Note note : notes) {  List<String> ids = notebook.getInterpreterSettingManager()  .getInterpreterBinding(note.getId());  **for** (String id : ids) {  **if** (id.equals(interpreterGroupId)) {  broadcast(note.getId(), m);  }  }  }  }   **private void** broadcast(String noteId, Message m) {  List<NotebookSocket> socketsToBroadcast = Collections.*emptyList*();  **synchronized** (**noteSocketMap**) {  broadcastToWatchers(noteId, StringUtils.***EMPTY***, m);  List<NotebookSocket> socketLists = **noteSocketMap**.get(noteId);  **if** (socketLists == **null** || socketLists.size() == 0) {  **return**;  }  socketsToBroadcast = **new** ArrayList<>(socketLists);  }  ***LOG***.debug(**"SEND >> "** + m);  **for** (NotebookSocket conn : socketsToBroadcast) {  **try** {  conn.send(serializeMessage(m));  } **catch** (IOException e) {  ***LOG***.error(**"socket error"**, e);  }  }  }   **private void** broadcastExcept(String noteId, Message m, NotebookSocket exclude) {  List<NotebookSocket> socketsToBroadcast = Collections.*emptyList*();  **synchronized** (**noteSocketMap**) {  broadcastToWatchers(noteId, StringUtils.***EMPTY***, m);  List<NotebookSocket> socketLists = **noteSocketMap**.get(noteId);  **if** (socketLists == **null** || socketLists.size() == 0) {  **return**;  }  socketsToBroadcast = **new** ArrayList<>(socketLists);  }   ***LOG***.debug(**"SEND >> "** + m);  **for** (NotebookSocket conn : socketsToBroadcast) {  **if** (exclude.equals(conn)) {  **continue**;  }  **try** {  conn.send(serializeMessage(m));  } **catch** (IOException e) {  ***LOG***.error(**"socket error"**, e);  }  }  }   **private void** multicastToUser(String user, Message m) {  **if** (!**userConnectedSockets**.containsKey(user)) {  ***LOG***.warn(**"Multicasting to user {} that is not in connections map"**, user);  **return**;  }   **for** (NotebookSocket conn : **userConnectedSockets**.get(user)) {  unicast(m, conn);  }  }   **private void** unicast(Message m, NotebookSocket conn) {  **try** {  conn.send(serializeMessage(m));  } **catch** (IOException e) {  ***LOG***.error(**"socket error"**, e);  }  broadcastToWatchers(StringUtils.***EMPTY***, StringUtils.***EMPTY***, m);  }   **public void** unicastNoteJobInfo(NotebookSocket conn, Message fromMessage) **throws** IOException {  addConnectionToNote(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(), conn);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  List<Map<String, Object>> noteJobs = notebook().getJobListByUnixTime(**false**, 0, subject);  Map<String, Object> response = **new** HashMap<>();   response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, noteJobs);   conn.send(serializeMessage(**new** Message(OP.***LIST\_NOTE\_JOBS***).put(**"noteJobs"**, response)));  }   **public void** broadcastUpdateNoteJobInfo(**long** lastUpdateUnixTime) **throws** IOException {  List<Map<String, Object>> noteJobs = **new** LinkedList<>();  Notebook notebookObject = notebook();  List<Map<String, Object>> jobNotes = **null**;  **if** (notebookObject != **null**) {  jobNotes = notebook().getJobListByUnixTime(**false**, lastUpdateUnixTime, **null**);  noteJobs = jobNotes == **null** ? noteJobs : jobNotes;  }   Map<String, Object> response = **new** HashMap<>();  response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, noteJobs != **null** ? noteJobs : **new** LinkedList<>());   broadcast(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(),  **new** Message(OP.***LIST\_UPDATE\_NOTE\_JOBS***).put(**"noteRunningJobs"**, response));  }   **public void** unsubscribeNoteJobInfo(NotebookSocket conn) {  removeConnectionFromNote(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(), conn);  }   **public void** saveInterpreterBindings(NotebookSocket conn, Message fromMessage) {  String noteId = (String) fromMessage.**data**.get(**"noteId"**);  **try** {  List<String> settingIdList =  *gson*.fromJson(String.*valueOf*(fromMessage.**data**.get(**"selectedSettingIds"**)),  **new** TypeToken<ArrayList<String>>() {  }.getType());  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  notebook().bindInterpretersToNote(subject.getUser(), noteId, settingIdList);  broadcastInterpreterBindings(noteId,  InterpreterBindingUtils.*getInterpreterBindings*(notebook(), noteId));  } **catch** (Exception e) {  ***LOG***.error(**"Error while saving interpreter bindings"**, e);  }  }   **public void** getInterpreterBindings(NotebookSocket conn, Message fromMessage) **throws** IOException {  String noteId = (String) fromMessage.**data**.get(**"noteId"**);  List<InterpreterSettingsList> settingList =  InterpreterBindingUtils.*getInterpreterBindings*(notebook(), noteId);  conn.send(serializeMessage(  **new** Message(OP.***INTERPRETER\_BINDINGS***).put(**"interpreterBindings"**, settingList)));  }   *// 返回Note的Id和Name。* **public** List<Map<String, String>> generateNotesInfo(**boolean** needsReload,  AuthenticationInfo subject, Set<String> userAndRoles) {   Notebook notebook = notebook();   DataSmartConfiguration conf = notebook.getConf();  String homescreenNoteId = conf.getString(ConfVars.***ZEPPELIN\_NOTEBOOK\_HOMESCREEN***);  **boolean** hideHomeScreenNotebookFromList =  conf.getBoolean(ConfVars.***ZEPPELIN\_NOTEBOOK\_HOMESCREEN\_HIDE***);   **if** (needsReload) {  **try** {  notebook.reloadAllNotes(subject);  } **catch** (IOException e) {  ***LOG***.error(**"Fail to reload notes from repository"**, e);  }  }   List<Note> notes = notebook.getAllNotes(userAndRoles);  List<Map<String, String>> notesInfo = **new** LinkedList<>();  **for** (Note note : notes) {  Map<String, String> info = **new** HashMap<>();   **if** (hideHomeScreenNotebookFromList && note.getId().equals(homescreenNoteId)) {  **continue**;  }   info.put(**"id"**, note.getId());  info.put(**"name"**, note.getName());  notesInfo.add(info);  }   **return** notesInfo;  }   **public void** broadcastNote(Note note) {  broadcast(note.getId(), **new** Message(OP.***NOTE***).put(**"note"**, note));  }   **public void** broadcastInterpreterBindings(String noteId, List settingList) {  broadcast(noteId, **new** Message(OP.***INTERPRETER\_BINDINGS***).put(**"interpreterBindings"**, settingList));  }   **public void** unicastParagraph(Note note, Paragraph p, String user) {  **if** (!note.isPersonalizedMode() || p == **null** || user == **null**) {  **return**;  }   **if** (!**userConnectedSockets**.containsKey(user)) {  ***LOG***.warn(**"Failed to send unicast. user {} that is not in connections map"**, user);  **return**;  }   **for** (NotebookSocket conn : **userConnectedSockets**.get(user)) {  Message m = **new** Message(OP.***PARAGRAPH***).put(**"paragraph"**, p);  unicast(m, conn);  }  }   **public void** broadcastParagraph(Note note, Paragraph p) {  **if** (note.isPersonalizedMode()) {  broadcastParagraphs(p.getUserParagraphMap(), p);  } **else** {  broadcast(note.getId(), **new** Message(OP.***PARAGRAPH***).put(**"paragraph"**, p));  }  }   **public void** broadcastParagraphs(Map<String, Paragraph> userParagraphMap,  Paragraph defaultParagraph) {  **if** (**null** != userParagraphMap) {  **for** (String user : userParagraphMap.keySet()) {  multicastToUser(user,  **new** Message(OP.***PARAGRAPH***).put(**"paragraph"**, userParagraphMap.get(user)));  }  }  }   **private void** broadcastNewParagraph(Note note, Paragraph para) {  ***LOG***.info(**"Broadcasting paragraph on run call instead of note."**);  **int** paraIndex = note.getParagraphs().indexOf(para);  broadcast(note.getId(),  **new** Message(OP.***PARAGRAPH\_ADDED***).put(**"paragraph"**, para).put(**"index"**, paraIndex));  }   **public void** broadcastNoteList(AuthenticationInfo subject, HashSet userAndRoles) {  **if** (subject == **null**) {  subject = **new** AuthenticationInfo(StringUtils.***EMPTY***);  }  *//send first to requesting user* List<Map<String, String>> notesInfo = generateNotesInfo(**false**, subject, userAndRoles);  multicastToUser(subject.getUser(), **new** Message(OP.***NOTES\_INFO***).put(**"notes"**, notesInfo));  *//to others afterwards* broadcastNoteListExcept(notesInfo, subject);  }   *// 获取Note列表* **public void** unicastNoteList(NotebookSocket conn, AuthenticationInfo subject,  HashSet<String> userAndRoles) {  List<Map<String, String>> notesInfo = generateNotesInfo(**false**, subject, userAndRoles);  unicast(**new** Message(OP.***NOTES\_INFO***).put(**"notes"**, notesInfo), conn);  }   **public void** broadcastReloadedNoteList(AuthenticationInfo subject, HashSet userAndRoles) {  **if** (subject == **null**) {  subject = **new** AuthenticationInfo(StringUtils.***EMPTY***);  }   *//reload and reply first to requesting user  // 重新从Repo加载Notes* List<Map<String, String>> notesInfo = generateNotesInfo(**true**, subject, userAndRoles);  multicastToUser(subject.getUser(), **new** Message(OP.***NOTES\_INFO***).put(**"notes"**, notesInfo));  *//to others afterwards* broadcastNoteListExcept(notesInfo, subject);  }   **private void** broadcastNoteListExcept(List<Map<String, String>> notesInfo,  AuthenticationInfo subject) {  Set<String> userAndRoles;  NotebookAuthorization authInfo = NotebookAuthorization.*getInstance*();  **for** (String user : **userConnectedSockets**.keySet()) {  **if** (subject.getUser().equals(user)) {  **continue**;  }  *//reloaded already above; parameter - false* userAndRoles = authInfo.getRoles(user);  userAndRoles.add(user);  notesInfo = generateNotesInfo(**false**, **new** AuthenticationInfo(user), userAndRoles);  multicastToUser(user, **new** Message(OP.***NOTES\_INFO***).put(**"notes"**, notesInfo));  }  }   **void** permissionError(NotebookSocket conn, String op, String userName, Set<String> userAndRoles,  Set<String> allowed) **throws** IOException {  ***LOG***.info(**"Cannot {}. Connection readers {}. Allowed readers {}"**, op, userAndRoles, allowed);   conn.send(serializeMessage(**new** Message(OP.***AUTH\_INFO***).put(**"info"**,  **"Insufficient privileges to "** + op + **" note.\n\n"** + **"Allowed users or roles: "** + allowed  .toString() + **"\n\n"** + **"But the user "** + userName + **" belongs to: "** + userAndRoles  .toString())));  }   */\*\*  \** ***@return*** *false if user doesn't have reader permission for this paragraph  \*/* **private boolean** hasParagraphReaderPermission(NotebookSocket conn,  Notebook notebook, String noteId,  HashSet<String> userAndRoles,  String principal, String op)  **throws** IOException {   NotebookAuthorization notebookAuthorization = notebook.getNotebookAuthorization();  **if** (!notebookAuthorization.isReader(noteId, userAndRoles)) {  permissionError(conn, op, principal, userAndRoles,  notebookAuthorization.getOwners(noteId));  **return false**;  }   **return true**;  }   */\*\*  \** ***@return*** *false if user doesn't have runner permission for this paragraph  \*/* **private boolean** hasParagraphRunnerPermission(NotebookSocket conn,  Notebook notebook, String noteId,  HashSet<String> userAndRoles,  String principal, String op)  **throws** IOException {   NotebookAuthorization notebookAuthorization = notebook.getNotebookAuthorization();  **if** (!notebookAuthorization.isRunner(noteId, userAndRoles)) {  permissionError(conn, op, principal, userAndRoles,  notebookAuthorization.getOwners(noteId));  **return false**;  }   **return true**;  }   */\*\*  \** ***@return*** *false if user doesn't have writer permission for this paragraph  \*/* **private boolean** hasParagraphWriterPermission(NotebookSocket conn,  Notebook notebook, String noteId,  HashSet<String> userAndRoles,  String principal, String op)  **throws** IOException {   NotebookAuthorization notebookAuthorization = notebook.getNotebookAuthorization();  **if** (!notebookAuthorization.isWriter(noteId, userAndRoles)) {  permissionError(conn, op, principal, userAndRoles,  notebookAuthorization.getOwners(noteId));  **return false**;  }   **return true**;  }   */\*\*  \** ***@return*** *false if user doesn't have owner permission for this paragraph  \*/* **private boolean** hasParagraphOwnerPermission(NotebookSocket conn,  Notebook notebook, String noteId,  HashSet<String> userAndRoles,  String principal, String op)  **throws** IOException {   NotebookAuthorization notebookAuthorization = notebook.getNotebookAuthorization();  **if** (!notebookAuthorization.isOwner(noteId, userAndRoles)) {  permissionError(conn, op, principal, userAndRoles,  notebookAuthorization.getOwners(noteId));  **return false**;  }   **return true**;  }   *// 获取一个Note* **private void** sendNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {   ***LOG***.info(**"New operation from {} : {} : {} : {} : {}"**, conn.getRequest().getRemoteAddr(),  conn.getRequest().getRemotePort(), fromMessage.**principal**, fromMessage.**op**,  fromMessage.get(**"id"**));   String noteId = (String) fromMessage.get(**"id"**);  **if** (noteId == **null**) {  **return**;  }   String user = fromMessage.**principal**;   Note note = notebook.getNote(noteId);  **if** (note != **null**) {   **if** (!hasParagraphReaderPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"read"**)) {  **return**;  }  *// 添加当前的正在应用当前noteid的socket* addConnectionToNote(note.getId(), conn);   **if** (note.isPersonalizedMode()) {  note = note.getUserNote(user);  }  conn.send(serializeMessage(**new** Message(OP.***NOTE***).put(**"note"**, note)));  sendAllAngularObjects(note, user, conn);  } **else** {  conn.send(serializeMessage(**new** Message(OP.***NOTE***).put(**"note"**, **null**)));  }  }   *// 返回Home的Note，* **private void** sendHomeNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  String noteId = notebook.getConf().getString(ConfVars.***ZEPPELIN\_NOTEBOOK\_HOMESCREEN***);  String user = fromMessage.**principal**;   Note note = **null**;  **if** (noteId != **null**) {  note = notebook.getNote(noteId);  }   **if** (note != **null**) {  **if** (!hasParagraphReaderPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"read"**)) {  **return**;  }   *// 将noteid，websocket加入this.noteMap* addConnectionToNote(note.getId(), conn);  conn.send(serializeMessage(**new** Message(OP.***NOTE***).put(**"note"**, note)));  *// 向前端推送更新* sendAllAngularObjects(note, user, conn);  } **else** {  removeConnectionFromAllNote(conn);  conn.send(serializeMessage(**new** Message(OP.***NOTE***).put(**"note"**, **null**)));  }  }   **private void** updateNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** SchedulerException, IOException {  String noteId = (String) fromMessage.get(**"id"**);  String name = (String) fromMessage.get(**"name"**);  Map<String, Object> config = (Map<String, Object>) fromMessage.get(**"config"**);  **if** (noteId == **null**) {  **return**;  }  **if** (config == **null**) {  **return**;  }   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"update"**)) {  **return**;  }   Note note = notebook.getNote(noteId);  **if** (note != **null**) {  **boolean** cronUpdated = isCronUpdated(config, note.getConfig());  note.setName(name);  note.setConfig(config);  **if** (cronUpdated) {  notebook.refreshCron(note.getId());  }   AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  note.persist(subject);  broadcast(note.getId(), **new** Message(OP.***NOTE\_UPDATED***).put(**"name"**, name).put(**"config"**, config)  .put(**"info"**, note.getInfo()));  broadcastNoteList(subject, userAndRoles);  }  }   **private void** updatePersonalizedMode(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** SchedulerException, IOException {  String noteId = (String) fromMessage.get(**"id"**);  String personalized = (String) fromMessage.get(**"personalized"**);  **boolean** isPersonalized = personalized.equals(**"true"**) ? **true** : **false**;   **if** (noteId == **null**) {  **return**;  }   **if** (!hasParagraphOwnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"persoanlized"**)) {  **return**;  }   Note note = notebook.getNote(noteId);  **if** (note != **null**) {  note.setPersonalizedMode(isPersonalized);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  note.persist(subject);  broadcastNote(note);  }  }   **private void** renameNote(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  renameNote(conn, userAndRoles, notebook, fromMessage, **"rename"**);  }   *// 修改Note的名称* **private void** renameNote(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage, String op)  **throws** SchedulerException, IOException {  String noteId = (String) fromMessage.get(**"id"**);  String name = (String) fromMessage.get(**"name"**);   **if** (noteId == **null**) {  **return**;  }   **if** (!hasParagraphOwnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"rename"**)) {  **return**;  }   Note note = notebook.getNote(noteId);  **if** (note != **null**) {  note.setName(name);   AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  note.persist(subject);  broadcastNote(note);  broadcastNoteList(subject, userAndRoles);  }  }   **private void** renameFolder(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  renameFolder(conn, userAndRoles, notebook, fromMessage, **"rename"**);  }   **private void** renameFolder(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage, String op)  **throws** SchedulerException, IOException {  String oldFolderId = (String) fromMessage.get(**"id"**);  String newFolderId = (String) fromMessage.get(**"name"**);   **if** (oldFolderId == **null**) {  **return**;  }   **for** (Note note : notebook.getNotesUnderFolder(oldFolderId)) {  String noteId = note.getId();  **if** (!hasParagraphOwnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, op + **" folder of '"** + note.getName() + **"'"**)) {  **return**;  }  }   Folder oldFolder = notebook.renameFolder(oldFolderId, newFolderId);   **if** (oldFolder != **null**) {  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);   List<Note> renamedNotes = oldFolder.getNotesRecursively();  **for** (Note note : renamedNotes) {  note.persist(subject);  broadcastNote(note);  }   broadcastNoteList(subject, userAndRoles);  }  }   **private boolean** isCronUpdated(Map<String, Object> configA, Map<String, Object> configB) {  **boolean** cronUpdated = **false**;  **if** (configA.get(**"cron"**) != **null** && configB.get(**"cron"**) != **null** && configA.get(**"cron"**)  .equals(configB.get(**"cron"**))) {  cronUpdated = **true**;  } **else if** (configA.get(**"cron"**) == **null** && configB.get(**"cron"**) == **null**) {  cronUpdated = **false**;  } **else if** (configA.get(**"cron"**) != **null** || configB.get(**"cron"**) != **null**) {  cronUpdated = **true**;  }   **return** cronUpdated;  }   *// 创建一个Note* **private void** createNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message message) **throws** IOException {  AuthenticationInfo subject = **new** AuthenticationInfo(message.**principal**);   **try** {  Note note = **null**;   String defaultInterpreterId = (String) message.get(**"defaultInterpreterId"**);  **if** (!StringUtils.*isEmpty*(defaultInterpreterId)) {  List<String> interpreterSettingIds = **new** LinkedList<>();  interpreterSettingIds.add(defaultInterpreterId);  **for** (String interpreterSettingId : notebook.getInterpreterSettingManager().getInterpreterSettingIds()) {  **if** (!interpreterSettingId.equals(defaultInterpreterId)) {  interpreterSettingIds.add(interpreterSettingId);  }  }  note = notebook.createNote(interpreterSettingIds, subject);  } **else** {  note = notebook.createNote(subject);  }   note.addNewParagraph(subject); *// it's an empty note. so add one paragraph* **if** (message != **null**) {  String noteName = (String) message.get(**"name"**);  **if** (StringUtils.*isEmpty*(noteName)) {  noteName = **"Note "** + note.getId();  }  note.setName(noteName);  }   note.persist(subject);  addConnectionToNote(note.getId(), (NotebookSocket) conn);  conn.send(serializeMessage(**new** Message(OP.***NEW\_NOTE***).put(**"note"**, note)));  } **catch** (FileSystemException e) {  ***LOG***.error(**"Exception from createNote"**, e);  conn.send(serializeMessage(**new** Message(OP.***ERROR\_INFO***).put(**"info"**,  **"Oops! There is something wrong with the notebook file system. "** + **"Please check the logs for more details."**)));  **return**;  }  broadcastNoteList(subject, userAndRoles);  }   *// 删除一个Note* **private void** removeNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  String noteId = (String) fromMessage.get(**"id"**);  **if** (noteId == **null**) {  **return**;  }   **if** (!hasParagraphOwnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"remove"**)) {  **return**;  }   AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  notebook.removeNote(noteId, subject);  removeNote(noteId);  broadcastNoteList(subject, userAndRoles);  }   *// 删除文件夹下的所有Note* **private void** removeFolder(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  String folderId = (String) fromMessage.get(**"id"**);  **if** (folderId == **null**) {  **return**;  }   List<Note> notes = notebook.getNotesUnderFolder(folderId);  **for** (Note note : notes) {  String noteId = note.getId();   **if** (!hasParagraphOwnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"remove folder of '"** + note.getName() + **"'"**)) {  **return**;  }  }   AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  **for** (Note note : notes) {  notebook.removeNote(note.getId(), subject);  removeNote(note.getId());  }  broadcastNoteList(subject, userAndRoles);  }   *// 就是修改一些名称* **private void** moveNoteToTrash(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  String noteId = (String) fromMessage.get(**"id"**);  **if** (noteId == **null**) {  **return**;  }   Note note = notebook.getNote(noteId);  **if** (note != **null** && !note.isTrash()){  fromMessage.put(**"name"**, Folder.***TRASH\_FOLDER\_ID*** + **"/"** + note.getName());  renameNote(conn, userAndRoles, notebook, fromMessage, **"move"**);  notebook.moveNoteToTrash(note.getId());  }  }   *// 将整个文件夹放到回收站* **private void** moveFolderToTrash(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  String folderId = (String) fromMessage.get(**"id"**);  **if** (folderId == **null**) {  **return**;  }   Folder folder = notebook.getFolder(folderId);  **if** (folder != **null** && !folder.isTrash()) {  String trashFolderId = Folder.***TRASH\_FOLDER\_ID*** + **"/"** + folderId;  **if** (notebook.hasFolder(trashFolderId)){  DateTime currentDate = **new** DateTime();  DateTimeFormatter formatter = DateTimeFormat.*forPattern*(**"yyyy-MM-dd HH:mm:ss"**);  trashFolderId += Folder.***TRASH\_FOLDER\_CONFLICT\_INFIX*** + formatter.print(currentDate);  }   fromMessage.put(**"name"**, trashFolderId);  renameFolder(conn, userAndRoles, notebook, fromMessage, **"move"**);  }  }   **private void** restoreNote(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  String noteId = (String) fromMessage.get(**"id"**);   **if** (noteId == **null**) {  **return**;  }   Note note = notebook.getNote(noteId);  **if** (note != **null** && note.isTrash()) {  fromMessage.put(**"name"**, note.getName().replaceFirst(Folder.***TRASH\_FOLDER\_ID*** + **"/"**, **""**));  renameNote(conn, userAndRoles, notebook, fromMessage, **"restore"**);  }  }   **private void** restoreFolder(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  String folderId = (String) fromMessage.get(**"id"**);   **if** (folderId == **null**) {  **return**;  }   Folder folder = notebook.getFolder(folderId);  **if** (folder != **null** && folder.isTrash()) {  String restoreName = folder.getId().replaceFirst(Folder.***TRASH\_FOLDER\_ID*** + **"/"**, **""**).trim();   *// if the folder had conflict when it had moved to trash before* Pattern p = Pattern.*compile*(**"\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}$"**);  Matcher m = p.matcher(restoreName);  restoreName = m.replaceAll(**""**).trim();   fromMessage.put(**"name"**, restoreName);  renameFolder(conn, userAndRoles, notebook, fromMessage, **"restore"**);  }  }   **private void** restoreAll(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  Folder trashFolder = notebook.getFolder(Folder.***TRASH\_FOLDER\_ID***);  **if** (trashFolder != **null**) {  fromMessage.**data** = **new** HashMap<>();  fromMessage.put(**"id"**, Folder.***TRASH\_FOLDER\_ID***);  fromMessage.put(**"name"**, Folder.***ROOT\_FOLDER\_ID***);  renameFolder(conn, userAndRoles, notebook, fromMessage, **"restore trash"**);  }  }   **private void** emptyTrash(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** SchedulerException, IOException {  fromMessage.**data** = **new** HashMap<>();  fromMessage.put(**"id"**, Folder.***TRASH\_FOLDER\_ID***);  removeFolder(conn, userAndRoles, notebook, fromMessage);  }   *// 保存一个片段* **private void** updateParagraph(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** IOException {  String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }   Map<String, Object> params = (Map<String, Object>) fromMessage.get(**"params"**);  Map<String, Object> config = (Map<String, Object>) fromMessage.get(**"config"**);  String noteId = getOpenNoteId(conn);  **if** (noteId == **null**) {  noteId = (String) fromMessage.get(**"noteId"**);  }   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   **final** Note note = notebook.getNote(noteId);  Paragraph p = note.getParagraph(paragraphId);   p.**settings**.setParams(params);  p.setConfig(config);  p.setTitle((String) fromMessage.get(**"title"**));  p.setText((String) fromMessage.get(**"paragraph"**));   AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  **if** (note.isPersonalizedMode()) {  p = p.getUserParagraph(subject.getUser());  p.**settings**.setParams(params);  p.setConfig(config);  p.setTitle((String) fromMessage.get(**"title"**));  p.setText((String) fromMessage.get(**"paragraph"**));  }    note.persist(subject);   **if** (note.isPersonalizedMode()) {  Map<String, Paragraph> userParagraphMap =  note.getParagraph(paragraphId).getUserParagraphMap();  broadcastParagraphs(userParagraphMap, p);  } **else** {  broadcastParagraph(note, p);  }  }   *// 克隆一个Note* **private void** cloneNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException, CloneNotSupportedException {  String noteId = getOpenNoteId(conn);  String name = (String) fromMessage.get(**"name"**);  Note newNote = notebook.cloneNote(noteId, name, **new** AuthenticationInfo(fromMessage.**principal**));  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  addConnectionToNote(newNote.getId(), (NotebookSocket) conn);  conn.send(serializeMessage(**new** Message(OP.***NEW\_NOTE***).put(**"note"**, newNote)));  broadcastNoteList(subject, userAndRoles);  }   **private void** clearAllParagraphOutput(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** IOException {  **final** String noteId = (String) fromMessage.get(**"id"**);  **if** (StringUtils.*isBlank*(noteId)) {  **return**;  }   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"clear output"**)) {  **return**;  }   Note note = notebook.getNote(noteId);  note.clearAllParagraphOutput();  broadcastNote(note);  }   *// 导入一个Note* **protected** Note importNote(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  Note note = **null**;  **if** (fromMessage != **null**) {  String noteName = (String) ((Map) fromMessage.get(**"note"**)).get(**"name"**);  String noteJson = *gson*.toJson(fromMessage.get(**"note"**));  AuthenticationInfo subject = **null**;  **if** (fromMessage.**principal** != **null**) {  subject = **new** AuthenticationInfo(fromMessage.**principal**);  } **else** {  subject = **new** AuthenticationInfo(**"anonymous"**);  }  note = notebook.importNote(noteJson, noteName, subject);  note.persist(subject);  broadcastNote(note);  broadcastNoteList(subject, userAndRoles);  }  **return** note;  }   **private void** removeParagraph(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  **final** String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }  String noteId = getOpenNoteId(conn);   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   **final** Note note = notebook.getNote(noteId);   */\*\* Don't allow removing paragraph when there is only one paragraph in the Notebook \*/* **if** (note.getParagraphCount() > 1) {  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  Paragraph para = note.removeParagraph(subject.getUser(), paragraphId);  note.persist(subject);  **if** (para != **null**) {  broadcast(note.getId(), **new** Message(OP.***PARAGRAPH\_REMOVED***).  put(**"id"**, para.getId()));  }  }  }   **private void** clearParagraphOutput(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** IOException {  **final** String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }   String noteId = getOpenNoteId(conn);  **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   **final** Note note = notebook.getNote(noteId);  **if** (note.isPersonalizedMode()) {  String user = fromMessage.**principal**;  Paragraph p = note.clearPersonalizedParagraphOutput(paragraphId, user);  unicastParagraph(note, p, user);  } **else** {  note.clearParagraphOutput(paragraphId);  Paragraph paragraph = note.getParagraph(paragraphId);  broadcastParagraph(note, paragraph);  }  }   **private void** completion(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  String paragraphId = (String) fromMessage.get(**"id"**);  String buffer = (String) fromMessage.get(**"buf"**);  **int** cursor = (**int**) Double.*parseDouble*(fromMessage.get(**"cursor"**).toString());  Message resp = **new** Message(OP.***COMPLETION\_LIST***).put(**"id"**, paragraphId);  **if** (paragraphId == **null**) {  conn.send(serializeMessage(resp));  **return**;  }   **final** Note note = notebook.getNote(getOpenNoteId(conn));  List<InterpreterCompletion> candidates = note.completion(paragraphId, buffer, cursor);  resp.put(**"completions"**, candidates);  conn.send(serializeMessage(resp));  }   */\*\*  \* When angular object updated from client  \*  \** ***@param conn*** *the web socket.  \** ***@param notebook*** *the notebook.  \** ***@param fromMessage*** *the message.  \*/* **private void** angularObjectUpdated(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) {  String noteId = (String) fromMessage.get(**"noteId"**);  String paragraphId = (String) fromMessage.get(**"paragraphId"**);  String interpreterGroupId = (String) fromMessage.get(**"interpreterGroupId"**);  String varName = (String) fromMessage.get(**"name"**);  Object varValue = fromMessage.get(**"value"**);  String user = fromMessage.**principal**;  AngularObject ao = **null**;  **boolean** global = **false**;  *// propagate change to (Remote) AngularObjectRegistry* Note note = notebook.getNote(noteId);  **if** (note != **null**) {  List<InterpreterSetting> settings =  notebook.getInterpreterSettingManager().getInterpreterSettings(note.getId());  **for** (InterpreterSetting setting : settings) {  **if** (setting.getInterpreterGroup(user, note.getId()) == **null**) {  **continue**;  }  **if** (interpreterGroupId.equals(setting.getInterpreterGroup(user, note.getId())  .getId())) {  AngularObjectRegistry angularObjectRegistry =  setting.getInterpreterGroup(user, note.getId()).getAngularObjectRegistry();   *// first trying to get local registry* ao = angularObjectRegistry.get(varName, noteId, paragraphId);  **if** (ao == **null**) {  *// then try notebook scope registry* ao = angularObjectRegistry.get(varName, noteId, **null**);  **if** (ao == **null**) {  *// then try global scope registry* ao = angularObjectRegistry.get(varName, **null**, **null**);  **if** (ao == **null**) {  ***LOG***.warn(**"Object {} is not binded"**, varName);  } **else** {  *// path from client -> server* ao.set(varValue, **false**);  global = **true**;  }  } **else** {  *// path from client -> server* ao.set(varValue, **false**);  global = **false**;  }  } **else** {  ao.set(varValue, **false**);  global = **false**;  }  **break**;  }  }  }   **if** (global) { *// broadcast change to all web session that uses related  // interpreter.* **for** (Note n : notebook.getAllNotes()) {  List<InterpreterSetting> settings =  notebook.getInterpreterSettingManager().getInterpreterSettings(note.getId());  **for** (InterpreterSetting setting : settings) {  **if** (setting.getInterpreterGroup(user, n.getId()) == **null**) {  **continue**;  }  **if** (interpreterGroupId.equals(setting.getInterpreterGroup(user, n.getId())  .getId())) {  AngularObjectRegistry angularObjectRegistry =  setting.getInterpreterGroup(user, n.getId()).getAngularObjectRegistry();  **this**.broadcastExcept(n.getId(),  **new** Message(OP.***ANGULAR\_OBJECT\_UPDATE***).put(**"angularObject"**, ao)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, n.getId())  .put(**"paragraphId"**, ao.getParagraphId()), conn);  }  }  }  } **else** { *// broadcast to all web session for the note* **this**.broadcastExcept(note.getId(),  **new** Message(OP.***ANGULAR\_OBJECT\_UPDATE***).put(**"angularObject"**, ao)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, note.getId())  .put(**"paragraphId"**, ao.getParagraphId()), conn);  }  } **protected void** angularObjectClientBind(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** Exception {  String noteId = fromMessage.getType(**"noteId"**);  String varName = fromMessage.getType(**"name"**);  Object varValue = fromMessage.get(**"value"**);  String paragraphId = fromMessage.getType(**"paragraphId"**);  Note note = notebook.getNote(noteId);   **if** (paragraphId == **null**) {  **throw new** IllegalArgumentException(  **"target paragraph not specified for "** + **"angular value bind"**);  }   **if** (note != **null**) {  **final** InterpreterGroup interpreterGroup = findInterpreterGroupForParagraph(note, paragraphId);   **final** AngularObjectRegistry registry = interpreterGroup.getAngularObjectRegistry();  **if** (registry **instanceof** RemoteAngularObjectRegistry) {   RemoteAngularObjectRegistry remoteRegistry = (RemoteAngularObjectRegistry) registry;  pushAngularObjectToRemoteRegistry(noteId, paragraphId, varName, varValue, remoteRegistry,  interpreterGroup.getId(), conn);   } **else** {  pushAngularObjectToLocalRepo(noteId, paragraphId, varName, varValue, registry,  interpreterGroup.getId(), conn);  }  }  }   */\*\*  \* Remove the given Angular variable to the target  \* interpreter(s) angular registry given a noteId  \* and an optional list of paragraph id(s)  \*/* **protected void** angularObjectClientUnbind(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** Exception {  String noteId = fromMessage.getType(**"noteId"**);  String varName = fromMessage.getType(**"name"**);  String paragraphId = fromMessage.getType(**"paragraphId"**);  Note note = notebook.getNote(noteId);   **if** (paragraphId == **null**) {  **throw new** IllegalArgumentException(  **"target paragraph not specified for "** + **"angular value unBind"**);  }   **if** (note != **null**) {  **final** InterpreterGroup interpreterGroup = findInterpreterGroupForParagraph(note, paragraphId);   **final** AngularObjectRegistry registry = interpreterGroup.getAngularObjectRegistry();   **if** (registry **instanceof** RemoteAngularObjectRegistry) {  RemoteAngularObjectRegistry remoteRegistry = (RemoteAngularObjectRegistry) registry;  removeAngularFromRemoteRegistry(noteId, paragraphId, varName, remoteRegistry,  interpreterGroup.getId(), conn);  } **else** {  removeAngularObjectFromLocalRepo(noteId, paragraphId, varName, registry,  interpreterGroup.getId(), conn);  }  }  }   **private** InterpreterGroup findInterpreterGroupForParagraph(Note note, String paragraphId)  **throws** Exception {  **final** Paragraph paragraph = note.getParagraph(paragraphId);  **if** (paragraph == **null**) {  **throw new** IllegalArgumentException(**"Unknown paragraph with id : "** + paragraphId);  }  **return** paragraph.getCurrentRepl().getInterpreterGroup();  }   **private void** pushAngularObjectToRemoteRegistry(String noteId, String paragraphId, String varName,  Object varValue, RemoteAngularObjectRegistry remoteRegistry, String interpreterGroupId,  NotebookSocket conn) {   **final** AngularObject ao =  remoteRegistry.addAndNotifyRemoteProcess(varName, varValue, noteId, paragraphId);   **this**.broadcastExcept(noteId, **new** Message(OP.***ANGULAR\_OBJECT\_UPDATE***).put(**"angularObject"**, ao)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId), conn);  }   **private void** removeAngularFromRemoteRegistry(String noteId, String paragraphId, String varName,  RemoteAngularObjectRegistry remoteRegistry, String interpreterGroupId, NotebookSocket conn) {  **final** AngularObject ao =  remoteRegistry.removeAndNotifyRemoteProcess(varName, noteId, paragraphId);  **this**.broadcastExcept(noteId, **new** Message(OP.***ANGULAR\_OBJECT\_REMOVE***).put(**"angularObject"**, ao)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId), conn);  }   **private void** pushAngularObjectToLocalRepo(String noteId, String paragraphId, String varName,  Object varValue, AngularObjectRegistry registry, String interpreterGroupId,  NotebookSocket conn) {  AngularObject angularObject = registry.get(varName, noteId, paragraphId);  **if** (angularObject == **null**) {  angularObject = registry.add(varName, varValue, noteId, paragraphId);  } **else** {  angularObject.set(varValue, **true**);  }   **this**.broadcastExcept(noteId,  **new** Message(OP.***ANGULAR\_OBJECT\_UPDATE***).put(**"angularObject"**, angularObject)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId), conn);  }   **private void** removeAngularObjectFromLocalRepo(String noteId, String paragraphId, String varName,  AngularObjectRegistry registry, String interpreterGroupId, NotebookSocket conn) {  **final** AngularObject removed = registry.remove(varName, noteId, paragraphId);  **if** (removed != **null**) {  **this**.broadcastExcept(noteId,  **new** Message(OP.***ANGULAR\_OBJECT\_REMOVE***).put(**"angularObject"**, removed)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId), conn);  }  }   **private void** moveParagraph(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  **final** String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }   **final int** newIndex = (**int**) Double.*parseDouble*(fromMessage.get(**"index"**).toString());  String noteId = getOpenNoteId(conn);  **final** Note note = notebook.getNote(noteId);   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  note.moveParagraph(paragraphId, newIndex);  note.persist(subject);  broadcast(note.getId(),  **new** Message(OP.***PARAGRAPH\_MOVED***).put(**"id"**, paragraphId).put(**"index"**, newIndex));  }   **private** String insertParagraph(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage) **throws** IOException {  **final int** index = (**int**) Double.*parseDouble*(fromMessage.get(**"index"**).toString());  String noteId = getOpenNoteId(conn);  **final** Note note = notebook.getNote(noteId);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return null**;  }  Map<String, Object> config;  **if** (fromMessage.get(**"config"**) != **null**) {  config = (Map<String, Object>) fromMessage.get(**"config"**);  } **else** {  config = **new** HashMap<>();  }   Paragraph newPara = note.insertNewParagraph(index, subject);  newPara.setConfig(config);  note.persist(subject);  broadcastNewParagraph(note, newPara);   **return** newPara.getId();  }   **private void** copyParagraph(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  String newParaId = insertParagraph(conn, userAndRoles, notebook, fromMessage);   **if** (newParaId == **null**) {  **return**;  }  fromMessage.put(**"id"**, newParaId);   updateParagraph(conn, userAndRoles, notebook, fromMessage);  }   **private void** cancelParagraph(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  **final** String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }   String noteId = getOpenNoteId(conn);   **if** (!hasParagraphRunnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   **final** Note note = notebook.getNote(noteId);  Paragraph p = note.getParagraph(paragraphId);  p.abort();  }   **private void** runAllParagraphs(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook,  Message fromMessage) **throws** IOException {  **final** String noteId = (String) fromMessage.get(**"noteId"**);  **if** (StringUtils.*isBlank*(noteId)) {  **return**;  }   **if** (!hasParagraphRunnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"run all paragraphs"**)) {  **return**;  }   List<Map<String, Object>> paragraphs =  *gson*.fromJson(String.*valueOf*(fromMessage.**data**.get(**"paragraphs"**)),  **new** TypeToken<List<Map<String, Object>>>() {}.getType());   **for** (Map<String, Object> raw : paragraphs) {  String paragraphId = (String) raw.get(**"id"**);  **if** (paragraphId == **null**) {  **continue**;  }   String text = (String) raw.get(**"paragraph"**);  String title = (String) raw.get(**"title"**);  Map<String, Object> params = (Map<String, Object>) raw.get(**"params"**);  Map<String, Object> config = (Map<String, Object>) raw.get(**"config"**);   Note note = notebook.getNote(noteId);  Paragraph p = setParagraphUsingMessage(note, fromMessage,  paragraphId, text, title, params, config);   **if** (!persistAndExecuteSingleParagraph(conn, note, p, **true**)) {  *// stop execution when one paragraph fails.* **break**;  }  }  }   **private void** broadcastSpellExecution(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook, Message fromMessage)  **throws** IOException {   **final** String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }   String noteId = getOpenNoteId(conn);   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   String text = (String) fromMessage.get(**"paragraph"**);  String title = (String) fromMessage.get(**"title"**);  Status status = Status.*valueOf*((String) fromMessage.get(**"status"**));  Map<String, Object> params = (Map<String, Object>) fromMessage.get(**"params"**);  Map<String, Object> config = (Map<String, Object>) fromMessage.get(**"config"**);   **final** Note note = notebook.getNote(noteId);  Paragraph p = setParagraphUsingMessage(note, fromMessage, paragraphId,  text, title, params, config);  p.setResult(fromMessage.get(**"results"**));  p.setErrorMessage((String) fromMessage.get(**"errorMessage"**));  p.setStatusWithoutNotification(status);   *// Spell uses ISO 8601 formatted string generated from moment* String dateStarted = (String) fromMessage.get(**"dateStarted"**);  String dateFinished = (String) fromMessage.get(**"dateFinished"**);  SimpleDateFormat df = **new** SimpleDateFormat(**"yyyy-MM-dd'T'HH:mm:ss.SSSX"**);   **try** {  p.setDateStarted(df.parse(dateStarted));  } **catch** (ParseException e) {  ***LOG***.error(**"Failed parse dateStarted"**, e);  }   **try** {  p.setDateFinished(df.parse(dateFinished));  } **catch** (ParseException e) {  ***LOG***.error(**"Failed parse dateFinished"**, e);  }   addNewParagraphIfLastParagraphIsExecuted(note, p);  **if** (!persistNoteWithAuthInfo(conn, note, p)) {  **return**;  }   *// broadcast to other clients only* broadcastExcept(note.getId(),  **new** Message(OP.***RUN\_PARAGRAPH\_USING\_SPELL***).put(**"paragraph"**, p), conn);  }   **private void** runParagraph(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {  **final** String paragraphId = (String) fromMessage.get(**"id"**);  **if** (paragraphId == **null**) {  **return**;  }   String noteId = getOpenNoteId(conn);   **if** (!hasParagraphRunnerPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"write"**)) {  **return**;  }   *// 1. clear paragraph only if personalized,  // otherwise this will be handed in `onOutputClear`* **final** Note note = notebook.getNote(noteId);  **if** (note.isPersonalizedMode()) {  String user = fromMessage.**principal**;  Paragraph p = note.clearPersonalizedParagraphOutput(paragraphId, user);  unicastParagraph(note, p, user);  }   *// 2. set paragraph values* String text = (String) fromMessage.get(**"paragraph"**);  String title = (String) fromMessage.get(**"title"**);  Map<String, Object> params = (Map<String, Object>) fromMessage.get(**"params"**);  Map<String, Object> config = (Map<String, Object>) fromMessage.get(**"config"**);   Paragraph p = setParagraphUsingMessage(note, fromMessage, paragraphId,  text, title, params, config);   persistAndExecuteSingleParagraph(conn, note, p, **false**);  }   **private void** addNewParagraphIfLastParagraphIsExecuted(Note note, Paragraph p) {  *// if it's the last paragraph and not empty, let's add a new one* **boolean** isTheLastParagraph = note.isLastParagraph(p.getId());  **if** (!(Strings.*isNullOrEmpty*(p.getText()) ||  p.getText().trim().equals(p.getMagic())) &&  isTheLastParagraph) {  Paragraph newPara = note.addNewParagraph(p.getAuthenticationInfo());  broadcastNewParagraph(note, newPara);  }  }   */\*\*  \** ***@return*** *false if failed to save a note  \*/* **private boolean** persistNoteWithAuthInfo(NotebookSocket conn,  Note note, Paragraph p) **throws** IOException {  **try** {  note.persist(p.getAuthenticationInfo());  **return true**;  } **catch** (FileSystemException ex) {  ***LOG***.error(**"Exception from run"**, ex);  conn.send(serializeMessage(**new** Message(OP.***ERROR\_INFO***).put(**"info"**,  **"Oops! There is something wrong with the notebook file system. "** + **"Please check the logs for more details."**)));  *// don't run the paragraph when there is error on persisting the note information* **return false**;  }  }   **private boolean** persistAndExecuteSingleParagraph(NotebookSocket conn,  Note note, Paragraph p,  **boolean** blocking) **throws** IOException {  addNewParagraphIfLastParagraphIsExecuted(note, p);  **if** (!persistNoteWithAuthInfo(conn, note, p)) {  **return false**;  }   **try** {  **return** note.run(p.getId(), blocking);  } **catch** (Exception ex) {  ***LOG***.error(**"Exception from run"**, ex);  **if** (p != **null**) {  p.setReturn(**new** InterpreterResult(InterpreterResult.Code.***ERROR***, ex.getMessage()), ex);  p.setStatus(Status.***ERROR***);  broadcast(note.getId(), **new** Message(OP.***PARAGRAPH***).put(**"paragraph"**, p));  }  **return false**;  }  }   **private** Paragraph setParagraphUsingMessage(Note note, Message fromMessage, String paragraphId,  String text, String title, Map<String, Object> params,  Map<String, Object> config) {  Paragraph p = note.getParagraph(paragraphId);  p.setText(text);  p.setTitle(title);  AuthenticationInfo subject =  **new** AuthenticationInfo(fromMessage.**principal**, fromMessage.**roles**, fromMessage.**ticket**);  p.setAuthenticationInfo(subject);  p.**settings**.setParams(params);  p.setConfig(config);   **if** (note.isPersonalizedMode()) {  p = note.getParagraph(paragraphId);  p.setText(text);  p.setTitle(title);  p.setAuthenticationInfo(subject);  p.**settings**.setParams(params);  p.setConfig(config);  }   **return** p;  }   **private void** sendAllConfigurations(NotebookSocket conn, HashSet<String> userAndRoles,  Notebook notebook) **throws** IOException {  ZeppelinConfiguration conf = notebook.getConf();   Map<String, String> configurations =  conf.dumpConfigurations(conf, **new** ZeppelinConfiguration.ConfigurationKeyPredicate() {  @Override  **public boolean** apply(String key) {  **return** !key.contains(**"password"**) && !key.equals(  ZeppelinConfiguration.ConfVars.***ZEPPELIN\_NOTEBOOK\_AZURE\_CONNECTION\_STRING*** .getVarName());  }  });   conn.send(serializeMessage(  **new** Message(OP.***CONFIGURATIONS\_INFO***).put(**"configurations"**, configurations)));  }   **private void** checkpointNote(NotebookSocket conn, Notebook notebook, Message fromMessage)  **throws** IOException {  String noteId = (String) fromMessage.get(**"noteId"**);  String commitMessage = (String) fromMessage.get(**"commitMessage"**);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  Revision revision = notebook.checkpointNote(noteId, commitMessage, subject);  **if** (!Revision.*isEmpty*(revision)) {  List<Revision> revisions = notebook.listRevisionHistory(noteId, subject);  conn.send(  serializeMessage(**new** Message(OP.***LIST\_REVISION\_HISTORY***).put(**"revisionList"**, revisions)));  } **else** {  conn.send(serializeMessage(**new** Message(OP.***ERROR\_INFO***).put(**"info"**,  **"Couldn't checkpoint note revision: possibly storage doesn't support versioning. "** + **"Please check the logs for more details."**)));  }  }   **private void** listRevisionHistory(NotebookSocket conn, Notebook notebook, Message fromMessage)  **throws** IOException {  String noteId = (String) fromMessage.get(**"noteId"**);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  List<Revision> revisions = notebook.listRevisionHistory(noteId, subject);   conn.send(  serializeMessage(**new** Message(OP.***LIST\_REVISION\_HISTORY***).put(**"revisionList"**, revisions)));  }   **private void** setNoteRevision(NotebookSocket conn, HashSet<String> userAndRoles, Notebook notebook,  Message fromMessage) **throws** IOException {   String noteId = (String) fromMessage.get(**"noteId"**);  String revisionId = (String) fromMessage.get(**"revisionId"**);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);   **if** (!hasParagraphWriterPermission(conn, notebook, noteId,  userAndRoles, fromMessage.**principal**, **"update"**)) {  **return**;  }   Note headNote = **null**;  **boolean** setRevisionStatus;  **try** {  headNote = notebook.setNoteRevision(noteId, revisionId, subject);  setRevisionStatus = headNote != **null**;  } **catch** (Exception e) {  setRevisionStatus = **false**;  ***LOG***.error(**"Failed to set given note revision"**, e);  }  **if** (setRevisionStatus) {  notebook.loadNoteFromRepo(noteId, subject);  }   conn.send(serializeMessage(**new** Message(OP.***SET\_NOTE\_REVISION***).put(**"status"**, setRevisionStatus)));   **if** (setRevisionStatus) {  Note reloadedNote = notebook.getNote(headNote.getId());  broadcastNote(reloadedNote);  } **else** {  conn.send(serializeMessage(**new** Message(OP.***ERROR\_INFO***).put(**"info"**,  **"Couldn't set note to the given revision. "** + **"Please check the logs for more details."**)));  }  }   **private void** getNoteByRevision(NotebookSocket conn, Notebook notebook, Message fromMessage)  **throws** IOException {  String noteId = (String) fromMessage.get(**"noteId"**);  String revisionId = (String) fromMessage.get(**"revisionId"**);  AuthenticationInfo subject = **new** AuthenticationInfo(fromMessage.**principal**);  Note revisionNote = notebook.getNoteByRevision(noteId, revisionId, subject);  conn.send(serializeMessage(  **new** Message(OP.***NOTE\_REVISION***).put(**"noteId"**, noteId).put(**"revisionId"**, revisionId)  .put(**"note"**, revisionNote)));  }   */\*\*  \* This callback is for the paragraph that runs on ZeppelinServer  \*  \** ***@param output*** *output to append  \*/* @Override  **public void** onOutputAppend(String noteId, String paragraphId, **int** index, String output) {  Message msg = **new** Message(OP.***PARAGRAPH\_APPEND\_OUTPUT***).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId).put(**"index"**, index).put(**"data"**, output);  broadcast(noteId, msg);  }   */\*\*  \* This callback is for the paragraph that runs on ZeppelinServer  \*  \** ***@param output*** *output to update (replace)  \*/* @Override  **public void** onOutputUpdated(String noteId, String paragraphId, **int** index,  InterpreterResult.Type type, String output) {  Message msg = **new** Message(OP.***PARAGRAPH\_UPDATE\_OUTPUT***).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId).put(**"index"**, index).put(**"type"**, type).put(**"data"**, output);  Note note = notebook().getNote(noteId);  **if** (note.isPersonalizedMode()) {  String user = note.getParagraph(paragraphId).getUser();  **if** (**null** != user) {  multicastToUser(user, msg);  }  } **else** {  broadcast(noteId, msg);  }  }    */\*\*  \* This callback is for the paragraph that runs on ZeppelinServer  \*/* @Override  **public void** onOutputClear(String noteId, String paragraphId) {  Notebook notebook = notebook();  **final** Note note = notebook.getNote(noteId);  note.clearParagraphOutput(paragraphId);  Paragraph paragraph = note.getParagraph(paragraphId);  broadcastParagraph(note, paragraph);  }   */\*\*  \* When application append output  \*/* @Override  **public void** onOutputAppend(String noteId, String paragraphId, **int** index, String appId,  String output) {  Message msg =  **new** Message(OP.***APP\_APPEND\_OUTPUT***).put(**"noteId"**, noteId).put(**"paragraphId"**, paragraphId)  .put(**"index"**, index).put(**"appId"**, appId).put(**"data"**, output);  broadcast(noteId, msg);  }   */\*\*  \* When application update output  \*/* @Override  **public void** onOutputUpdated(String noteId, String paragraphId, **int** index, String appId,  InterpreterResult.Type type, String output) {  Message msg =  **new** Message(OP.***APP\_UPDATE\_OUTPUT***).put(**"noteId"**, noteId).put(**"paragraphId"**, paragraphId)  .put(**"index"**, index).put(**"type"**, type).put(**"appId"**, appId).put(**"data"**, output);  broadcast(noteId, msg);  }   @Override  **public void** onLoad(String noteId, String paragraphId, String appId, HeliumPackage pkg) {  Message msg = **new** Message(OP.***APP\_LOAD***).put(**"noteId"**, noteId).put(**"paragraphId"**, paragraphId)  .put(**"appId"**, appId).put(**"pkg"**, pkg);  broadcast(noteId, msg);  }   @Override  **public void** onStatusChange(String noteId, String paragraphId, String appId, String status) {  Message msg =  **new** Message(OP.***APP\_STATUS\_CHANGE***).put(**"noteId"**, noteId).put(**"paragraphId"**, paragraphId)  .put(**"appId"**, appId).put(**"status"**, status);  broadcast(noteId, msg);  }   @Override  **public void** onGetParagraphRunners(String noteId, String paragraphId,  RemoteWorksEventListener callback) {  Notebook notebookIns = notebook();  List<InterpreterContextRunner> runner = **new** LinkedList<>();   **if** (notebookIns == **null**) {  ***LOG***.info(**"intepreter request notebook instance is null"**);  callback.onFinished(notebookIns);  }   **try** {  Note note = notebookIns.getNote(noteId);  **if** (note != **null**) {  **if** (paragraphId != **null**) {  Paragraph paragraph = note.getParagraph(paragraphId);  **if** (paragraph != **null**) {  runner.add(paragraph.getInterpreterContextRunner());  }  } **else** {  **for** (Paragraph p : note.getParagraphs()) {  runner.add(p.getInterpreterContextRunner());  }  }  }  callback.onFinished(runner);  } **catch** (NullPointerException e) {  ***LOG***.warn(e.getMessage());  callback.onError();  }  }   @Override  **public void** onRemoteRunParagraph(String noteId, String paragraphId) **throws** Exception {  Notebook notebookIns = notebook();  **try** {  **if** (notebookIns == **null**) {  **throw new** Exception(**"onRemoteRunParagraph notebook instance is null"**);  }  Note noteIns = notebookIns.getNote(noteId);  **if** (noteIns == **null**) {  **throw new** Exception(String.*format*(**"Can't found note id %s"**, noteId));  }   Paragraph paragraph = noteIns.getParagraph(paragraphId);  **if** (paragraph == **null**) {  **throw new** Exception(String.*format*(**"Can't found paragraph %s %s"**, noteId, paragraphId));  }   Set<String> userAndRoles = Sets.*newHashSet*();  userAndRoles.add(SecurityUtils.*getPrincipal*());  userAndRoles.addAll(SecurityUtils.*getRoles*());  **if** (!notebookIns.getNotebookAuthorization().hasRunAuthorization(userAndRoles, noteId)) {  **throw new** ForbiddenException(String.*format*(**"can't execute note %s"**, noteId));  }   AuthenticationInfo subject = **new** AuthenticationInfo(SecurityUtils.*getPrincipal*());  paragraph.setAuthenticationInfo(subject);   noteIns.run(paragraphId);   } **catch** (Exception e) {  **throw** e;  }  }   */\*\*  \* Notebook Information Change event  \*/* **public static class** NotebookInformationListener **implements** NotebookEventListener {   **private** NotebookServer **notebookServer**;   **public** NotebookInformationListener(NotebookServer notebookServer) {  **this**.**notebookServer** = notebookServer;  }   @Override  **public void** onParagraphRemove(Paragraph p) {  **try** {  **notebookServer**.broadcastUpdateNoteJobInfo(System.*currentTimeMillis*() - 5000);  } **catch** (IOException ioe) {  ***LOG***.error(**"can not broadcast for job manager {}"**, ioe.getMessage());  }  }   @Override  **public void** onNoteRemove(Note note) {  **try** {  **notebookServer**.broadcastUpdateNoteJobInfo(System.*currentTimeMillis*() - 5000);  } **catch** (IOException ioe) {  ***LOG***.error(**"can not broadcast for job manager {}"**, ioe.getMessage());  }   List<Map<String, Object>> notesInfo = **new** LinkedList<>();  Map<String, Object> info = **new** HashMap<>();  info.put(**"noteId"**, note.getId());  *// set paragraphs* List<Map<String, Object>> paragraphsInfo = **new** LinkedList<>();   *// notebook json object root information.* info.put(**"isRunningJob"**, **false**);  info.put(**"unixTimeLastRun"**, 0);  info.put(**"isRemoved"**, **true**);  info.put(**"paragraphs"**, paragraphsInfo);  notesInfo.add(info);   Map<String, Object> response = **new** HashMap<>();  response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, notesInfo);   **notebookServer**.broadcast(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(),  **new** Message(OP.***LIST\_UPDATE\_NOTE\_JOBS***).put(**"noteRunningJobs"**, response));   }   @Override  **public void** onParagraphCreate(Paragraph p) {  Notebook notebook = **notebookServer**.notebook();  List<Map<String, Object>> notebookJobs = notebook.getJobListByParagraphId(p.getId());  Map<String, Object> response = **new** HashMap<>();  response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, notebookJobs);   **notebookServer**.broadcast(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(),  **new** Message(OP.***LIST\_UPDATE\_NOTE\_JOBS***).put(**"noteRunningJobs"**, response));  }   @Override  **public void** onNoteCreate(Note note) {  Notebook notebook = **notebookServer**.notebook();  List<Map<String, Object>> notebookJobs = notebook.getJobListByNoteId(note.getId());  Map<String, Object> response = **new** HashMap<>();  response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, notebookJobs);   **notebookServer**.broadcast(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(),  **new** Message(OP.***LIST\_UPDATE\_NOTE\_JOBS***).put(**"noteRunningJobs"**, response));  }   @Override  **public void** onParagraphStatusChange(Paragraph p, Status status) {  Notebook notebook = **notebookServer**.notebook();  List<Map<String, Object>> notebookJobs = notebook.getJobListByParagraphId(p.getId());   Map<String, Object> response = **new** HashMap<>();  response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, notebookJobs);   **notebookServer**.broadcast(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(),  **new** Message(OP.***LIST\_UPDATE\_NOTE\_JOBS***).put(**"noteRunningJobs"**, response));  }   @Override  **public void** onUnbindInterpreter(Note note, InterpreterSetting setting) {  Notebook notebook = **notebookServer**.notebook();  List<Map<String, Object>> notebookJobs = notebook.getJobListByNoteId(note.getId());  Map<String, Object> response = **new** HashMap<>();  response.put(**"lastResponseUnixTime"**, System.*currentTimeMillis*());  response.put(**"jobs"**, notebookJobs);   **notebookServer**.broadcast(JOB\_MANAGER\_SERVICE.***JOB\_MANAGER\_PAGE***.getKey(),  **new** Message(OP.***LIST\_UPDATE\_NOTE\_JOBS***).put(**"noteRunningJobs"**, response));  }  }    */\*\*  \* Need description here.  \*/* **public static class** ParagraphListenerImpl **implements** ParagraphJobListener {   **private** NotebookServer **notebookServer**;  **private** Note **note**;   **public** ParagraphListenerImpl(NotebookServer notebookServer, Note note) {  **this**.**notebookServer** = notebookServer;  **this**.**note** = note;  }   @Override  **public void** onProgressUpdate(Job job, **int** progress) {  **notebookServer**.broadcast(**note**.getId(),  **new** Message(OP.***PROGRESS***).put(**"id"**, job.getId()).put(**"progress"**, progress));  }   @Override  **public void** beforeStatusChange(Job job, Status before, Status after) {  }   @Override  **public void** afterStatusChange(Job job, Status before, Status after) {  **if** (after == Status.***ERROR***) {  **if** (job.getException() != **null**) {  ***LOG***.error(**"Error"**, job.getException());  }  }   **if** (job.isTerminated()) {  **if** (job.getStatus() == Status.***FINISHED***) {  ***LOG***.info(**"Job {} is finished successfully, status: {}"**, job.getId(), job.getStatus());  } **else** {  ***LOG***.warn(**"Job {} is finished, status: {}, exception: {}, result: {}"** , job.getId(),  job.getStatus(), job.getException(), job.getReturn());  }   **try** {  *//****TODO(khalid): may change interface for JobListener and pass subject from interpreter* note**.persist(job **instanceof** Paragraph ? ((Paragraph) job).getAuthenticationInfo() : **null**);  } **catch** (IOException e) {  ***LOG***.error(e.toString(), e);  }  }  **if** (job **instanceof** Paragraph) {  Paragraph p = (Paragraph) job;  p.setStatusToUserParagraph(job.getStatus());  **notebookServer**.broadcastParagraph(**note**, p);  }  **try** {  **notebookServer**.broadcastUpdateNoteJobInfo(System.*currentTimeMillis*() - 5000);  } **catch** (IOException e) {  ***LOG***.error(**"can not broadcast for job manager {}"**, e);  }  }   */\*\*  \* This callback is for paragraph that runs on RemoteInterpreterProcess  \*/* @Override  **public void** onOutputAppend(Paragraph paragraph, **int** idx, String output) {  Message msg =  **new** Message(OP.***PARAGRAPH\_APPEND\_OUTPUT***).put(**"noteId"**, paragraph.getNote().getId())  .put(**"paragraphId"**, paragraph.getId()).put(**"data"**, output);   **notebookServer**.broadcast(paragraph.getNote().getId(), msg);  }   */\*\*  \* This callback is for paragraph that runs on RemoteInterpreterProcess  \*/* @Override  **public void** onOutputUpdate(Paragraph paragraph, **int** idx, InterpreterResultMessage result) {  String output = result.getData();  Message msg =  **new** Message(OP.***PARAGRAPH\_UPDATE\_OUTPUT***).put(**"noteId"**, paragraph.getNote().getId())  .put(**"paragraphId"**, paragraph.getId()).put(**"data"**, output);   **notebookServer**.broadcast(paragraph.getNote().getId(), msg);  }   @Override  **public void** onOutputUpdateAll(Paragraph paragraph, List<InterpreterResultMessage> msgs) {  *//* ***TODO*** }  }   @Override  **public** ParagraphJobListener getParagraphJobListener(Note note) {  **return new** ParagraphListenerImpl(**this**, note);  }   **public** NotebookEventListener getNotebookInformationListener() {  **return new** NotebookInformationListener(**this**);  }   *// 更新前端数据* **private void** sendAllAngularObjects(Note note, String user, NotebookSocket conn)  **throws** IOException {  List<InterpreterSetting> settings = notebook().getInterpreterSettingManager().getInterpreterSettings(note.getId());  **if** (settings == **null** || settings.size() == 0) {  **return**;  }   **for** (InterpreterSetting intpSetting : settings) {  **if** (intpSetting.getInterpreterGroup(user, note.getId()) == **null**) {  **continue**;  }  AngularObjectRegistry registry =  intpSetting.getInterpreterGroup(user, note.getId()).getAngularObjectRegistry();  List<AngularObject> objects = registry.getAllWithGlobal(note.getId());  **for** (AngularObject object : objects) {  conn.send(serializeMessage(  **new** Message(OP.***ANGULAR\_OBJECT\_UPDATE***).put(**"angularObject"**, object)  .put(**"interpreterGroupId"**,  intpSetting.getInterpreterGroup(user, note.getId()).getId())  .put(**"noteId"**, note.getId()).put(**"paragraphId"**, object.getParagraphId())));  }  }  }   @Override  **public void** onAdd(String interpreterGroupId, AngularObject object) {  onUpdate(interpreterGroupId, object);  }   @Override  **public void** onUpdate(String interpreterGroupId, AngularObject object) {  Notebook notebook = notebook();  **if** (notebook == **null**) {  **return**;  }   List<Note> notes = notebook.getAllNotes();  **for** (Note note : notes) {  **if** (object.getNoteId() != **null** && !note.getId().equals(object.getNoteId())) {  **continue**;  }   List<InterpreterSetting> intpSettings =  notebook.getInterpreterSettingManager().getInterpreterSettings(note.getId());  **if** (intpSettings.isEmpty()) {  **continue**;  }   broadcast(note.getId(), **new** Message(OP.***ANGULAR\_OBJECT\_UPDATE***).put(**"angularObject"**, object)  .put(**"interpreterGroupId"**, interpreterGroupId).put(**"noteId"**, note.getId())  .put(**"paragraphId"**, object.getParagraphId()));  }  }   @Override  **public void** onRemove(String interpreterGroupId, String name, String noteId, String paragraphId) {  Notebook notebook = notebook();  List<Note> notes = notebook.getAllNotes();  **for** (Note note : notes) {  **if** (noteId != **null** && !note.getId().equals(noteId)) {  **continue**;  }   List<String> settingIds =  notebook.getInterpreterSettingManager().getInterpreterBinding(note.getId());  **for** (String id : settingIds) {  **if** (interpreterGroupId.contains(id)) {  broadcast(note.getId(),  **new** Message(OP.***ANGULAR\_OBJECT\_REMOVE***).put(**"name"**, name).put(**"noteId"**, noteId)  .put(**"paragraphId"**, paragraphId));  **break**;  }  }  }  }   **private void** getEditorSetting(NotebookSocket conn, Message fromMessage) **throws** IOException {  String paragraphId = (String) fromMessage.get(**"paragraphId"**);  String replName = (String) fromMessage.get(**"magic"**);  String noteId = getOpenNoteId(conn);  String user = fromMessage.**principal**;  Message resp = **new** Message(OP.***EDITOR\_SETTING***);  resp.put(**"paragraphId"**, paragraphId);  Interpreter interpreter =  notebook().getInterpreterFactory().getInterpreter(user, noteId, replName);  resp.put(**"editor"**, notebook().getInterpreterSettingManager().  getEditorSetting(interpreter, user, noteId, replName));  conn.send(serializeMessage(resp));  }   **private void** getInterpreterSettings(NotebookSocket conn, AuthenticationInfo subject)  **throws** IOException {  List<InterpreterSetting> availableSettings = notebook().getInterpreterSettingManager().get();  conn.send(serializeMessage(  **new** Message(OP.***INTERPRETER\_SETTINGS***).put(**"interpreterSettings"**, availableSettings)));  }   @Override  **public void** onMetaInfosReceived(String settingId, Map<String, String> metaInfos) {  InterpreterSetting interpreterSetting =  notebook().getInterpreterSettingManager().get(settingId);  interpreterSetting.setInfos(metaInfos);  }   **private void** switchConnectionToWatcher(NotebookSocket conn, Message messagereceived)  **throws** IOException {  **if** (!isSessionAllowedToSwitchToWatcher(conn)) {  ***LOG***.error(**"Cannot switch this client to watcher, invalid security key"**);  **return**;  }  ***LOG***.info(**"Going to add {} to watcher socket"**, conn);  *// add the connection to the watcher.* **if** (**watcherSockets**.contains(conn)) {  ***LOG***.info(**"connection alrerady present in the watcher"**);  **return**;  }  **watcherSockets**.add(conn);   *// remove this connection from regular zeppelin ws usage.* removeConnectionFromAllNote(conn);  **connectedSockets**.remove(conn);  removeUserConnection(conn.getUser(), conn);  }   **private boolean** isSessionAllowedToSwitchToWatcher(NotebookSocket session) {  String watcherSecurityKey = session.getRequest().getHeader(WatcherSecurityKey.***HTTP\_HEADER***);  **return** !(StringUtils.*isBlank*(watcherSecurityKey) || !watcherSecurityKey  .equals(WatcherSecurityKey.*getKey*()));  }   */\*\*  \* Send websocket message to all connections regardless of notebook id  \*/* **private void** broadcastToAllConnections(String serialized) {  broadcastToAllConnectionsExcept(**null**, serialized);  }   **private void** broadcastToAllConnectionsExcept(NotebookSocket exclude, String serialized) {  **synchronized** (**connectedSockets**) {  **for** (NotebookSocket conn: **connectedSockets**) {  **if** (exclude != **null** && exclude.equals(conn)) {  **continue**;  }   **try** {  conn.send(serialized);  } **catch** (IOException e) {  ***LOG***.error(**"Cannot broadcast message to watcher"**, e);  }  }  }  }   **private void** broadcastToWatchers(String noteId, String subject, Message message) {  **synchronized** (**watcherSockets**) {  **for** (NotebookSocket watcher : **watcherSockets**) {  **try** {  watcher.send(  WatcherMessage.*builder*(noteId).subject(subject).message(serializeMessage(message))  .build().toJson());  } **catch** (IOException e) {  ***LOG***.error(**"Cannot broadcast message to watcher"**, e);  }  }  }  }   @Override  **public void** onParaInfosReceived(String noteId, String paragraphId,  String interpreterSettingId, Map<String, String> metaInfos) {  Note note = notebook().getNote(noteId);  **if** (note != **null**) {  Paragraph paragraph = note.getParagraph(paragraphId);  **if** (paragraph != **null**) {  InterpreterSetting setting = notebook().getInterpreterSettingManager()  .get(interpreterSettingId);  setting.addNoteToPara(noteId, paragraphId);  String label = metaInfos.get(**"label"**);  String tooltip = metaInfos.get(**"tooltip"**);  List<String> keysToRemove = Arrays.*asList*(**"noteId"**, **"paraId"**, **"label"**, **"tooltip"**);  **for** (String removeKey : keysToRemove) {  metaInfos.remove(removeKey);  }  paragraph  .updateRuntimeInfos(label, tooltip, metaInfos, setting.getGroup(), setting.getId());  broadcast(  note.getId(),  **new** Message(OP.***PARAS\_INFO***).put(**"id"**, paragraphId).put(**"infos"**,  paragraph.getRuntimeInfos()));  }  }  }   **public void** clearParagraphRuntimeInfo(InterpreterSetting setting) {  Map<String, Set<String>> noteIdAndParaMap = setting.getNoteIdAndParaMap();  **if** (noteIdAndParaMap != **null** && !noteIdAndParaMap.isEmpty()) {  **for** (String noteId : noteIdAndParaMap.keySet()) {  Set<String> paraIdSet = noteIdAndParaMap.get(noteId);  **if** (paraIdSet != **null** && !paraIdSet.isEmpty()) {  **for** (String paraId : paraIdSet) {  Note note = notebook().getNote(noteId);  **if** (note != **null**) {  Paragraph paragraph = note.getParagraph(paraId);  **if** (paragraph != **null**) {  paragraph.clearRuntimeInfo(setting.getId());  broadcast(noteId, **new** Message(OP.***PARAGRAPH***).put(**"paragraph"**, paragraph));  }  }  }  }  }  }  setting.clearNoteIdAndParaMap();  } } |

# 前台主程序

|  |
| --- |
| */\*  \* Copyright (c) 2018. 联思智云（北京）科技有限公司. All rights reserved.   \*/* **import** { Component, OnInit } **from '@angular/core'**; **import** {EventService1} **from "../../service/event/event.service"**; **import** {WebsocketMessageService} **from "../../service/websocket/websocket-message.service"**; **import** {ActivatedRoute, Params, Router} **from "@angular/router"**; **import** {NoteVarShareService} **from "../../service/note-var-share/note-var-share.service"**; **import** {HttpClient} **from "@angular/common/http"**; **import** {BaseUrlService} **from "../../service/base-url/base-url.service"**; **import** {GlobalService} **from "../../service/global/global.service"**; **import** {Location} **from "@angular/common"**; **import** {NoteActionService} **from "../../service/note-action/note-action.service"**; **import** {SaveAsService} **from "../../service/save-as/save-as.service"**; **import** {*isParagraphRunning*} **from "./paragraph/paragraph.status"**; **import** {ArrayOrderingService} **from "../../service/array-ordering/array-ordering.service"**; **import** {Car} **from "../../demo/domain/car"**; **import** {*isUndefined*} **from "util"**;  @Component({  **selector**: **'app-notebook'**,  **templateUrl**: **'./notebook.component.html'**,  **styleUrls**: [**'./notebook.component.css'**] }) **export class** NotebookComponent **implements** OnInit {   */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 更新Note名称 START \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* **titleEditor** = **false   noteName** editNoteTitle() {  **this**.**titleEditor** = **true** }   showNoteTitle() {  **this**.**titleEditor** = **false** }   */\*\* Update the note name \*/* updateNoteName(newName) {  **const** trimmedNewName = newName.trim()  **if** (trimmedNewName.**length** > 0 && **this**.**note**.**name** !== trimmedNewName) {  **this**.**note**.**name** = trimmedNewName  **this**.websocketMsgSrv.renameNote(**this**.**note**.**id**, **this**.**note**.**name**)  }  }   */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 更新Note名称 END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   //****TODO* TRASH\_FOLDER\_ID** *//当前Note的ID* **noteId** *//当前的Note实例* **note** *//是否切换编辑状态* **editorToggled** = **false** *//****TODO ?* tableToggled** = **false** *//是否只读* **viewOnly** = **false** *//是否显示配置* **showSetting** = **false** *//当前Note三种模式* **looknfeelOption** = [**'default'**, **'simple'**, **'report'**]   *//Cron的表达式* **cronOption** = [  {**name**: **'None'**, **value**: undefined},  {**name**: **'1m'**, **value**: **'0 0/1 \* \* \* ?'**},  {**name**: **'5m'**, **value**: **'0 0/5 \* \* \* ?'**},  {**name**: **'1h'**, **value**: **'0 0 0/1 \* \* ?'**},  {**name**: **'3h'**, **value**: **'0 0 0/3 \* \* ?'**},  {**name**: **'6h'**, **value**: **'0 0 0/6 \* \* ?'**},  {**name**: **'12h'**, **value**: **'0 0 0/12 \* \* ?'**},  {**name**: **'1d'**, **value**: **'0 0 0 \* \* ?'**}  ]   formatRevisionDate(date) {  **return** moment.*unix*(date).format(**'MMMM Do YYYY, h:mm a'**)  }   **paragraphUrl   asIframe   interpreterSettings** = []  **interpreterBindings** = []  **interpreterBindingsOrig** = []   **isNoteDirty** = **null  saveTimer** = **null  paragraphWarningDialog** = {}   **permissions  isOwner   allowLeave   connectedOnce** = **false** isRevisionPath(path) {  **let** pattern = **new *RegExp***(**'^.\*\/notebook\/[a-zA-Z0-9\_]\*\/revision\/[a-zA-Z0-9\_]\*'**)  **return** pattern.test(path)  }   **noteRevisions** = []  **currentRevision** = **'Head'  revisionView** = **this**.isRevisionPath(**this**.location.path())   **showPermissions   search** = {  **searchText**: **''**,  **occurrencesExists**: **false**,  **needHighlightFirst**: **false**,  **occurrencesHidden**: **false**,  **replaceText**: **''**,  **needToSendNextOccurrenceAfterReplace**: **false**,  **occurrencesCount**: 0,  **currentOccurrence**: 0,  **searchBoxOpened**: **false**,  **searchBoxWidth**: 350,  **left**: **'0px'** }  **currentSearchParagraph** = 0   **isAnonymous** getCronOptionNameFromValue(value) {  **if** (!value) {  **return ''** }   **for** (**let** o **in this**.**cronOption**) {  **if** (**this**.**cronOption**[o].**value** === value) {  **return this**.**cronOption**[o].**name** }  }  **return** value  }   blockAnonUsers() {  **let** dataSmartVersion = **this**.globalService.**dataSmartVersion  let** content = **'Only authenticated user can set the permission.'** +  **'<a data-toggle="tooltip" data-placement="top" title="Learn more" target="\_blank" href='** + url + **'>'** +  **'<i class="icon-question" />'** +  **'</a>'** BootstrapDialog.show({  **closable**: **false**,  **closeByBackdrop**: **false**,  **closeByKeyboard**: **false**,  **title**: **'No permission'**,  **message**: content,  **buttons**: [{  **label**: **'Close'**,  action: **function** (dialog) {  dialog.close()  }  }]  })  }    focusParagraphOnClick(clickEvent) {  **if** (!**this**.**note**) {  **return** }  **for** (**let** i = 0; i < **this**.**note**.**paragraphs**.**length**; i++) {  **let** paragraphId = **this**.**note**.**paragraphs**[i].**id  if** (jQuery.contains(angular.element(**'#'** + paragraphId + **'\_container'**)[0], clickEvent.**target**)) {   *// 发送片段聚焦事件，* $scope.$broadcast(**'focusParagraph'**, paragraphId, 0, **true**)  **break** }  }  }   *// register mouseevent handler for focus paragraph* **document**.addEventListener(**'click'**, **$scope**.**focusParagraphOnClick**)   keyboardShortcut(keyEvent) {  *// handle keyevent* **if** (!**this**.**viewOnly** && !**this**.**revisionView**) {  **this**.eventService.broadcast(**'keyEvent'**, keyEvent)  }  }   *// register mouseevent handler for focus paragraph* **document**.addEventListener(**'keydown'**, **this**.**keyboardShortcut**)   *//片段双击事件* paragraphOnDoubleClick(paragraphId) {  **this**.eventService.broadcast(**'doubleClickParagraph'**, paragraphId)  }   *// Move the note to trash and go back to the main page* moveNoteToTrash(noteId) {  **this**.noteActionService.moveNoteToTrash(noteId, **true**)  }   *// Remove the note permanently if it's in the trash* removeNote(noteId) {  **this**.noteActionService.*removeNote*(noteId, **true**)  }   isTrash(note) {  **return** note ? note.**name**.split(**'/'**)[0] === **this**.**TRASH\_FOLDER\_ID** : **false** }   *// Export notebook* exportNote() {  **let** jsonContent = ***JSON***.stringify(**this**.**note**)  **this**.saveAsService.saveAs(jsonContent, **this**.**note**.**name**, **'json'**)  }   *// Clone note* cloneNote(noteId) {  **let** self = **this**;  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **''**,  **message**: **'Do you want to clone this note?'**,  callback: **function** (result) {  **if** (result) {  self.websocketMsgSrv.cloneNote(noteId)  $location.path(**'/'**)  }  }  })  }   *// checkpoint/commit notebook* checkpointNote(commitMessage) {  **let** self = **this**;  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **''**,  **message**: **'Commit note to current repository?'**,  callback: **function** (result) {  **if** (result) {  self.websocketMsgSrv.checkpointNote($routeParams.**noteId**, commitMessage)  }  }  })  ***document***.getElementById(**'note.checkpoint.message'**).value = **''** }   *// set notebook head to given revision* setNoteRevision() {  **let** self = **this**;  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **''**,  **message**: **'Set notebook head to current revision?'**,  callback: **function** (result) {  **if** (result) {  websocketMsgSrv.setNoteRevision($routeParams.**noteId**, $routeParams.**revisionId**)  }  }  })  }   visitRevision(revision) {  **if** (revision.**id**) {  **if** (revision.**id** === **'Head'**) {  **this**.location.go(**'/notebook/'** + **this**.**noteId**)  } **else** {  **this**.location.go(**'/notebook/'** + **this**.**noteId** + **'/revision/'** + revision.**id**)  }  } **else** {  */\*ngToast.danger({content: 'There is a problem with this Revision',  verticalPosition: 'top',  dismissOnTimeout: false  })\*/* }  }   runAllParagraphs(noteId) {  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **''**,  **message**: **'Run all paragraphs?'**,  callback: **function** (result) {  **if** (result) {  **const** paragraphs = $scope.**note**.**paragraphs**.map(p => {  **return** {  **id**: p.**id**,  **title**: p.**title**,  **paragraph**: p.**text**,  **config**: p.**config**,  **params**: p.**settings**.**params** }  })  websocketMsgSrv.runAllParagraphs(noteId, paragraphs)  }  }  })  }   saveNote() {  **if** (**this**.**note** && **this**.**note**.**paragraphs**) {  \_.forEach(**this**.**note**.**paragraphs**, **function** (par) {  angular  .element(**'#'** + par.**id** + **'\_paragraphColumn\_main'**)  .scope()  .saveParagraph(par)  })  **this**.**isNoteDirty** = **null** }  }   clearAllParagraphOutput(noteId) {  **this**.noteActionService.*clearAllParagraphOutput*(noteId)  }   toggleAllEditor() {  **if** (**this**.**editorToggled**) {  **this**.eventService.broadcast(**'openEditor'**)  } **else** {  **this**.eventService.broadcast(**'closeEditor'**)  }  **this**.**editorToggled** = !**this**.**editorToggled** }   showAllEditor() {  **this**.eventService.broadcast(**'openEditor'**)  }   hideAllEditor() {  **this**.eventService.broadcast(**'closeEditor'**)  }   toggleAllTable() {  **if** (**this**.**tableToggled**) {  **this**.eventService.broadcast(**'openTable'**)  } **else** {  **this**.eventService.broadcast(**'closeTable'**)  }  **this**.**tableToggled** = !**this**.**tableToggled** }   showAllTable() {  **this**.eventService.broadcast(**'openTable'**)  }   hideAllTable() {  **this**.eventService.broadcast(**'closeTable'**)  }   */\*\*  \** ***@returns*** *{boolean} true if one more paragraphs are running. otherwise return false.  \*/* isNoteRunning() {  **if** (!**this**.**note**) { **return false** }   **for** (**let** i = 0; i < **this**.**note**.**paragraphs**.**length**; i++) {  **if** (*isParagraphRunning*(**this**.**note**.**paragraphs**[i])) {  **return true** }  }   **return false** }   killSaveTimer() {  **if** (**this**.**saveTimer**) {  *//$timeout.cancel(this.saveTimer)* **this**.**saveTimer** = **null** }  }   startSaveTimer() {  **this**.killSaveTimer()  **this**.**isNoteDirty** = **true** }   setLookAndFeel(looknfeel) {  **this**.**note**.**config**.**looknfeel** = looknfeel  **if** (**this**.**revisionView** === **true**) {  **this**.eventService.broadcast(**'setLookAndFeel'**, **this**.**note**.**config**.**looknfeel**)  } **else** {  **this**.setConfig()  }  }   */\*\* Set cron expression for this note \*\*/* setCronScheduler(cronExpr) {  **if** (cronExpr) {  **if** (!**this**.**note**.**config**.cronExecutingUser) {  **this**.**note**.**config**.**cronExecutingUser** = **this**.globalService.**ticket**.**principal** }  } **else** {  **this**.**note**.**config**.**cronExecutingUser** = **''** }  **this**.**note**.**config**.**cron** = cronExpr  **this**.setConfig()  }   */\*\* Set the username of the user to be used to execute all notes in notebook \*\*/* setCronExecutingUser(cronExecutingUser) {  **this**.**note**.**config**.**cronExecutingUser** = cronExecutingUser  **this**.setConfig()  }   */\*\* Set release resource for this note \*\*/* setReleaseResource(value) {  **this**.**note**.**config**.**releaseresource** = value  **this**.setConfig()  }   */\*\* Update note config \*\*/* setConfig(config) {  **if** (config) {  **this**.**note**.**config** = config  }  **this**.websocketMsgSrv.updateNote(**this**.**note**.**id**, **this**.**note**.**name**, **this**.**note**.**config**)  }     cleanParagraphExcept(paragraphId, note) {  **let** noteCopy = {}  noteCopy.id = note.**id** noteCopy.name = note.**name** noteCopy.config = note.**config** noteCopy.info = note.**info** noteCopy.paragraphs = []  **for** (**let** i = 0; i < note.**paragraphs**.**length**; i++) {  **if** (note.**paragraphs**[i].**id** === paragraphId) {  noteCopy.paragraphs[0] = note.**paragraphs**[i]  **if** (!noteCopy.paragraphs[0].**config**) {  noteCopy.paragraphs[0].**config** = {}  }  noteCopy.paragraphs[0].**config**.**editorHide** = **true** noteCopy.paragraphs[0].**config**.**tableHide** = **false  break** }  }  **return** noteCopy  }   addPara(paragraph, index) {  **this**.**note**.**paragraphs**.splice(index, 0, paragraph)  **this**.**note**.**paragraphs**.map(para => {  **if** (para.**id** === paragraph.**id**) {  para.**focus** = **true** }  })  }   removePara(paragraphId) {  **let** removeIdx  \_.each(**this**.**note**.**paragraphs**, **function** (para, idx) {  **if** (para.**id** === paragraphId) {  removeIdx = idx  }  })  **return this**.**note**.**paragraphs**.splice(removeIdx, 1)  }   **interpreterSelectionListeners** = {  accept: **function** (sourceItemHandleScope, destSortableScope) { **return true** },  itemMoved: **function** (event) {},  orderChanged: **function** (event) {}  }   openSetting() {  **this**.**showSetting** = **true  this**.getInterpreterBindings()  }   closeSetting() {  **if** (**this**.isSettingDirty()) {  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **''**,  **message**: **'Interpreter setting changes will be discarded.'**,  callback: **function** (result) {  **if** (result) {  $scope.$apply(**function** () {  $scope.**showSetting** = **false** })  }  }  })  } **else** {  **this**.**showSetting** = **false** }  }   saveSetting() {  **let** self = **this**;  **let** selectedSettingIds = []  **for** (**let** no **in this**.**interpreterBindings**) {  **let** setting = **this**.**interpreterBindings**[no]  **if** (setting.**selected**) {  selectedSettingIds.push(setting.**id**)  }  }  **this**.websocketMsgSrv.saveInterpreterBindings(**this**.**note**.**id**, selectedSettingIds)  ***console***.log(**'Interpreter bindings %o saved'**, selectedSettingIds)   **this**.**showSetting** = **false** }   *toggleSetting* = **function** () {  **if** (**this**.**showSetting**) {  **this**.closeSetting()  } **else** {  **this**.openSetting()  **this**.closePermissions()  angular.element(**'html, body'**).animate({ **scrollTop**: 0 }, **'slow'**)  }  }   openPermissions() {  **this**.**showPermissions** = **true  this**.getPermissions(***console***.log())  }   closePermissions() {  **if** (**this**.isPermissionsDirty()) {  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **''**,  **message**: **'Changes will be discarded.'**,  callback: **function** (result) {  **if** (result) {  $scope.$apply(**function** () {  $scope.**showPermissions** = **false** })  }  }  })  } **else** {  **this**.**showPermissions** = **false** }  }   *hasMatches* = **function**() {  **return this**.**search**.**occurrencesCount** > 0  }   markAllOccurrences() {  **this**.**search**.**occurrencesCount** = 0  **this**.**search**.**occurrencesHidden** = **false  this**.**currentSearchParagraph** = 0  **this**.eventService.broadcast(**'markAllOccurrences'**, **this**.**search**.**searchText**)  **this**.**search**.**currentOccurrence** = **this**.**search**.**occurrencesCount** > 0 ? 1 : 0  }   markAllOccurrencesAndHighlightFirst() {  **this**.**search**.**needHighlightFirst** = **true  this**.markAllOccurrences()  }   increaseCurrentOccurence() {  ++**this**.**search**.**currentOccurrence  if** (**this**.**search**.**currentOccurrence** > **this**.**search**.**occurrencesCount**) {  **this**.**search**.**currentOccurrence** = 1  }  }   decreaseCurrentOccurence() {  --**this**.**search**.**currentOccurrence  if** (**this**.**search**.**currentOccurrence** === 0) {  **this**.**search**.**currentOccurrence** = **this**.**search**.**occurrencesCount** }  }   sendNextOccurrenceMessage() {  **if** (**this**.**search**.**occurrencesCount** === 0) {  **this**.markAllOccurrences()  **if** (**this**.**search**.**occurrencesCount** === 0) {  **return** }  }  **if** (**this**.**search**.**occurrencesHidden**) {  **this**.markAllOccurrences()  }  **this**.eventService.broadcast(**'nextOccurrence'**, **this**.**note**.**paragraphs**[**this**.**currentSearchParagraph**].**id**)  }   sendPrevOccurrenceMessage() {  **if** (**this**.**search**.**occurrencesCount** === 0) {  **this**.markAllOccurrences()  **if** (**this**.**search**.**occurrencesCount** === 0) {  **return** }  }  **if** (**this**.**search**.**occurrencesHidden**) {  **this**.markAllOccurrences()  **this**.**currentSearchParagraph** = **this**.**note**.**paragraphs**.**length** - 1  }  **this**.eventService.broadcast(**'prevOccurrence'**, **this**.**note**.**paragraphs**[**this**.**currentSearchParagraph**].**id**)  }   increaseCurrentSearchParagraph() {  ++**this**.**currentSearchParagraph  if** (**this**.**currentSearchParagraph** >= **this**.**note**.**paragraphs**.**length**) {  **this**.**currentSearchParagraph** = 0  }  }   decreaseCurrentSearchParagraph() {  --**this**.**currentSearchParagraph  if** (**this**.**currentSearchParagraph** === -1) {  **this**.**currentSearchParagraph** = **this**.**note**.**paragraphs**.**length** - 1  }  }   nextOccurrence() {  **this**.sendNextOccurrenceMessage()  **this**.increaseCurrentOccurence()  }   prevOccurrence() {  **this**.sendPrevOccurrenceMessage()  **this**.decreaseCurrentOccurence()  }   replace() {  **if** (**this**.**search**.**occurrencesCount** === 0) {  **this**.markAllOccurrencesAndHighlightFirst()  **if** (**this**.**search**.**occurrencesCount** === 0) {  **return** }  }  **if** (**this**.**search**.**occurrencesHidden**) {  **this**.markAllOccurrencesAndHighlightFirst()  **return** }  **this**.eventService.broadcast(**'replaceCurrent'**, **this**.**search**.**searchText**, **this**.**search**.**replaceText**)  **if** (**this**.**search**.**needToSendNextOccurrenceAfterReplace**) {  **this**.sendNextOccurrenceMessage()  **this**.**search**.**needToSendNextOccurrenceAfterReplace** = **false** }  }   replaceAll() {  **if** (**this**.**search**.**occurrencesCount** === 0) {  **return** }  **if** (**this**.**search**.**occurrencesHidden**) {  **this**.markAllOccurrencesAndHighlightFirst()  }  **this**.eventService.broadcast(**'replaceAll'**, **this**.**search**.**searchText**, **this**.**search**.**replaceText**)  **this**.markAllOccurrencesAndHighlightFirst()  }   onPressOnFindInput(event) {  **if** (event.**keyCode** === 13) {  **this**.nextOccurrence()  }  }   makeSearchBoxVisible() {  **if** (**this**.**search**.**searchBoxOpened**) {  **this**.**search**.**searchBoxOpened** = **false  *console***.log(**'make 0'**)  **this**.**search**.**left** = **'0px'** } **else** {  **this**.**search**.**searchBoxOpened** = **true** *//let searchGroupRect = angular.element('#searchGroup')[0].getBoundingClientRect()* ***console***.log(**'make visible'**)  *//let dropdownRight = searchGroupRect.left + this.search.searchBoxWidth* ***console***.log(dropdownRight + **' '** + ***window***.**innerWidth**)  **if** (dropdownRight + 5 > ***window***.**innerWidth**) {  **this**.**search**.**left** = ***window***.**innerWidth** - dropdownRight - 15 + **'px'** }  }  }   searchClicked() {  **this**.makeSearchBoxVisible()  }   restartInterpreter(interpreter) {  **const** thisConfirm = BootstrapDialog.confirm({  **closable**: **false**,  **closeByBackdrop**: **false**,  **closeByKeyboard**: **false**,  **title**: **''**,  **message**: **'Do you want to restart '** + interpreter.**name** + **' interpreter?'**,  callback: **function**(result) {  **if** (result) {  **let** payload = {  **'noteId'**: $scope.**note**.**id** }   thisConfirm.$modalFooter.find(**'button'**).addClass(**'disabled'**)  thisConfirm.$modalFooter.find(**'button:contains("OK")'**)  .html(**'<i class="fa fa-circle-o-notch fa-spin"></i> Saving Setting'**)   $http.put(baseUrlSrv.getRestApiBase() + **'/interpreter/setting/restart/'** + interpreter.**id**, payload)  .success(**function**(data, status, headers, config) {  **let** index = \_.findIndex($scope.**interpreterSettings**, {**'id'**: interpreter.**id**})  $scope.**interpreterSettings**[index] = data.**body** thisConfirm.close()  }).error(**function** (data, status, headers, config) {  thisConfirm.close()  ***console***.log(**'Error %o %o'**, status, data.**message**)  BootstrapDialog.show({  **title**: **'Error restart interpreter.'**,  **message**: data.**message** })  })  **return false** }  }  })  }   savePermissions() {  **if** (**this**.**isAnonymous** || **this**.globalService.**ticket**.**principal**.trim().**length** === 0) {  **this**.blockAnonUsers()  }  **this**.convertPermissionsToArray()  **if** (**this**.isOwnerEmpty()) {  BootstrapDialog.show({  **closable**: **false**,  **title**: **'Setting Owners Permissions'**,  **message**: **'Please fill the [Owners] field. If not, it will set as current user.\n\n'** +  **'Current user : [ '** + $rootScope.**ticket**.**principal** + **']'**,  **buttons**: [  {  **label**: **'Set'**,  action: **function**(dialog) {  dialog.close()  $scope.**permissions**.**owners** = [$rootScope.**ticket**.**principal**]  $scope.setPermissions()  }  },  {  **label**: **'Cancel'**,  action: **function**(dialog) {  dialog.close()  $scope.openPermissions()  }  }  ]  })  } **else** {  **this**.setPermissions()  }  }   togglePermissions() {  **let** principal = **this**.globalService.**ticket**.**principal  this**.**isAnonymous** = principal === **'anonymous'** ? **true** : **false  if** (!!principal && **this**.**isAnonymous**) {  **this**.blockAnonUsers()  } **else** {  **if** (**this**.**showPermissions**) {  **this**.closePermissions()  angular.element(**'#selectOwners'**).select2({})  angular.element(**'#selectReaders'**).select2({})  angular.element(**'#selectRunners'**).select2({})  angular.element(**'#selectWriters'**).select2({})  } **else** {  **this**.openPermissions()  **this**.closeSetting()  }  }  }   toggleNotePersonalizedMode() {  **let** personalizedMode = **this**.**note**.**config**.personalizedMode  **if** (**this**.**isOwner**) {  BootstrapDialog.confirm({  **closable**: **true**,  **title**: **'Setting the result display'**,  message: **function** (dialog) {  **let** modeText = $scope.**note**.**config**.personalizedMode === **'true'** ? **'collaborate'** : **'personalize'  return 'Do you want to <span class="text-info">'** + modeText + **'</span> your analysis?'** },  callback: **function** (result) {  **if** (result) {  **if** ($scope.**note**.**config**.personalizedMode === undefined) {  $scope.**note**.**config**.**personalizedMode** = **'false'** }  $scope.**note**.**config**.**personalizedMode** = personalizedMode === **'true'** ? **'false'** : **'true'** websocketMsgSrv.updatePersonalizedMode($scope.**note**.**id**, $scope.**note**.**config**.personalizedMode)  }  }  })  }  }   isSettingDirty() {  **if** (angular.equals($scope.**interpreterBindings**, $scope.**interpreterBindingsOrig**)) {  **return false** } **else** {  **return true** }  }   isPermissionsDirty() {  **if** (angular.equals($scope.**permissions**, $scope.permissionsOrig)) {  **return false** } **else** {  **return true** }  }    isOwnerEmpty() {  **if** (**this**.**permissions**.owners.**length** > 0) {  **for** (**let** i = 0; i < **this**.**permissions**.owners.**length**; i++) {  **if** (**this**.**permissions**.owners[i].trim().**length** > 0) {  **return false** } **else if** (i === **this**.**permissions**.owners.**length** - 1) {  **return true** }  }  } **else** {  **return true** }  }   **constructor**(**private** eventService:EventService1,  **private** websocketMsgSrv:WebsocketMessageService,  **private** route:ActivatedRoute,  **private** noteVarShareService:NoteVarShareService,  **private** router:Router,  **private** httpClient:HttpClient,  **private** baseUrlSrv:BaseUrlService,  **private** globalService:GlobalService,  **private** location:Location,  **private** noteActionService:NoteActionService,  **private** saveAsService:SaveAsService,  **private** arrayOrderingSrv:ArrayOrderingService) {   **this**.**note** = {}  }   **orderListCars**: Car[];   getNoteName(note) {  **if** (note) {  **return this**.arrayOrderingSrv.getNoteName(note)  }  }   ngOnInit(): **void** {   **this**.**orderListCars** = [  {**vin**: **'r3278r2'**, **year**: 2010, **brand**: **'Audi'**, **color**: **'Black'**},  {**vin**: **'jhto2g2'**, **year**: 2015, **brand**: **'BMW'**, **color**: **'White'**},  {**vin**: **'h453w54'**, **year**: 2012, **brand**: **'Honda'**, **color**: **'Blue'**},  {**vin**: **'g43gwwg'**, **year**: 1998, **brand**: **'Renault'**, **color**: **'White'**},  {**vin**: **'gf45wg5'**, **year**: 2011, **brand**: **'VW'**, **color**: **'Red'**},  {**vin**: **'bhv5y5w'**, **year**: 2015, **brand**: **'Jaguar'**, **color**: **'Blue'**},  {**vin**: **'ybw5fsd'**, **year**: 2012, **brand**: **'Ford'**, **color**: **'Yellow'**},  {**vin**: **'45665e5'**, **year**: 2011, **brand**: **'Mercedes'**, **color**: **'Brown'**},  {**vin**: **'he6sb5v'**, **year**: 2015, **brand**: **'Ford'**, **color**: **'Black'**}  ];   **let** self = **this**;  **this**.route.**params**.subscribe((params: Params) => {  **this**.**noteId** = params[**'noteId'**]   *//设置Note内容获取到的回调* **this**.eventService.subscribe(**'setNoteContent'**, **function** (note) {  self.initNoteContent(note)  })   *//设置Bingding信息获取到的回调* **this**.eventService.subscribe(**'interpreterBindings'**, **function** (data) {  self.callbackInterpreterBindings(data)  })   *// 监听Websocket的连接状态* **this**.eventService.subscribe(**'setConnectedStatus'**, **function** (msg) {  **if** (self.**connectedOnce** && msg) {  self.initNotebook()  }  self.**connectedOnce** = **true** })   **this**.eventService.subscribe(**'listRevisionHistory'**, **function** (data) {  ***console***.debug(**'received list of revisions %o'**, data)  **this**.**noteRevisions** = data.revisionList  **this**.**noteRevisions**.splice(0, 0, {  **id**: **'Head'**,  **message**: **'Head'** })  **if** ($routeParams.**revisionId**) {  **let** index = \_.findIndex(**this**.**noteRevisions**, {**'id'**: $routeParams.**revisionId**})  **if** (index > -1) {  **this**.**currentRevision** = **this**.**noteRevisions**[index].**message** }  }  })   **this**.eventService.subscribe(**'noteRevision'**, **function** (data) {  ***console***.log(**'received note revision %o'**, data)  **if** (data.**note**) {  **this**.**note** = data.**note  this**.initializeLookAndFeel()  } **else** {  self.location.go(**'/'**)  }  })   **this**.eventService.subscribe(**'setNoteRevisionResult'**, **function** (data) {  ***console***.log(**'received set note revision result %o'**, data)  **if** (data.**status**) {  self.location.go(**'/notebook/'** + self.**noteId**)  }  })   **this**.eventService.subscribe(**'addParagraph'**, **function** (event, paragraph, index) {  **if** (self.**paragraphUrl** || self.**revisionView** === **true**) {  **return** }  self.addPara(paragraph, index)  })   **this**.eventService.subscribe(**'removeParagraph'**, **function** (event, paragraphId) {  **if** (self.**paragraphUrl** || self.**revisionView** === **true**) {  **return** }  self.removePara(paragraphId)  })   **this**.eventService.subscribe(**'moveParagraph'**, **function** (event, paragraphId, newIdx) {  **if** (self.**revisionView** === **true**) {  **return** }  **let** removedPara = self.removePara(paragraphId)  **if** (removedPara && removedPara.**length** === 1) {  self.addPara(removedPara[0], newIdx)  }  })   **this**.eventService.subscribe(**'updateNote'**, **function** (event, name, config, info) {  */\*\* update Note name \*/* **if** (name !== self.**note**.**name**) {  ***console***.log(**'change note name to : %o'**, self.**note**.**name**)  self.**note**.**name** = name  }  self.**note**.**config** = config  self.**note**.**info** = info  self.initializeLookAndFeel()  })   **this**.eventService.subscribe(**'occurrencesExists'**, **function**(event, count) {  self.**search**.**occurrencesCount** += count  **if** (self.**search**.**needHighlightFirst**) {  self.sendNextOccurrenceMessage()  self.**search**.**needHighlightFirst** = **false** }  })   **this**.eventService.subscribe(**'noNextOccurrence'**, **function**(event) {  **this**.increaseCurrentSearchParagraph()  **this**.sendNextOccurrenceMessage()  })   **this**.eventService.subscribe(**'noPrevOccurrence'**, **function**(event) {  self.decreaseCurrentSearchParagraph()  self.sendPrevOccurrenceMessage()  })   **this**.eventService.subscribe(**'editorClicked'**, **function**() {  self.**search**.**occurrencesHidden** = **true** self.eventService.broadcast(**'unmarkAll'**)  })   **this**.eventService.subscribe(**'occurrencesCountChanged'**, **function**(event, cnt) {  self.**search**.**occurrencesCount** += cnt  **if** (self.**search**.**occurrencesCount** === 0) {  self.**search**.**currentOccurrence** = 0  } **else** {  self.**search**.**currentOccurrence** += cnt + 1  **if** (self.**search**.**currentOccurrence** > self.**search**.**occurrencesCount**) {  self.**search**.**currentOccurrence** = 1  }  }  })   **this**.eventService.subscribe(**'noNextOccurrenceAfterReplace'**, **function**() {  self.**search**.**occurrencesCount** = 0  self.**search**.**needHighlightFirst** = **false** self.**search**.**needToSendNextOccurrenceAfterReplace** = **false** self.eventService.broadcast(**'checkOccurrences'**)  self.increaseCurrentSearchParagraph()  **if** (self.**search**.**occurrencesCount** > 0) {  self.**search**.**needToSendNextOccurrenceAfterReplace** = **true** }  })   **this**.eventService.subscribe(**'toggleSearchBox'**, **function**() {  */\*let elem = angular.element('#searchGroup')  if (self.search.searchBoxOpened) {  elem.removeClass('open')  } else {  elem.addClass('open')  }  $timeout(self.makeSearchBoxVisible())\*/* })   **this**.eventService.subscribe(**'moveParagraphUp'**, **function** (paragraph) {  **let** newIndex = -1  **for** (**let** i = 0; i < self.**note**.**paragraphs**.**length**; i++) {  **if** (self.**note**.**paragraphs**[i].**id** === paragraph.**id**) {  newIndex = i - 1  **break** }  }  **if** (newIndex < 0 || newIndex >= self.**note**.**paragraphs**.**length**) {  **return** }  *// save dirtyText of moving paragraphs.* **let** prevParagraph = self.**note**.**paragraphs**[newIndex]  self.websocketMsgSrv.moveParagraph(paragraph.**id**, newIndex)  })   **this**.eventService.subscribe(**'moveParagraphDown'**, **function** (event, paragraph) {  **let** newIndex = -1  **for** (**let** i = 0; i < self.**note**.**paragraphs**.**length**; i++) {  **if** (self.**note**.**paragraphs**[i].**id** === paragraph.**id**) {  newIndex = i + 1  **break** }  }   **if** (newIndex < 0 || newIndex >= self.**note**.**paragraphs**.**length**) {  **return** }  *// save dirtyText of moving paragraphs.* **let** nextParagraph = self.**note**.**paragraphs**[newIndex]  angular  .element(**'#'** + paragraph.**id** + **'\_paragraphColumn\_main'**)  .scope()  .saveParagraph(paragraph)  angular  .element(**'#'** + nextParagraph.**id** + **'\_paragraphColumn\_main'**)  .scope()  .saveParagraph(nextParagraph)  self.websocketMsgSrv.moveParagraph(paragraph.**id**, newIndex)  })   **this**.eventService.subscribe(**'moveFocusToPreviousParagraph'**, **function** (event, currentParagraphId) {  **let** focus = **false  for** (**let** i = self.**note**.**paragraphs**.**length** - 1; i >= 0; i--) {  **if** (focus === **false**) {  **if** (self.**note**.**paragraphs**[i].**id** === currentParagraphId) {  focus = **true  continue** }  } **else** {  self.eventService.broadcast(**'focusParagraph'**, self.**note**.**paragraphs**[i].**id**, -1)  **break** }  }  })   **this**.eventService.subscribe(**'moveFocusToNextParagraph'**, **function** (event, currentParagraphId) {  **let** focus = **false  for** (**let** i = 0; i < self.**note**.**paragraphs**.**length**; i++) {  **if** (focus === **false**) {  **if** (self.**note**.**paragraphs**[i].**id** === currentParagraphId) {  focus = **true  continue** }  } **else** {  self.eventService.broadcast(**'focusParagraph'**, self.**note**.**paragraphs**[i].**id**, 0)  **break** }  }  })   **this**.eventService.subscribe(**'insertParagraph'**, **function** (event, paragraphId, position) {  **if** (self.**revisionView** === **true**) {  **return** }  **let** newIndex = -1  **for** (**let** i = 0; i < self.**note**.**paragraphs**.**length**; i++) {  **if** (self.**note**.**paragraphs**[i].**id** === paragraphId) {  *// determine position of where to add new paragraph; default is below* **if** (position === **'above'**) {  newIndex = i  } **else** {  newIndex = i + 1  }  **break** }  }   **if** (newIndex < 0 || newIndex > self.**note**.**paragraphs**.**length**) {  **return** }  self.websocketMsgSrv.insertParagraph(newIndex)  })   **this**.eventService.subscribe(**'$routeChangeStart'**, **function** (event, next, current) {  **if** (!self.**note** || !self.**note**.**paragraphs**) {  **return** }  **if** (self.**note** && self.**note**.**paragraphs**) {  self.**note**.**paragraphs**.map(par => {  **if** (self.**allowLeave** === **true**) {  **return** }  })  }  })   **this**.eventService.subscribe(**'$destroy'**, **function** () {  *//angular.element(window).off('beforeunload')* self.killSaveTimer()  self.saveNote()   ***document***.removeEventListener(**'click'**, self.**focusParagraphOnClick**)  ***document***.removeEventListener(**'keydown'**, self.**keyboardShortcut**)  })   **this**.eventService.subscribe(**'$unBindKeyEvent'**, **function** () {  ***document***.removeEventListener(**'click'**, self.**focusParagraphOnClick**)  ***document***.removeEventListener(**'keydown'**, self.**keyboardShortcut**)  })   **this**.initNotebook()  })  }   */\*\* 初始化笔记 \*/* initNotebook():**void** {  **this**.noteVarShareService.*clear*()  **if** ($routeParams.**revisionId**) {  websocketMsgSrv.getNoteByRevision($routeParams.**noteId**, $routeParams.**revisionId**)  } **else** {  websocketMsgSrv.getNote($routeParams.**noteId**)  }  **this**.websocketMsgSrv.getNote(**this**.**noteId**)   **this**.websocketMsgSrv.listRevisionHistory($routeParams.**noteId**)  **let** currentRoute = $route.**current  if** (currentRoute) {  *setTimeout*(  **function** () {  **let** routeParams = currentRoute.**params  let** $id = angular.element(**'#'** + routeParams.**paragraph** + **'\_container'**)   **if** ($id.**length** > 0) {  *// adjust for navbar* **let** top = $id.offset().**top** - 103  angular.element(**'html, body'**).scrollTo({**top**: top, **left**: 0})  }  },  1000  )  }  }   */\*\* 初始化Note \*/* initNoteContent(note):**void** {  **if** (note === undefined) {  **this**.router.navigate([**'/login'**]);  }   **this**.**note** = note[0]  **this**.**noteName** = **this**.getNoteName(**this**.**note**)  */\*$scope.paragraphUrl = $routeParams.paragraphId  $scope.asIframe = $routeParams.asIframe\*/* **if** (**this**.**paragraphUrl**) {  $scope.**note** = cleanParagraphExcept($scope.**paragraphUrl**, $scope.**note**)  $scope.$broadcast(**'$unBindKeyEvent'**, $scope.$unBindKeyEvent)  $rootScope.$broadcast(**'setIframe'**, $scope.**asIframe**)  initializeLookAndFeel()  **return** }   **this**.initializeLookAndFeel()   *// open interpreter binding setting when there're none selected* **this**.getInterpreterBindings()  **this**.getPermissions(***console***.log())  **let** isPersonalized = **this**.**note**.**config**.personalizedMode  isPersonalized = isPersonalized === undefined ? **'false'** : isPersonalized  **this**.**note**.**config**.**personalizedMode** = isPersonalized  }   *//初始化外观* initializeLookAndFeel():**void** {  **if** (!**this**.**note**.**config**.**looknfeel**) {  **this**.**note**.**config**.**looknfeel** = **'default'** } **else** {  **this**.**viewOnly** = **this**.**note**.**config**.**looknfeel** === **'report'** ? **true** : **false** }   **if** (**this**.**note**.**paragraphs** && **this**.**note**.**paragraphs**[0]) {  **this**.**note**.**paragraphs**[0].**focus** = **true** }  **this**.eventService.broadcast(**'setLookAndFeel'**, **this**.**note**.**config**.**looknfeel**)  }   *//获取binding信息* getInterpreterBindings():**void** {  **this**.websocketMsgSrv.getInterpreterBindings(**this**.**note**.**id**)  }   *//监听Bing信息的回调* callbackInterpreterBindings(data):**void** {  **this**.**interpreterBindings** = data[0].**interpreterBindings** *//this.interpreterBindingsOrig = angular.copy(this.interpreterBindings) // to check dirty* **let** selected = **false  let** key  **let** setting   **for** (key **in this**.**interpreterBindings**) {  setting = **this**.**interpreterBindings**[key]  **if** (setting.**selected**) {  selected = **true  break** }  }   **if** (!selected) {  *// make default selection* **let** selectedIntp = {}  **for** (key **in this**.**interpreterBindings**) {  setting = **this**.**interpreterBindings**[key]  **if** (!selectedIntp[setting.**name**]) {  setting.**selected** = **true** selectedIntp[setting.**name**] = **true** }  }  **this**.**showSetting** = **true** }   }   *// 获取Note的权限* getPermissions(callback):**void** {  **let** self = **this**;  **this**.httpClient.get(**this**.baseUrlSrv.getRestApiBase() + **'/notebook/'**+ **this**.**note**.**id** + **'/permissions'**)  .subscribe(  response => {  self.**permissions** = response[**'body'**]  *//self.permissionsOrig = angular.copy(self.permissions) // to check dirty* **let** selectJson = {  **tokenSeparators**: [**','**, **' '**],  **ajax**: {  url: **function** (params) {  **if** (!params.term) {  **return false** }  **return** self.baseUrlSrv.getRestApiBase() + **'/security/userlist/'** + params.term  },  **delay**: 250,  processResults: **function** (data, params) {  **let** results = []   **if** (data.**body**.users.**length** !== 0) {  **let** users = []  **for** (**let** len = 0; len < data.**body**.users.**length**; len++) {  users.push({  **'id'**: data.**body**.users[len],  **'text'**: data.**body**.users[len]  })  }  results.push({  **'text'**: **'Users :'**,  **'children'**: users  })  }  **if** (data.**body**.**roles**.**length** !== 0) {  **let** roles = []  **for** (**let** len = 0; len < data.**body**.**roles**.**length**; len++) {  roles.push({  **'id'**: data.**body**.**roles**[len],  **'text'**: data.**body**.**roles**[len]  })  }  results.push({  **'text'**: **'Roles :'**,  **'children'**: roles  })  }  **return** {  **results**: results,  **pagination**: {  **more**: **false** }  }  },  **cache**: **false** },  **width**: **' '**,  **tags**: **true**,  **minimumInputLength**: 3  }   self.setIamOwner()  **if** (callback) {  callback()  }  },  errorResponse => {  **let** redirect = errorResponse.headers(**'Location'**)  **if** (errorResponse.**status** === 401 && redirect !== undefined) {  *// Handle page redirect* ***window***.**location**.**href** = redirect  }  }  );  }   setIamOwner():**boolean** {  **this**.**isOwner** = **true  return true** }   columnWidthClass(n) {  **if** (**this**.**asIframe** || n == undefined) {  **return 'ui-g-12'** } **else** {  **return 'paragraph-col ui-g-'** + n  }  }   */\*\*  \* 从note中根据id获取相应的片段  \** ***@param*** *{string} id  \*/* getParagraphById(id:**string**){   **for** (**let** paragraph **of this**.**note**.**paragraphs**) {  **if**(paragraph.**id** == id){  **return** paragraph  }  }   **return null** }  } |