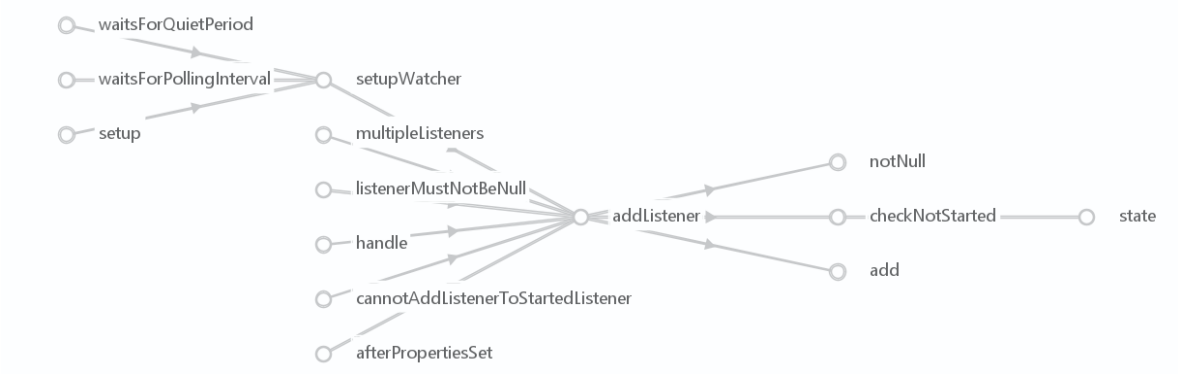
Spring-boot

File-watcher

`FileSystemWatcher.java` is able to watch the file changes in the specific folder. Spring-boot can make it easier to build a whole micro service, so monitoring the file systems is a key essential feature.

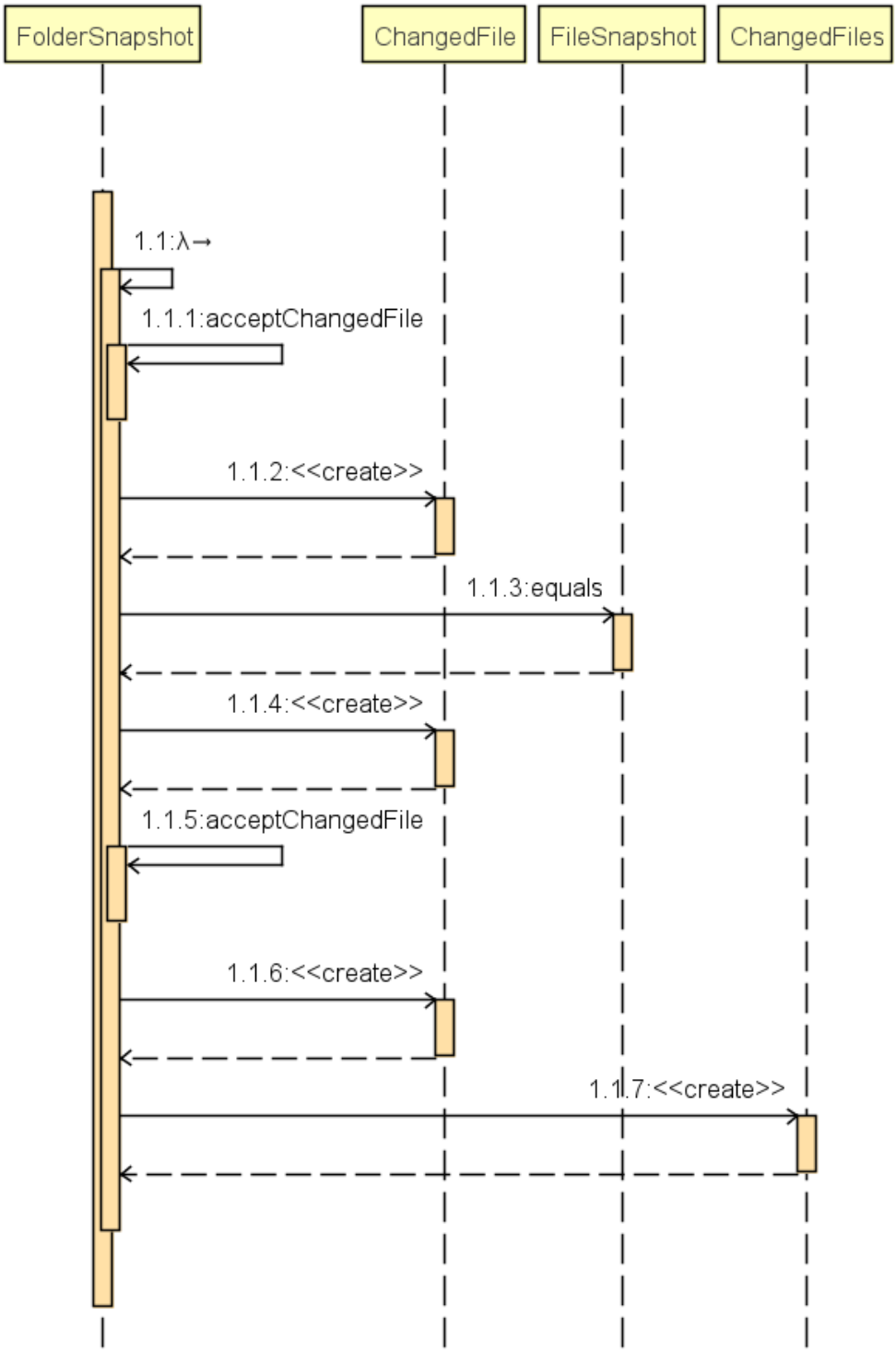
`addListener()` is the start of our file system. It is called in the `setupWatcher.java` .




```
public void addListener(FileChangeListener fileChangeListener) {
        Assert.notNull(fileChangeListener, "FileChangeListener must not be
null");
        synchronized (this.monitor) {
            checkNotStarted();
            this.listeners.add(fileChangeListener);
        }
    }
```

`addSourceFolder()` is for file directory. The `folder` value is a map and it maps its folder directory and its `FolderSnapshot`. `FolderSnapshot.java` is a class that tracks every files in that folder. Firstly, `collectFiles()` is a function to collect every files in this folder and add every files in the child folder. Secondly, it supports the operation to get all changed files. It keeps a record of the old version files, and compare each item in the latest file systems. If there is a change is the file, it was added in the set called `changes`. To compare files, it overrides the `hashCode()` and compare with hashcode.

```
@Override
public int hashCode() {
        int hashCode = this.folder.hashCode();
        hashCode = 31 * hashCode + this.files.hashCode();
        return hashCode;
    }
```



The file information is saved in `FileSnapshot.java`, it holds the current file name, if the file exists, file length, and the last modified time of this file.

When the thread is started, the flow of the functions is as follows.

1. Check `remainingScans` and decide whether to run `scan()`.

```
int remainingScans = this.remainingScans.get();
            while (remainingScans > 0 || remainingScans == -1) {
                try {
                    if (remainingScans > 0) {
                        this.remainingScans.decrementAndGet();
                    }
                    scan();
                }
                catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                }
                remainingScans = this.remainingScans.get();
            }
```

2. In the `scan()` method, it calls the previous file map aforementioned and the current file map. And it calls `isDifferent()` method, which returns a boolean to flag if there is a different. If there actually is a difference, it will call `updateSnapshots()` to update the file map.

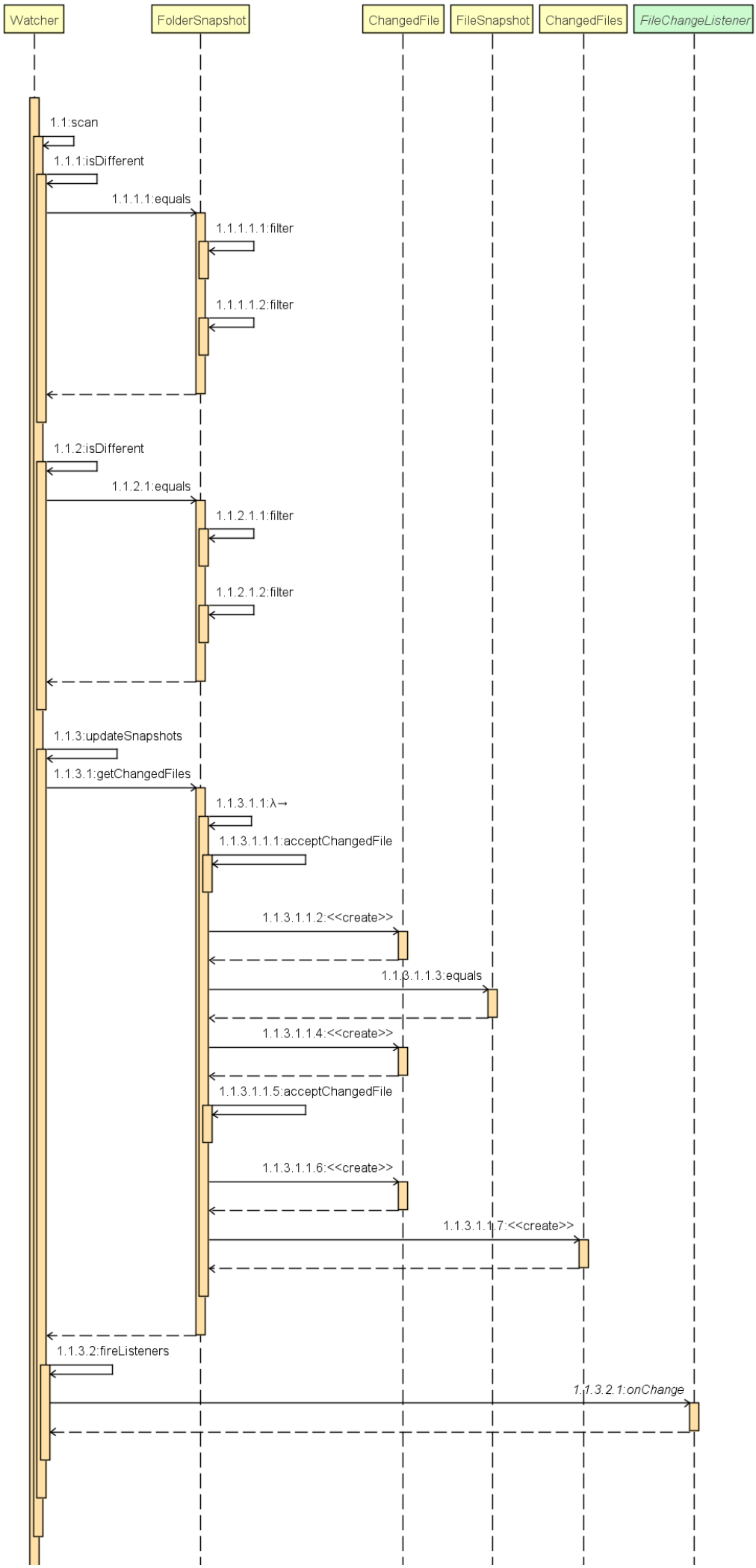
```
private void scan() throws InterruptedException {
            Thread.sleep(this.pollInterval - this.quietPeriod);
            Map<File, FolderSnapshot> previous;
            Map<File, FolderSnapshot> current = this.folders;
            do {
                previous = current;
                current = getCurrentSnapshots();
                Thread.sleep(this.quietPeriod);
            }
            while (isDifferent(previous, current));
            if (isDifferent(this.folders, current)) {
                updateSnapshots(current.values());
            }
        }
```

3. In the `isDifferent()` method, it firstly compares the key(file) between each map. And if they has the same keys, it will compare the values.

```
private boolean isDifferent(Map<File, FolderSnapshot> previous,
Map<File, FolderSnapshot> current) {
            if (!previous.keySet().equals(current.keySet())) {
                return true;
            }
            for (Map.Entry<File, FolderSnapshot> entry :
previous.entrySet()) {
                FolderSnapshot previousFolder = entry.getValue();
                FolderSnapshot currentFolder =
current.get(entry.getKey());
                if (!previousFolder.equals(currentFolder,
this.triggerFilter)) {
                    return true;
                }
            }
            return false;
        }
```

4. In the `updateSnapshots()` function, it takes the current snapshot as the input, and it stores the changed file in the `changeSet`.

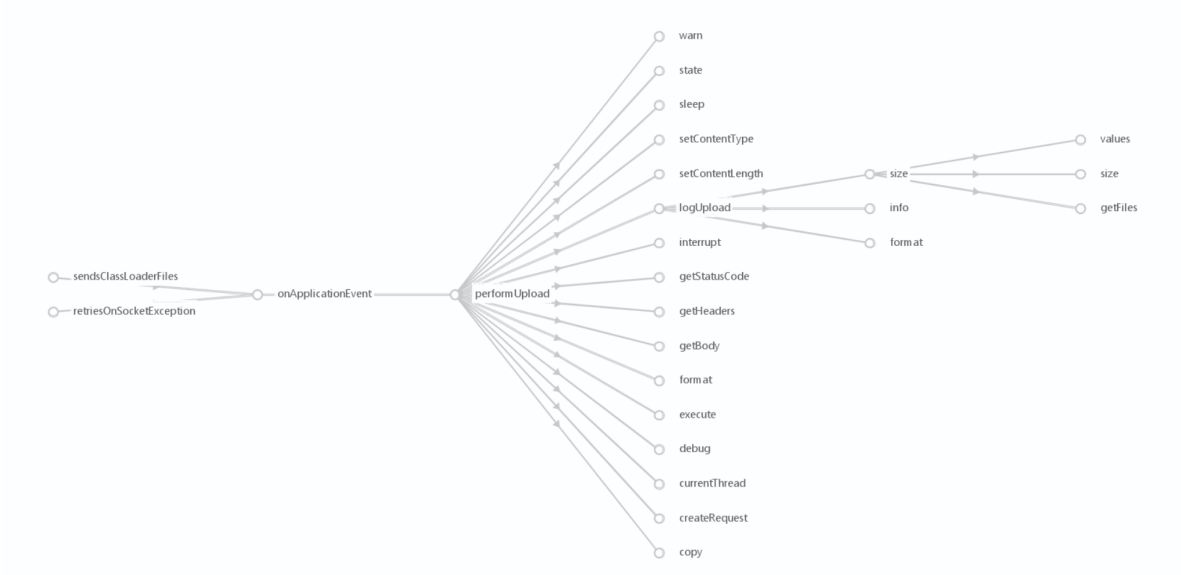
```
private void updateSnapshots(Collection<FolderSnapshot> snapshots) {
            Map<File, FolderSnapshot> updated = new LinkedHashMap<>();
            Set<ChangedFiles> changeSet = new LinkedHashSet<>();
            for (FolderSnapshot snapshot : snapshots) {
                FolderSnapshot previous =
this.folders.get(snapshot.getFolder());
                updated.put(snapshot.getFolder(), snapshot);
                ChangedFiles changedFiles =
previous.getChangedFiles(snapshot, this.triggerFilter);
                if (!changedFiles.getFiles().isEmpty()) {
                    changeSet.add(changedFiles);
                }
            }
            if (!changeSet.isEmpty()) {
                fireListeners(Collections.unmodifiableSet(changeSet));
            }
            this.folders = updated;
        }
```


Watcher
FolderSnapshot
ChangedFile
FileSnapshot
ChangedFiles
FileChangeListener
1.1:scan
1.1.1:isDifferent
1.1.1.1:equals
1.1.1.1.1:filter
1.1.1.1.2:filter
1.1.2:isDifferent
1.1.2.1:equals
1.1.2.1.1:filter
1.1.2.1.2:filter
1.1.3:updateSnapshots
1.1.3.1:getChangedFiles
1.1.3.1.1:λ→
1.1.3.1.1.1:acceptChangedFile
1.1.3.1.1.2:<<create>>
1.1.3.1.1.3:equals
1.1.3.1.1.4:<<create>>
1.1.3.1.1.5:acceptChangedFile
1.1.3.1.1.6:<<create>>
1.1.3.1.1.7:<<create>>
1.1.3.2:fireListeners
1.1.3.2.1:onChange

Client - Server

Client

The communication between client and server from spring-boot is another important feature. Let's first take a look at the client end.



The graph above shows the flow of the client.

The `ClassPathChangeUploader.java` listens and pushes all the classpath updates to the remote endpoint. Its constructor has two parameters, `String url` and `ClientHttpRequestFactory requestFactory`. `ClientHttpRequestFactory` is an interface that create a new Client HTTP request by this line. And the constructer also check the correctness of url.

```
1 ClientHttpRequest createRequest(URI uri, HttpMethod httpMethod) throws
  IOException;
```

When client starts, it will call `onApplicationEvent()` This method takes `ClassPathChangedEvent.java` as the parameter and this class contains all the changes related to the classpath change. With this parameter, client could gather all the source files by passing it to a `ClassLoaderFiles`. It implements `ClassLoaderFileRepository` and `Serializable` and it could iterate all the folders to fetch all the files in that directory. In fact, in the `onApplicationEvent()`, `getClassLoaderFiles()` is called to return a `ClassLoaderFiles` with all the changed files. Then the files are serialized into a byte array and later pushed to the remote endpoint.

```
private ClassLoaderFiles getClassLoaderFiles(ClassPathChangedEvent event)
throws IOException {
        ClassLoaderFiles files = new ClassLoaderFiles();
        for (ChangedFiles changedFiles : event.getChangeSet()) {
            String sourceFolder =
changedFiles.getSourceFolder().getAbsolutePath();
            for (ChangedFile changedFile : changedFiles) {
                files.addFile(sourceFolder, changedFile.getRelativeName(),
asClassLoaderFile(changedFile));
            }
        }
        return files;
    }
```

To push the changes to the remote endpoint, we use `performUpload()` . This function takes the aforementioned `classLoaderFiles` as the parameter. Firstly, it utilized `ClientHttpRequest` to build the correct http request format. Secondly, it calls `ClientHttpResponse` to check if the communication is established. If the status is `OK` , then we will pass it to the `logUpload()` function.

```
private void performUpload(ClassLoaderFiles classLoaderFiles, byte[] bytes)
throws IOException {
        try {
            while (true) {
                try {
                    ClientHttpRequest request =
this.requestFactory.createRequest(this.uri, HttpMethod.POST);
                    HttpHeaders headers = request.getHeaders();

headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
                    headers.setContentLength(bytes.length);
                    FileCopyUtils.copy(bytes, request.getBody());
                    ClientHttpResponse response = request.execute();
                    HttpStatus statusCode = response.getStatusCode();
                    Assert.state(statusCode == HttpStatus.OK,
                            () -> "Unexpected " + statusCode + " response
uploading class files");
                    logUpload(classLoaderFiles);
                    return;
                }
                catch (SocketException ex) {
                    logger.warn(LogMessage.format(
                            "A failure occurred when uploading to %s.
Upload will be retried in 2 seconds", this.uri));
                    logger.debug("Upload failure", ex);
                    Thread.sleep(2000);
                }
            }
        }
        catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
            throw new IllegalStateException(ex);
        }
    }
```

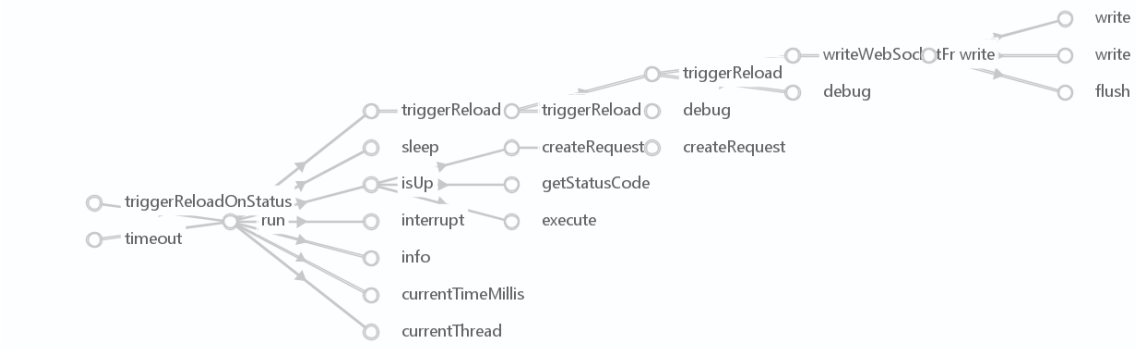
`DelayedLiveReloadTrigger` is a class that waits to triggers live reload until the remote server has restarted. Its run method sleeps until the remote sever is up. And if the server continues to sleep, when the sleep time reaches a limit, it will stop waiting.

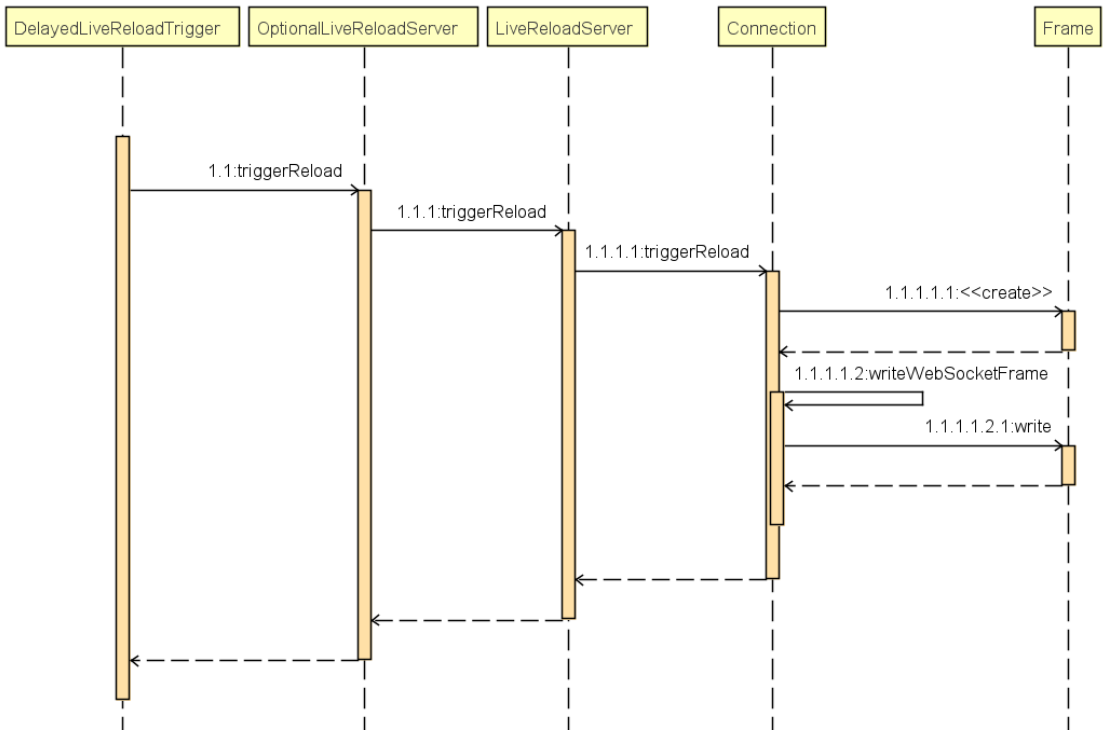
```
public void run() {
        try {
            Thread.sleep(this.shutdownTime);
            long start = System.currentTimeMillis();
            while (!isUp()) {
                long runTime = System.currentTimeMillis() - start;
                if (runTime > this.timeout) {
                    return;
                }
                Thread.sleep(this.sleepTime);
            }
            logger.info("Remote server has changed, triggering
LiveReload");
            this.liveReloadServer.triggerReload();
        }
        catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
```

To determine if the server is up, we just create a random request and see if we get a proper response, which is a `response.getStatusCode() == HttpStatus.OK`.

```
private boolean isUp() {
        try {
            ClientHttpRequest request = createRequest();
            ClientHttpResponse response = request.execute();
            return response.getStatusCode() == HttpStatus.OK;
        }
        catch (Exception ex) {
            return false;
        }
    }
```

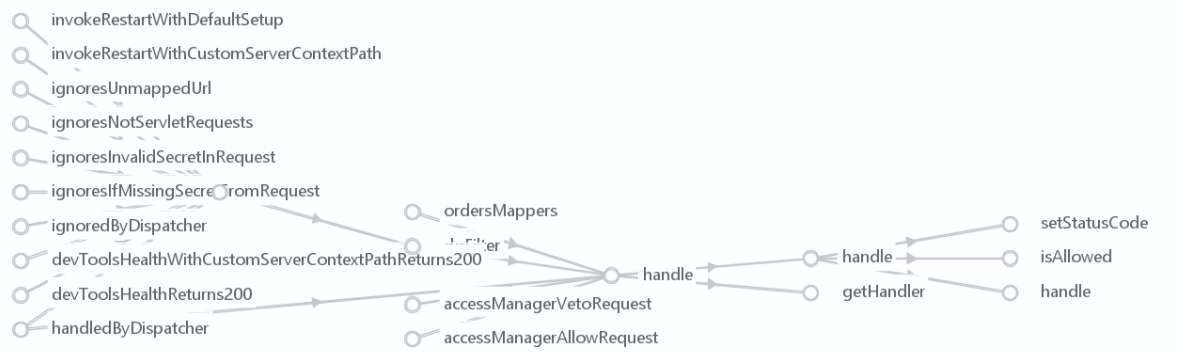
Then we take a look at the server side.





Server

The flow of handling request is as follows.

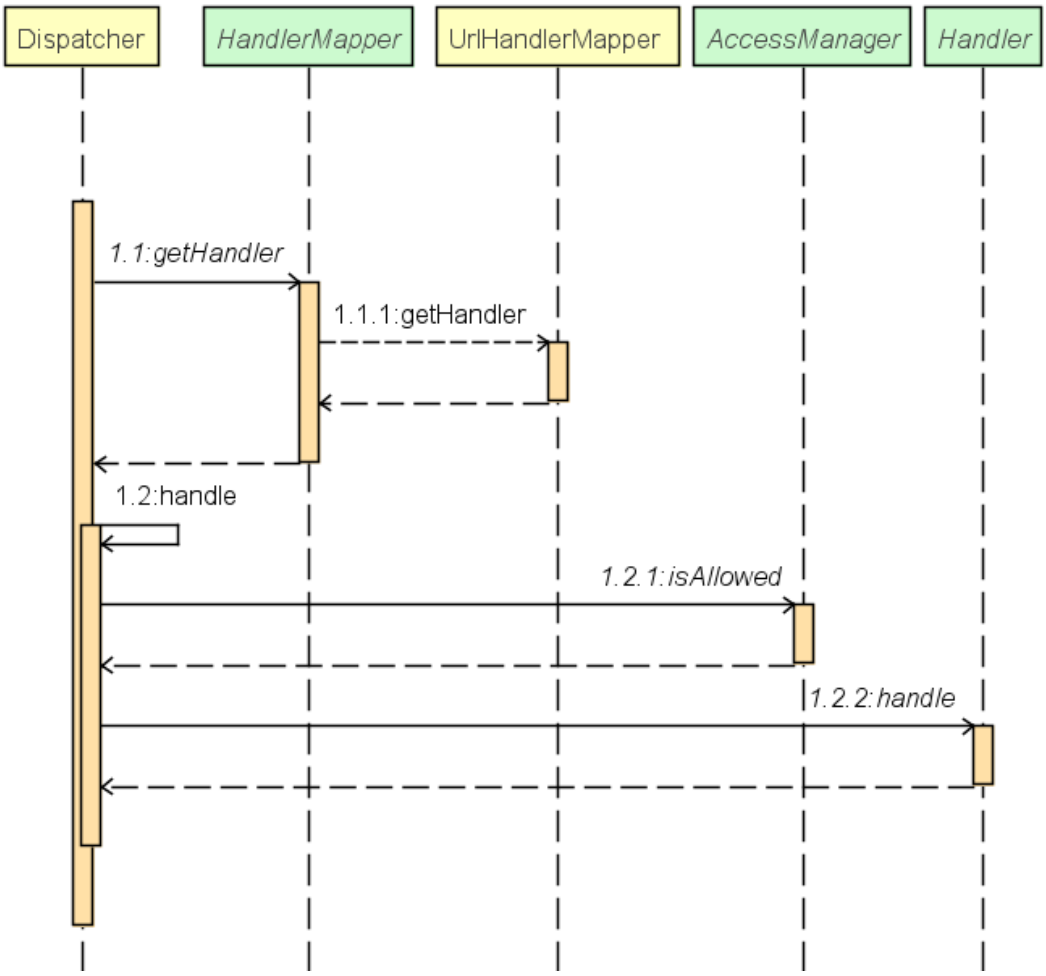


Firstly, the request is sent to the `Dispatcher` and it was sent to the `handle()` function. The Dispatcher will assign the request to its `HandlerMapper` which would then assign `Handler` to make response to that request.

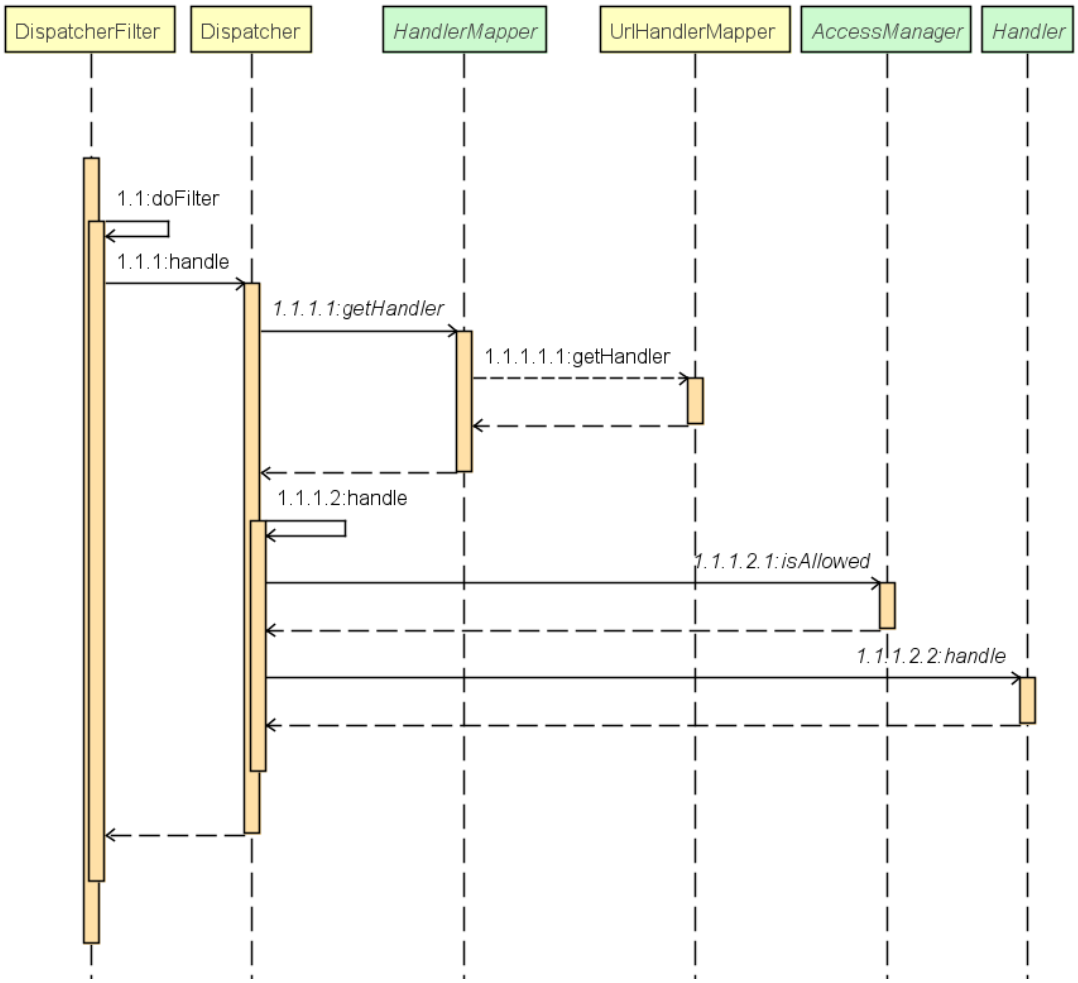
```
1  public boolean handle(ServerHttpRequest request, ServerHttpResponse
   response) throws IOException {
2          for (HandlerMapper mapper : this.mappers) {
3              Handler handler = mapper.getHandler(request);
4              if (handler != null) {
5                  handle(handler, request, response);
6                  return true;
7              }
8          }
9          return false;
10     }
```

Secondly, `HandlerMapper` is an interface implemented by `UrlHandlerMapper`. If the url maps the url inside the request, the handler will return the request. Or there must be the loss in communication, then it will return `null`.

```
public Handler getHandler(ServerHttpRequest request) {
        if (this.requestUri.equals(request.getURI().getPath())) {
            return this.handler;
        }
        return null;
    }
```



If needed, Spring-boot supports the `DispatcherFilter` before the `Dispatcher`.


DispatcherFilter
Dispatcher
HandlerMapper
UrlHandlerMapper
AccessManager
Handler
1.1:doFilter
1.1.1:handle
1.1.1.1:getHandler
1.1.1.1.1:getHandler
1.1.1.2:handle
1.1.1.2.1:isAllowed
1.1.1.2.2:handle