# Index

[What is GreenLightning?](#What)

* [Overview Video](https://drive.google.com/file/d/1QCL-pDSAcU0V4pS-qwXG8nzleBrb1kD8/view?ts=5aeb77b7)

[Links](#Links)

[Terms to Know](#Terms)

[Making a Project](#Making)

Importing a Project

* Eclipse (preferred)
* IntelliJ

[GreenLightning Coding Walkthroughs](#Walkthrough)

1. [Listeners](#listeners)
2. [PubSub](#pubsubtut)
3. [StateChange](#state)
4. [MQQTClient](#m4)
5. [HTTPServer](#_5)_HTTPServer)
6. [HTTPClient](#_6)_HTTPClient)

[Example Projects](#_Example_Projects)

1. Startup
2. Shutdown
3. Timer
4. MQQTClient
5. HTTPServer
6. HTTPClient

# What is GreenLightning?

GreenLightning is a performant light-weight microservice runtime that can execute garbage-free and lock-free in a Java compact-1 JVM. HTTP and MQTT are built in with easy and security-first APIs.

The framework is based on a declarative event driven actor model. Your resulting code ends up being smaller in complexity and easier to maintain. Operational logic such as thread synchronization, exception handling, and null object conditions are not only removed from the business logic but is handled by the framework.

In other words, you can easily write multithreaded asynchronous logic without having to schedule a single task or lock a single block of memory!

Another stand-out feature of GreenLightning is live fine-grained telemetry/operational information published via the built-in web-server. You can collect instantaneous metrics such as CPU% on a class level and data throughput without a specialized build and deployment. The information is available in production.

GreenLightning is a Java 8 functional API for embedded systems that's built on a small footprint, garbage free compact 1 Java web server and message routing platform,

GreenLightning is...

* Fast - Built on top of GreenLightning, FogLight is a garbage-free, lock-free, and low latency way to talk directly to hardware.
* Simple - Taking advantage of the latest Java 8 APIs, GreenLightning has a clean and fluent set of APIs that make it easy to learn and apply with minimal training.
* Secure - By taking advantage of the compile-time graph validation system, all GreenLightning applications can be compiled and compressed to a point where injecting malicious code into the final production JAR would prove difficult, if not impossible.

# Overview Video

<https://drive.google.com/file/d/1QCL-pDSAcU0V4pS-qwXG8nzleBrb1kD8/view?ts=5aeb77b7>

# Terms to Know

**MQTT** (**Message Queuing Telemetry Transport**) is an [ISO standard](https://en.wikipedia.org/wiki/International_Organization_for_Standardization) (ISO/IEC PRF 20922) [publish-subscribe](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern)-based messaging protocol. It works on top of the [TCP/IP protocol](https://en.wikipedia.org/wiki/TCP/IP). It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. The [publish-subscribe messaging pattern](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern) requires a [message broker](https://en.wikipedia.org/wiki/Message_broker).

The **Hypertext Transfer Protocol** (**HTTP**) is an [application protocol](https://en.wikipedia.org/wiki/Application_protocol) for distributed, collaborative, and [hypermedia](https://en.wikipedia.org/wiki/Hypermedia) information systems. HTTP is the foundation of data communication for the [World Wide Web](https://en.wikipedia.org/wiki/World_Wide_Web).

[Hypertext](https://en.wikipedia.org/wiki/Hypertext) is structured text that uses logical links ([hyperlinks](https://en.wikipedia.org/wiki/Hyperlinks)) between [nodes](https://en.wikipedia.org/wiki/Node_(networking)) containing text. HTTP is the protocol to exchange or transfer hypertext.

**HTTP Secure** (**HTTPS**) is an extension of HTTP for [secure communication](https://en.wikipedia.org/wiki/Secure_communications) over a [computer network](https://en.wikipedia.org/wiki/Network_operating_system), and is widely used on the [Internet](https://en.wikipedia.org/wiki/Internet). In HTTPS, the [communication protocol](https://en.wikipedia.org/wiki/Communication_protocol) is encrypted by [Transport Layer Security](https://en.wikipedia.org/wiki/Transport_Layer_Security) (TLS), or formerly, its predecessor, Secure Sockets Layer (SSL). The protocol is therefore also often referred to as **HTTP over TLS**, or **HTTP over SSL**.

**REpresentational State Transfer** (**REST**) is an architectural style that defines a set of constraints and properties based on [HTTP](https://en.wikipedia.org/wiki/HTTP). Web Services that conform to the REST architectural style, or **RESTful** [web services](https://en.wikipedia.org/wiki/Web_service), provide interoperability between computer systems on the [Internet](https://en.wikipedia.org/wiki/Internet). REST-compliant web services allow the requesting systems to access and manipulate textual representations of [web resources](https://en.wikipedia.org/wiki/Web_resource) by using a uniform and predefined set of [stateless](https://en.wikipedia.org/wiki/Stateless_protocol) operations.

**Publish–subscribe or PubSub** is a [messaging pattern](https://en.wikipedia.org/wiki/Messaging_pattern) where senders of [messages](https://en.wikipedia.org/wiki/Message_passing), called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

**Advantages**

**Loose coupling**

Publishers are [loosely coupled](https://en.wikipedia.org/wiki/Loose_coupling) to subscribers, and need not even know of their existence. With the topic being the focus, publishers and subscribers can remain ignorant of system topology. Each can continue to operate normally regardless of the other. In the traditional tightly coupled [client–server paradigm](https://en.wikipedia.org/wiki/Client%E2%80%93server_model), the client cannot post messages to the server while the server process is not running, nor can the server receive messages unless the client is running. Many pub/sub systems decouple not only the locations of the publishers and subscribers, but also decouple them temporally. A common strategy used by [middleware analysts](https://en.wikipedia.org/wiki/Middleware_analyst) with such pub/sub systems is to take down a publisher to allow the subscriber to work through the backlog (a form of [bandwidth throttling](https://en.wikipedia.org/wiki/Bandwidth_throttling)).

**Scalability**

Provides the opportunity for better [scalability](https://en.wikipedia.org/wiki/Scalability) than traditional client–server, through parallel operation, message caching, tree-based or network-based routing, etc. However, in certain types of tightly coupled, high-volume enterprise environments, as systems scale up to become data centers with thousands of servers sharing the pub/sub infrastructure, current vendor systems often lose this benefit; scalability for pub/sub products under high load in these contexts is a research challenge.

Outside of the enterprise environment, on the other hand, the pub/sub paradigm has proven its scalability to volumes far beyond those of a single data center, providing Internet-wide distributed messaging through web syndication protocols such as [RSS](https://en.wikipedia.org/wiki/RSS) and [Atom](https://en.wikipedia.org/wiki/Atom_(standard)). These syndication protocols accept higher latency and lack of delivery guarantees in exchange for the ability for even a low-end web server to syndicate messages to (potentially) millions of separate subscriber nodes.

**Disadvantages**

The most serious problems with pub/sub systems are a side-effect of their main advantage: the decoupling of publisher from subscriber.

**Message delivery issues**

A pub/sub system must be designed carefully to be able to provide stronger system properties that an application might require, such as assured delivery.

* The broker in a pub/sub system may be designed to deliver messages for a specified time, but then stop attempting delivery, whether it has received confirmation of successful receipt of the message by all subscribers. A pub/sub system designed in this way cannot guarantee delivery of messages to any applications that might require such assured delivery. Tighter coupling of the designs of such a publisher and subscriber pair must be enforced outside of the pub/sub architecture to accomplish such assured delivery (e.g. by requiring the subscriber to publish receipt messages).
* A publisher in a pub/sub system may assume that a subscriber is listening, when in fact it is not. A factory may utilize a pub/sub system where equipment can publish problems or failures to a subscriber that displays and logs those problems. If the logger fails (crashes), equipment problem publishers won't necessarily receive notice of the logger failure, and error messages will not be displayed or recorded by any equipment on the pub/sub system. This is also a design challenge for alternative messaging architectures, such as a client/server system. In a client/server system, when an error logger fails, the system will receive an indication of the error logger (server) failure. However, the client/server system will have to deal with that failure by having redundant logging servers online, or by dynamically spawning fallback logging servers. This adds complexity to the client and server designs, as well as to the client/server architecture. However, in a pub/sub system, redundant logging subscribers that are exact duplicates of the existing logger can be added to the system to increase logging reliability without any impact to any other equipment on the system. In a pub/sub system, the feature of assured error message logging can be added incrementally, after implementing the basic functionality of equipment problem message logging.

The pub/sub pattern scales well for small networks with a small number of publisher and subscriber nodes and low message volume. However, as the number of nodes and messages grows, the likelihood of instabilities increases, limiting the maximum scalability of a pub/sub network. Example throughput instabilities at large scales include:

* Load surges—periods when subscriber requests saturate network throughput followed by periods of low message volume (underutilized network bandwidth)
* Slowdowns—as increasingly applications use the system (even if they are communicating on separate pub/sub channels) the message volume flow to an individual subscriber will slow

For pub/sub systems that use brokers (servers), the argument for a broker to send messages to a subscriber is [in-band](https://en.wikipedia.org/wiki/In-band_control), and can be subject to security problems. Brokers might be fooled into sending notifications to the wrong client, amplifying denial of service requests against the client. Brokers themselves could be overloaded as they allocate resources to track created subscriptions.

Even with systems that do not rely on brokers, a subscriber might be able to receive data that it is not authorized to receive. An unauthorized publisher may be able to introduce incorrect or damaging messages into the pub/sub system. This is especially true with systems that [broadcast](https://en.wikipedia.org/wiki/Broadcasting_(computing)) or [multicast](https://en.wikipedia.org/wiki/Multicast) their messages. [Encryption](https://en.wikipedia.org/wiki/Encryption) (e.g. [Transport Layer Security](https://en.wikipedia.org/wiki/Transport_Layer_Security) (SSL/TLS)) can prevent unauthorized access, but cannot prevent damaging messages from being introduced by authorized publishers. Architectures other than pub/sub, such as client/server systems, are also vulnerable to authorized message senders that behave maliciously.

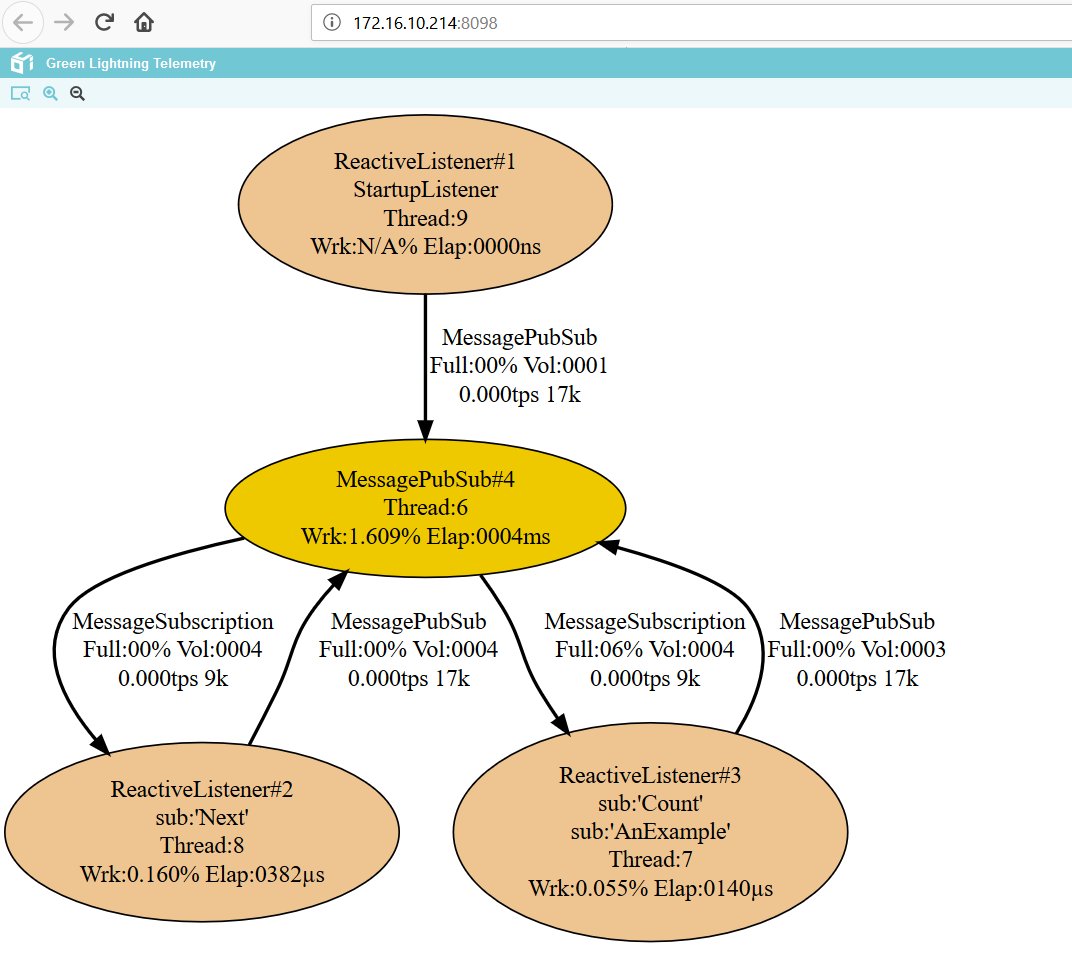
Publish–subscribe is a sibling of the [message queue](https://en.wikipedia.org/wiki/Message_queue) paradigm, and is typically one part of a larger [message-oriented middleware](https://en.wikipedia.org/wiki/Message-oriented_middleware) system. Most messaging systems support both the pub/sub and message queue models in their [API](https://en.wikipedia.org/wiki/Application_programming_interface), e.g. [Java Message Service](https://en.wikipedia.org/wiki/Java_Message_Service) (JMS).

This pattern provides greater network [scalability](https://en.wikipedia.org/wiki/Scalability) and a more dynamic [network topology](https://en.wikipedia.org/wiki/Network_topology), with a resulting decreased flexibility to modify the publisher and the structure of the published data.

**The Actor-Based Approach** or **actor model** in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation. In response to a message that it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state but can only affect each other through messages (avoiding the need for any locks).

**Telemetry** is an automated communications process by which measurements and other data are collected at remote or inaccessible points and transmitted to receiving equipment for monitoring.

Below is what GreenLightning’s telemetry might look like:



# 

# Links

Starter Project Instructions: <https://github.com/oci-pronghorn/GreenLighter>

Sample Projects: <https://github.com/oci-pronghorn/GreenLightning-API> <https://github.com/oci-pronghorn/GreenLightning-Examples>

The code on GitHub: <https://github.com/oci-pronghorn/GreenLightning>

The build server: <https://pronghorn.ci.cloudbees.com/job/GreenLightning/>

Performance Comparisons: <https://github.com/oci-pronghorn/GreenLoader>

# Making a Project

**GreenLighter (A Maven Archetype for GreenLightning projects)**

**What you will need before you start:**

[**Java 8**](https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html)

[**Maven**](https://maven.apache.org/install.html)

* which downloads and manages the libraries and APIs needed to get the Grove device working.

[**Git**](https://git-scm.com/)

* which clones a template Maven project with the necessary dependencies already set up.

[**PuTTY**](http://www.putty.org/)

* which allows you to interact and control your device

**Starting Your Own Project**

In the command line or terminal of your local machine, enter:

git clone https://github.com/oci-pronghorn/GreenLighter.git

cd GreenLighter

mvn install

**Create a new directory outside of the GreenLighter project directory for your new IoT project.**

Now, cd into the directory for your to be created IoT project, and enter:

mvn archetype: generate -DarchetypeGroupId=com.ociweb -DarchetypeArtifactId=GreenLighter -DarchetypeVersion=0.1.0-SNAPSHOT

The terminal now asks for:

groupID: type in *com.ociweb* then press Enter

ArtifactID: type in name of your project then press Enter

version: 1.0-SNAPSHOT: Ignore, Press Enter

package: com.ociweb: Ignore, Press Enter

Y:: Type *Y*, press Enter

This will create a folder named after your project, which includes all the project files. Let’s call our project *ProjectXYZ*.  
If you’re working from Terminal, open the file “ProjectXYZ”/src/main/java/com/ociweb/IoTApp.java . You can start implementing the project code from here.

If you’re using an IDE, open the created Maven project - *ProjectXYZ* and start working from IoTApp.java

Once you’re done with the implementation, open your project folder in terminal and type

mvn install

.. to build the project. This will create a .jar file named ProjectXYZ.jar in the **/target** folder (note that there are other .jar files in **/target**, but we don’t have to worry about those). This jar is executable and contains all its needed dependencies. Transfer this .jar file to your device and use the command

java -jar ProjectXYZ.jar

.. to execute it.

**Importing the Maven project in Eclipse**

Select File -> Import

Click on "Existing Maven Projects" under Maven, then click "Next"

Click "Browse" and select the directory (folder) under your project that contains the "src" folder as well as a "pom.xml" file.

Click "finish"

**Importing the Maven project in NetBeans**

Select File -> Open Project

Browse to the directory (folder) under your project that contains the "src" folder as well as a "pom.xml" file.

Click "Open Project"

Note: In NetBeans, instead of typing mvn install, you can also build your project by clicking "Build".

**Importing the Maven project in IntelliJ**

Select File -> Open.

Browse to the directory (folder) under your project that contains the "src" folder as well as a "pom.xml" file.

Click "OK".

The import will be performed automatically.

**Building the project**

Open your project folder in the terminal of your choosing and type

### Signing your project with your own key:

We have provided a key to sign your project; however, if you would like to use a key of your own choosing, you can do so by replacing the ocikeystore.jks file located in **projectname/src/main/resources/certificate/ocikeystore.jks** with your own .jks key or one provided from an official Certificate Authority (CA).

.. To create your own jks file, simply navigate to your jdk\_version folder on your computer and locate the bin folder. Within the folder you will see a .exe file called keytool. While at the CMD of this folder type: "keytool -genkey -alias my\_certificate\_alias -keyalg RSA -keysize 4096 -keystore keystore.jks" press enter and you will be prompted with a wizard to fill in the rest. Take note of your password and alias. Next, using whichever IDE you prefer; navigate to the POM file within the main folder and edit the properties section at the top to fit the values of your own key. Replace the alias with the alias you just used as well as the password. Change the last part of the **keystore.path** section from "../ocikeystore.jks" to the name of your "../keystore.filetype".

.. If you are using a Certificate Authority (CA) to sign your project, simply put the information related to the CA in the properties of the POM file. The project will then automatically be signed when you perform a mvn install in the following step.

mvn install

.. to build the project. This will create a .jar file named ProjectXYZ.jar in the \*\*/target\*\* folder (note that there are other .jar files in \*\*/target\*\*, but we don’t have to worry about those). This jar is executable and contains all its needed dependencies.

### Importing and running the project to your device

After successfully building the project, ```cd``` into the \*\*/target\*\* folder. Now, use

scp ProjectXYZ.jar username@servername:

This will send the jar file to your RaspberryPi. You can also send the jar file to a specific location by adding the file path after the colon. For example, if your username was "pi", the server name was "raspberry" and you wanted to add the .jar file to your Projects folder, the command would look like this ...

scp ProjectXYZ.jar pi@raspberry:/Projects/

Once the project is your device, use PuTTY to connect to your device. In PuTTY, if needed, ```cd``` into the folder containing your .jar file and then use the following command on your device...

java -jar ProjectXYZ.jar

.. to execute it. To exit the app at any time, press Ctrl+c.

## Options

--t is for turning on or off TLS (https) support, keys are built in can’t change yet

--s is the folder holding the static site pages, these cannot be changed while its running note if this points to the index.html file it will be used as the default for the domain.

--l is for running in large mode where it allocates many GB for performance with -l False (the small mode) it is still much more performant than any other server I could find.

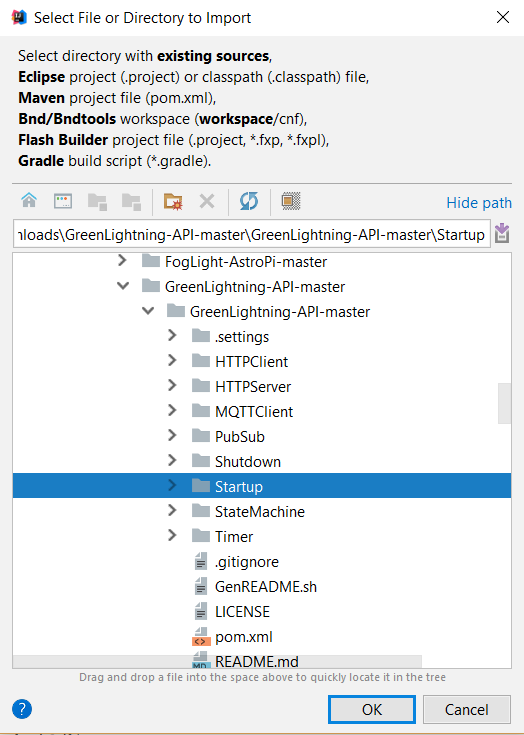
--h what host to run this on (e.g. ip) if not provided it will guess from network settings.

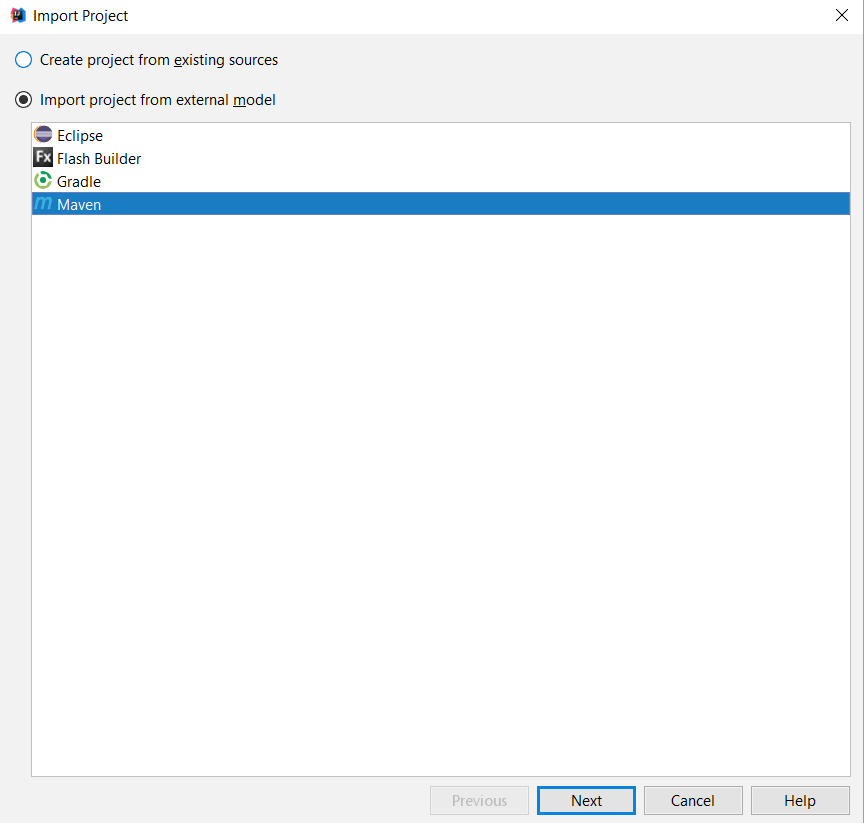
--p what port to run on the default is 8080 even when TLS is turned on.

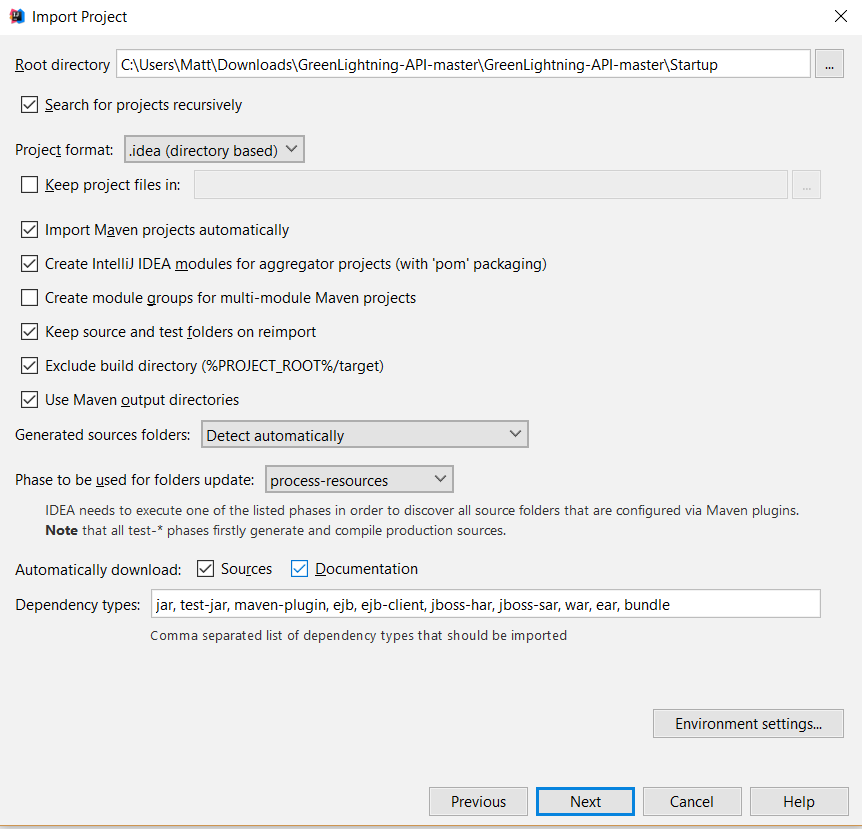
When running small mode on a compact 1 JVM the app will only use about 60MB. When running small mode on a normal JVM the app will use about 160MB.

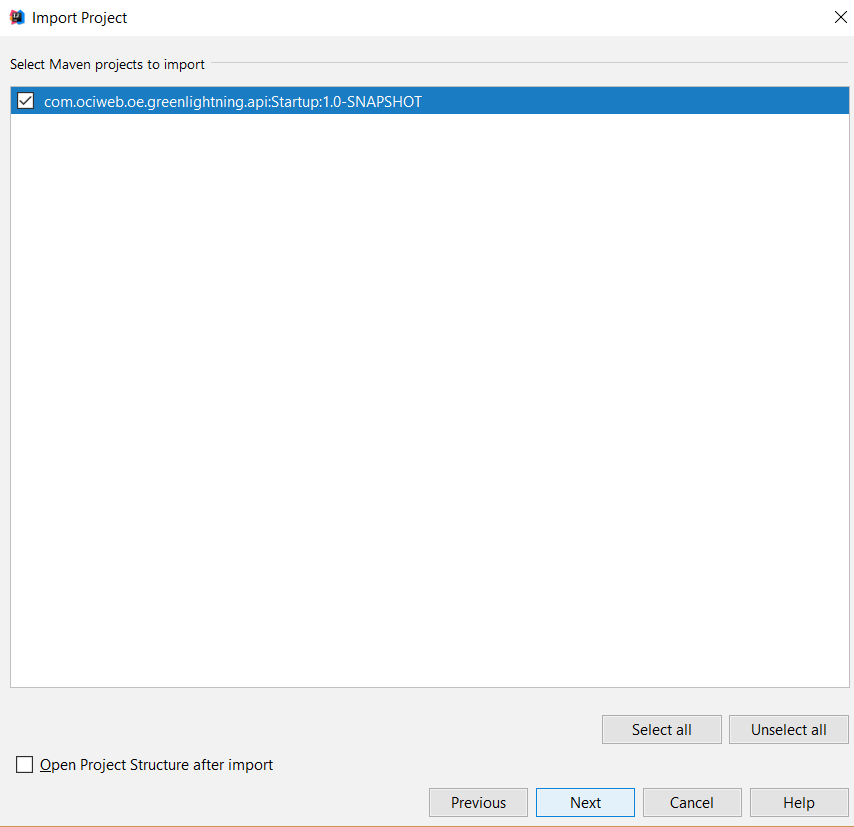
# Opening the project using Eclipse

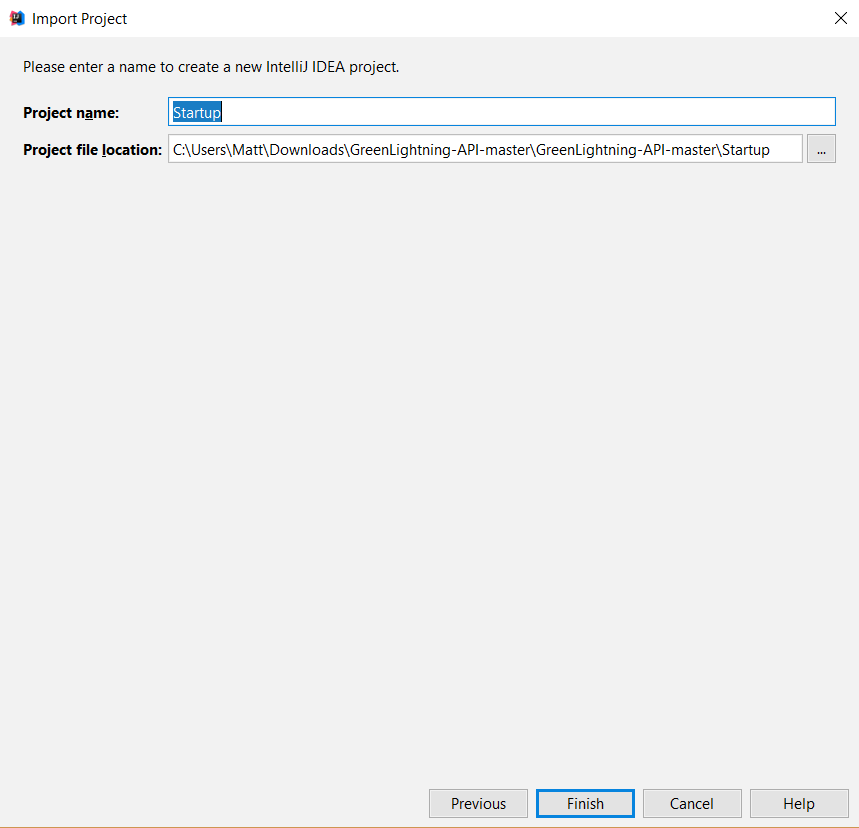
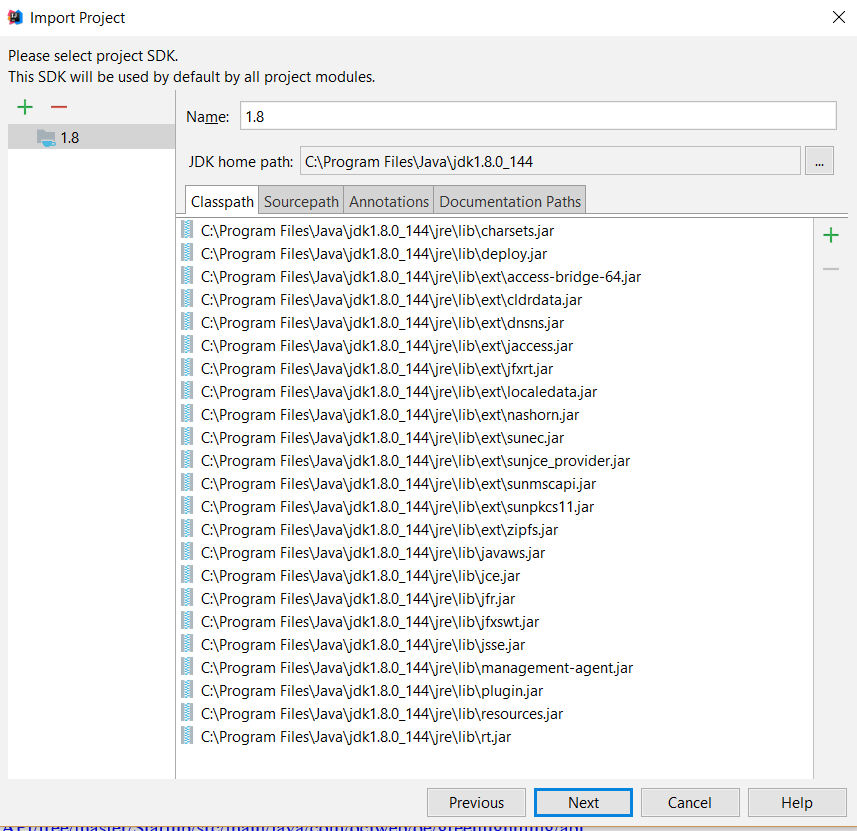
# Opening the project using IntelliJ



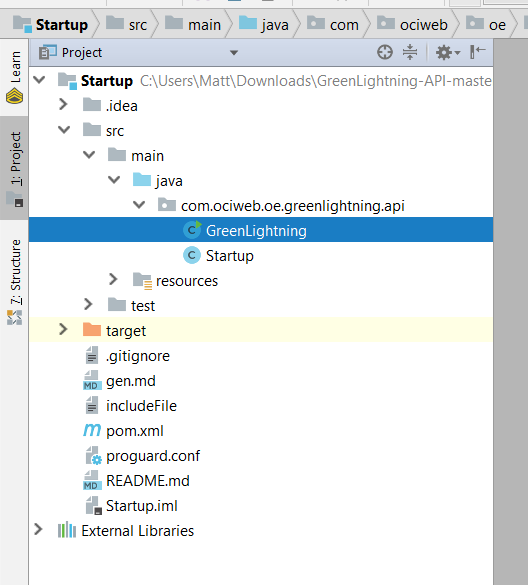








The Java files can be found in src -> main -> java -> com.ociweb.oe.greenlightning.api.



# 

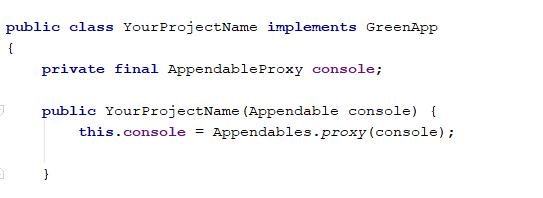
# GreenLightning Coding Walkthroughs

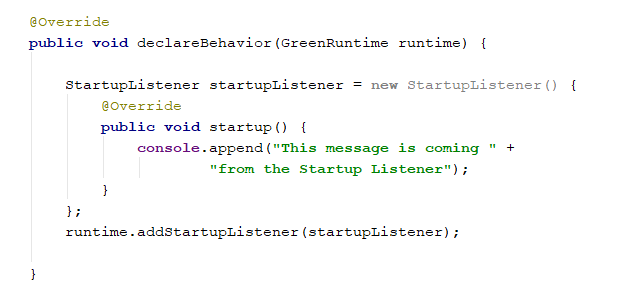
# Listeners

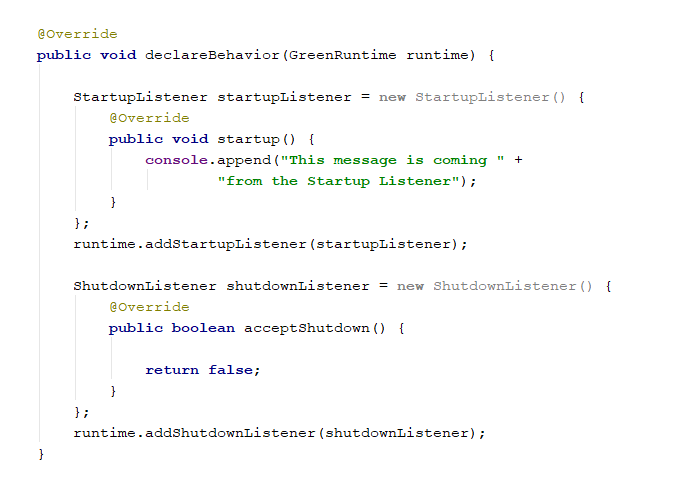
This example will introduce StartupListener, ShutdownListener, and TimeListener.

* 1. Follow the instructions in [making a project](#Making) to set up a new project.
  2. Open the project in your IDE such as [Eclipse](#intellij).
  3. Your project will have GreenLightning.java and YourProjectName.java inside the src folder. Open or double-click YourProjectName.java.
  4. GreenLightning behavior classes have Configuration and Behavior parts. Configuration is where you define routes, paths, enable telemetry, etc. Behavior is where we register behaviors, listeners, etc., and deal with fields and enumerators.
  5. Before we start, we’re going to enable logging to the console using Appendables. This also needs to be instantiated, so add a constructor for this class and pass in console. \*\*\*change to system.out­

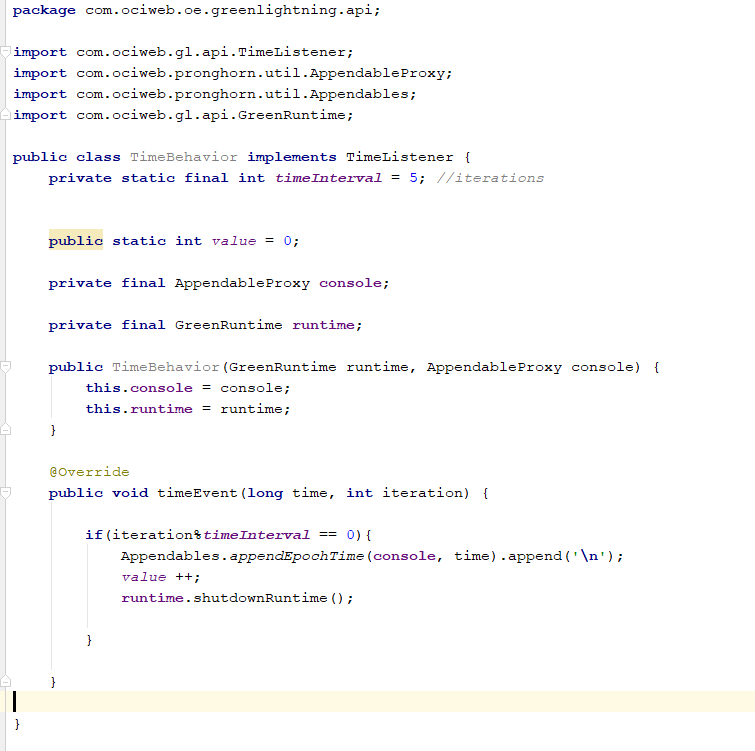
It should look something like this.



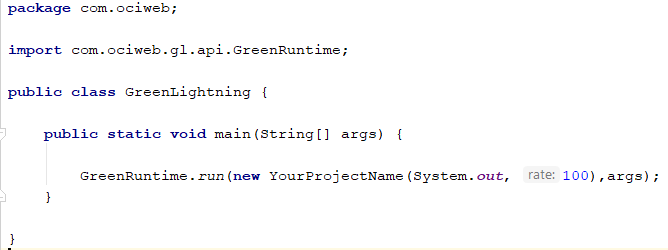
* 1. First, we’re going to add a StartupListener. This should go in the Behavior section of YourProjectName.java. A StartupListener is the only void Listener, as it doesn’t need to return anything.
  2. We’re only going to use console.append here to print that the app has started. It should look like this. 
  3. Next, we’re going to add a ShutdownListener. ShutdownListeners are Boolean methods, which are called when the app is shutting down. When a behavior calls runtime.shutDown(), all of the threads with a ShutdownListener are going to call acceptShutdown(), and if any of them return false, shutdown will be blocked by that thread and it will come back to it until it returns true. If all return true, then the app will shut down. For now, acceptShutdown() can just return false, and a condition will be added later.

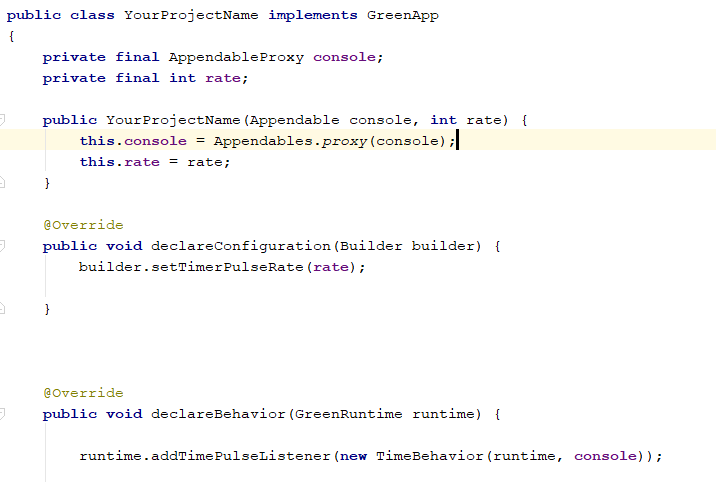


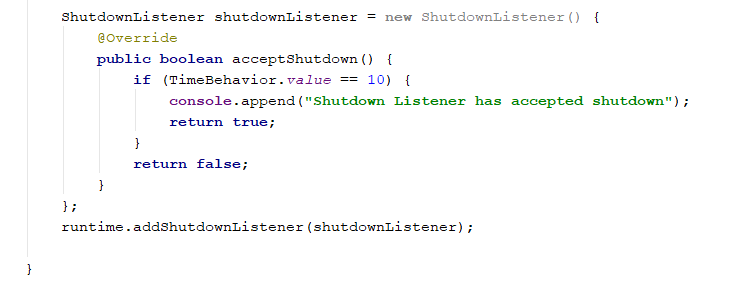
* 1. Now, in order to test startup and shutdown, we’re going to add another class called TimeBehavior, where we’re going to use the TimeListener. We’re going to have it log to the console for every second that passes. We’re also going to use an int variable and increment it every second, so that we can have a condition to test acceptShutdown() on (i.e. when five seconds have passed, or count == 5).
  2. After we have these, we can try and shut down the app inside of a Behavior, using runtime.shutdownRuntime().
  3. Now, create a new Java class inside the src folder of the project and name it TimeBehavior or something similar.
  4. Add another console variable the same way as before and create a variable to store your counter.
  5. Now, have this class implement TimeListener.
  6. Create a timeEvent method, and have it print to the console every second and increment the counter. After this, call runtime.shutdownRuntime(). Note that this will be called every second, but it’s going to get denied once we change acceptShutdown()’s return condition. This class should look like this.

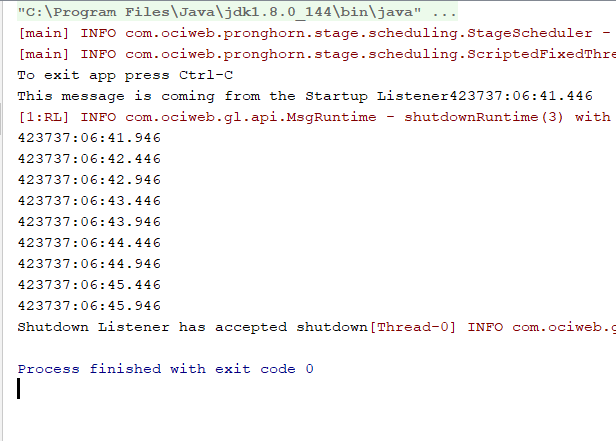


\*\*Note: Putting runtime.shutdownRuntime() inside this method is unnecessary, as this only needs to be called once – once it’s been called, the app will be trying to shut down as long as it runs. It is put here to illustrate that the shutdown is being denied. \*\*

* 1. System.out and a timer value will need to be passed in GreenLightning.java, like Nthis. 
  2. Also, implement the Time Listener in the original class like this, by adding a rate variable, and in the configuration method, setting the builder’s TimerPulseRate to that rate, and adding a TimePulseListener to runtime in the behavior section.



* 1. Now go back to the ShutdownListener and change the return statement to check the counter variable from the timer against a value, such as 5, causing it to return true when the counter is 5, and print that it has passed. 
  2. Now run the application. The Startup Listener should have printed its message. Now the app will log the time as it passes. It will be calling shutdown runtime, but won’t shut down until 5 seconds have passed to pass acceptShutdown()’s condition, and the Shutdown Listener’s console message should have been displayed.



# 02) PubSub

The following program will demonstrate a simple use of the addPubSubListener() method.

Run main in GreenLightning to start this application.

The code will generate seven random, lucky numbers. The first addPubSubListener() will generate a random number and add it to ArrayList luckyNums. Once that has occurred, it will publish a message under the topic of "Gen", which the second PubSubListener is subscribed to, meaning that it is always listening for any publication under that topic. The second PubSubListener will simply print out the newest lucky number, then publish a message under the topic of "Print", which the first PubSubListener is subscribed to, restarting the process for a total of seven rounds.

Clicking the address in “Telemetry Server is now ready” will open the live telemetry chart in your browser.

* 1. Follow the same instructions as before to start another new project.
  2. Create a class PubSub.java. We’re going to use a StartupListener again, but this time we’ll have our own implementation of its startup. It’s going to be called KickoffBehavior. Also, this class will have a PubSubListener, which will be implemented in GenerateBehavior. Lastly, we’re going to have a CountBehavior. Lastly, enable telemetry in the configuration.

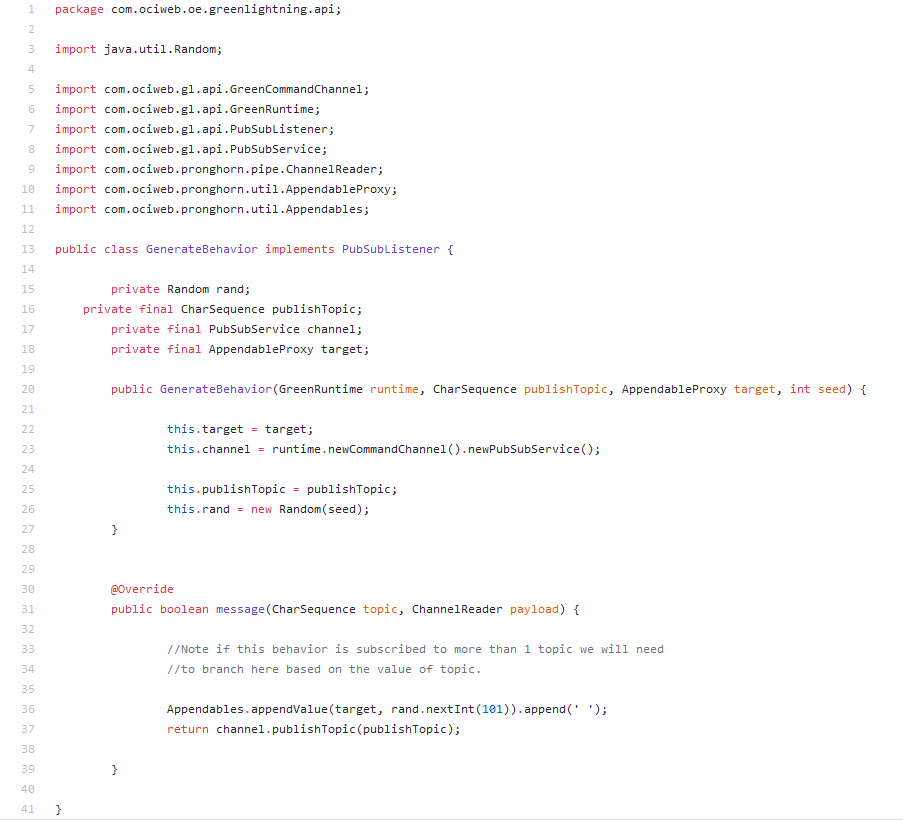
This is how the class should look:



* 1. Now, add the classes mentioned above (KickoffBehavior.java, GenerateBehavior.java, and CountBehavior.java).
  2. KickoffBehavior will need to implement StartupListener, so it needs the startup() method. It will also need a PubSubService variable. It should look like this:



* 1. GenerateBehavior will need to implement PubSubListener. It will need a Rand, a CharSequence for the topic, and a channel for the PubSubService. It will also have a boolean method that will return whether the channel was able to publish the topic. It will look like this:



* 1. Lastly, CountBehavior will implement PubSubMethodListener (note: not PubSubListener). It will have a channel for the PubSubService, a counter variable, a CharSequence for the topic like the previous class, and a Boolean variable for doing shutdown.
  2. This class will have a Boolean method for the logic behind counting the topics. If the count is below six, we publish a topic (storing this in a Boolean called result) and if it was successful, increment the count, then return result’s value. Otherwise, shut down the application, if the shutdown variable is true.

# 

* 1. GreenLightning.java should look like this:

# 

* 1. Running the application should yield this result:

# 

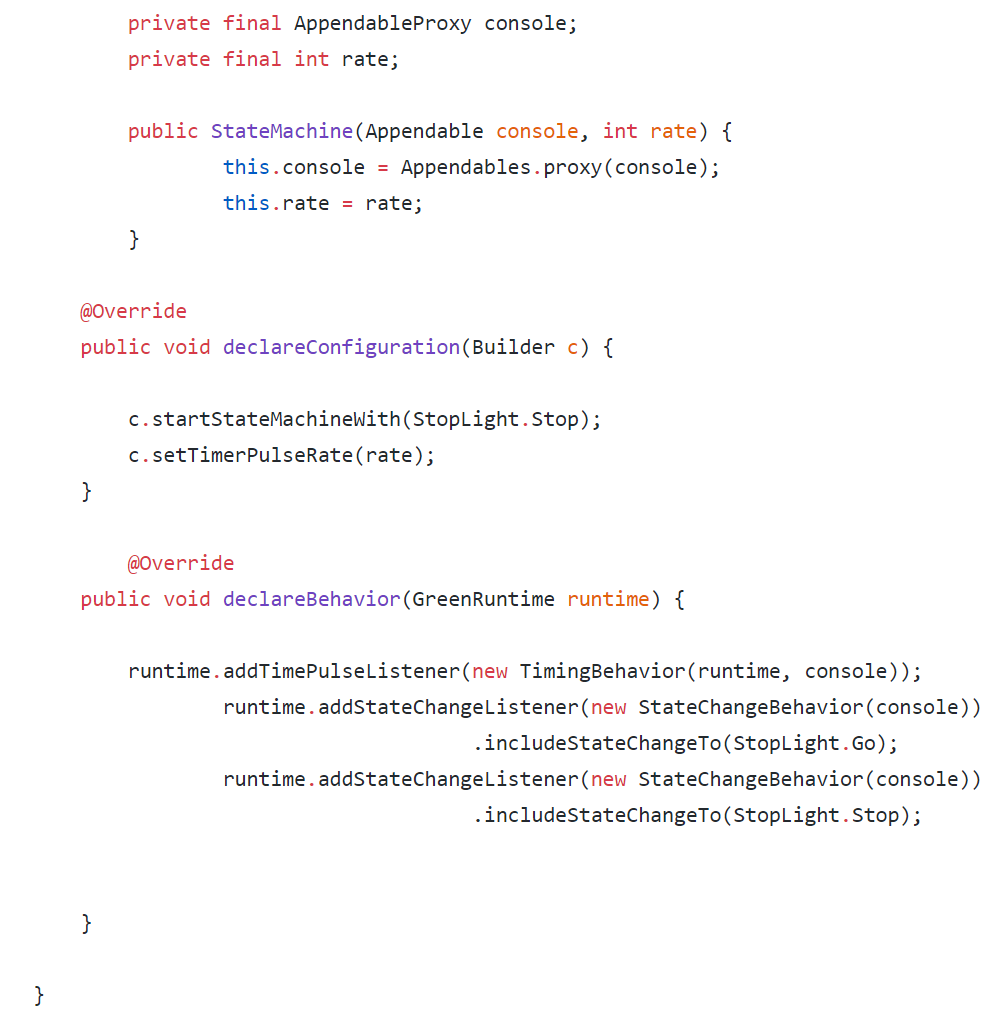
* 1. Clicking on the telemetry server should open a view like this:

# 

# StateChange

These classes are a basic demo of how to use the StateChangeListener method. In the main class, a stop light is simulated with 3 different states, Go, Caution, and Stop. In the declareConnections section, the stop light is initialized to the Stopstate to begin with. If a state is initialized there, you use a changeState() in a StartupListener as the two will clash when starting the program, so you must use one or the other. In the TimeBehavior class, a TimeListener is being used to change the state the state of the stop light. Every 5 seconds, the state is changed to the next state in the progression. In the StateChangeBehavior class, there is a StateChangeListener. Whenever it hears a change in state, it will print the new states color and will return true.

1. Create a new GreenLightning project named StateMachine.
2. In StateMachine.java, in the declareBehavior method, add a TimePulseListener and two StateChangeListeners, one for Go and one for Stop. Also, in declareConfiguration, start the state machine and set the timer pulse rate to your rate variable. See below:



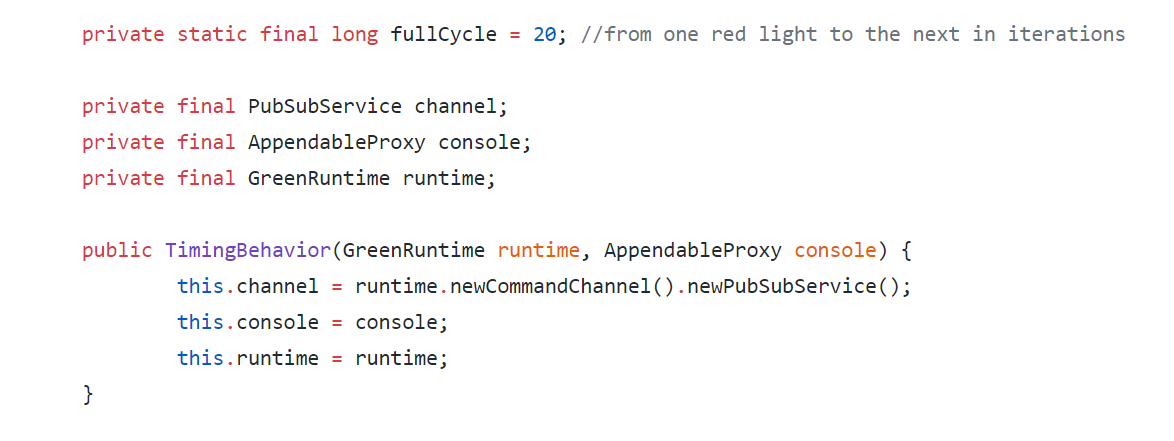
1. Add an enum for a StopLight object. Make a String for color. Go will be “green”, Caution will be “yellow”, and Stop will be “red”. Add a constructor for StopLight where its color = lightColor, and a getter for the color.­ It should look like this:



1. Add a class called StateChangeBehavior. It will need a stateChange method that takes in the new and old states of the StopLight. In it, append to the console the state change. The class will look like this.



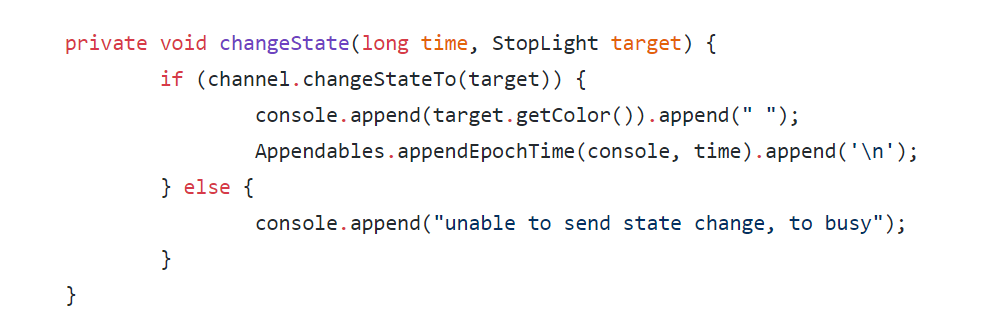
1. The last class to add is TimeBehavior, to implement TimeListener.
2. Add these variables:



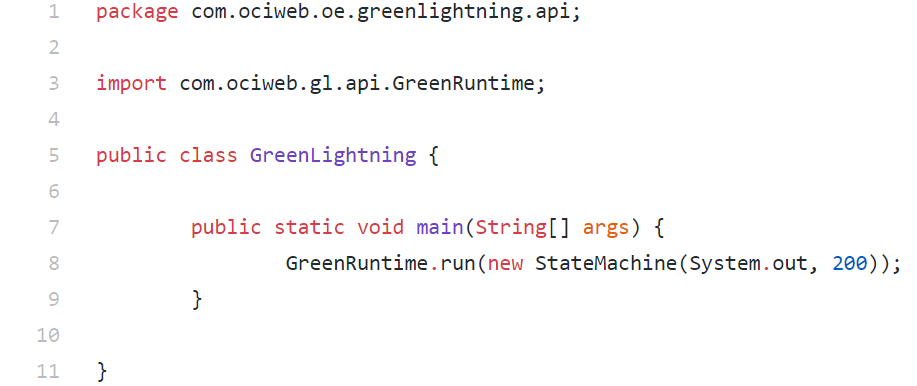
1. It will need the timeEvent method. The logic for this method will be using iteration mod fullCycle.
2. If the result of iteration mod full cycle is 0, then change the state to Go. If it is 8, then it is Caution or yellow. If it is 11, then change to Stop. Lastly, if the iteration is 3 full cycles, shut down the app by calling runtime.shutdownRuntime(7) – 7 is the delay amount. See below:



1. The last thing to add to this class is a void changeState method which takes a long called time and a StopLight called target. If the channel is changing the state to the target, print to the console the color of the StopLight and the time. Otherwise, print that it had failed. Like this:



1. The last thing to do is edit GreenLightning.java as follows:



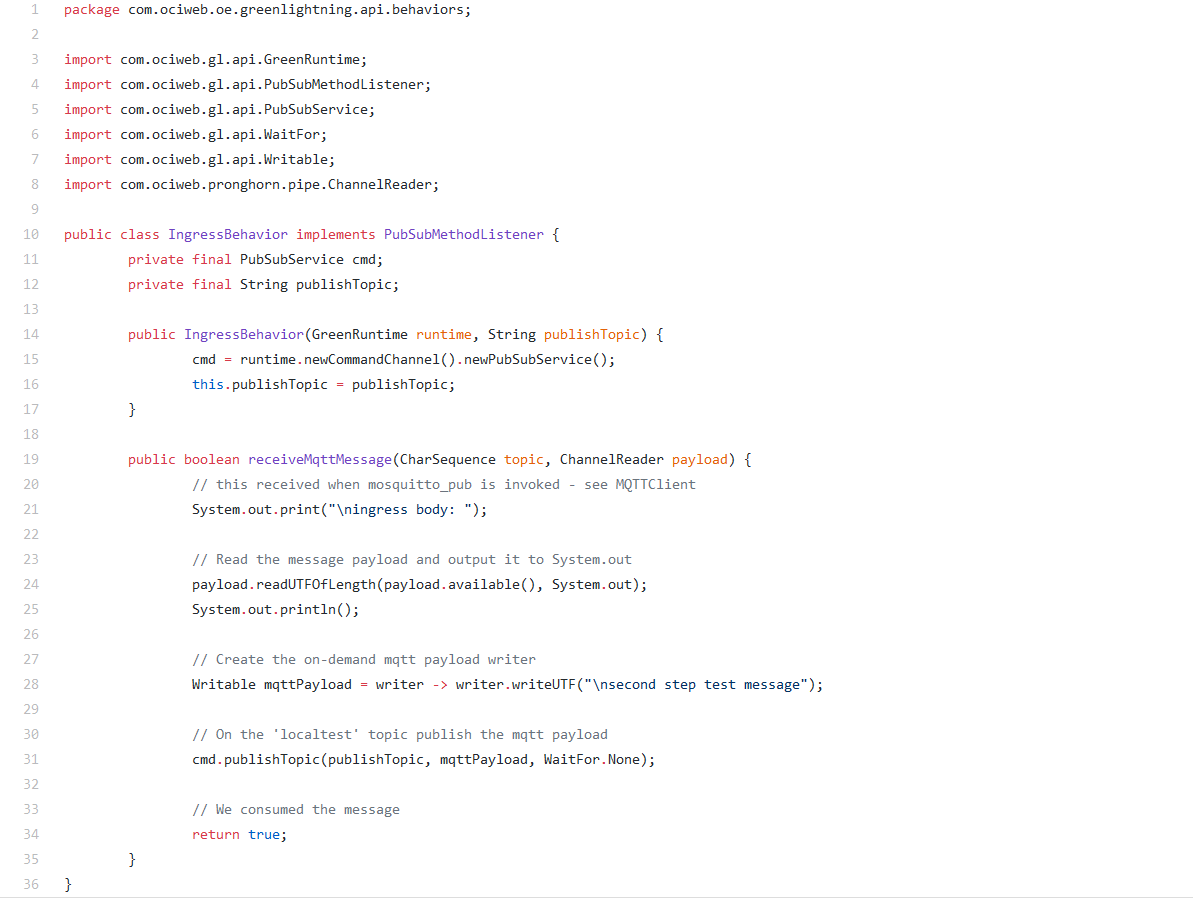
# MQQTClient

This class is a simple demonstration of MQTT (Message Queue Telemetry Transport). A lightweight messaging protocol, it was initially designed for constrained devices and low-bandwidth, high-latency or unreliable networks. This demo uses Mosquitto as a message broker, which means that the messages that are published will go through Mosquitto, which will send them to and subscribers of the topic.

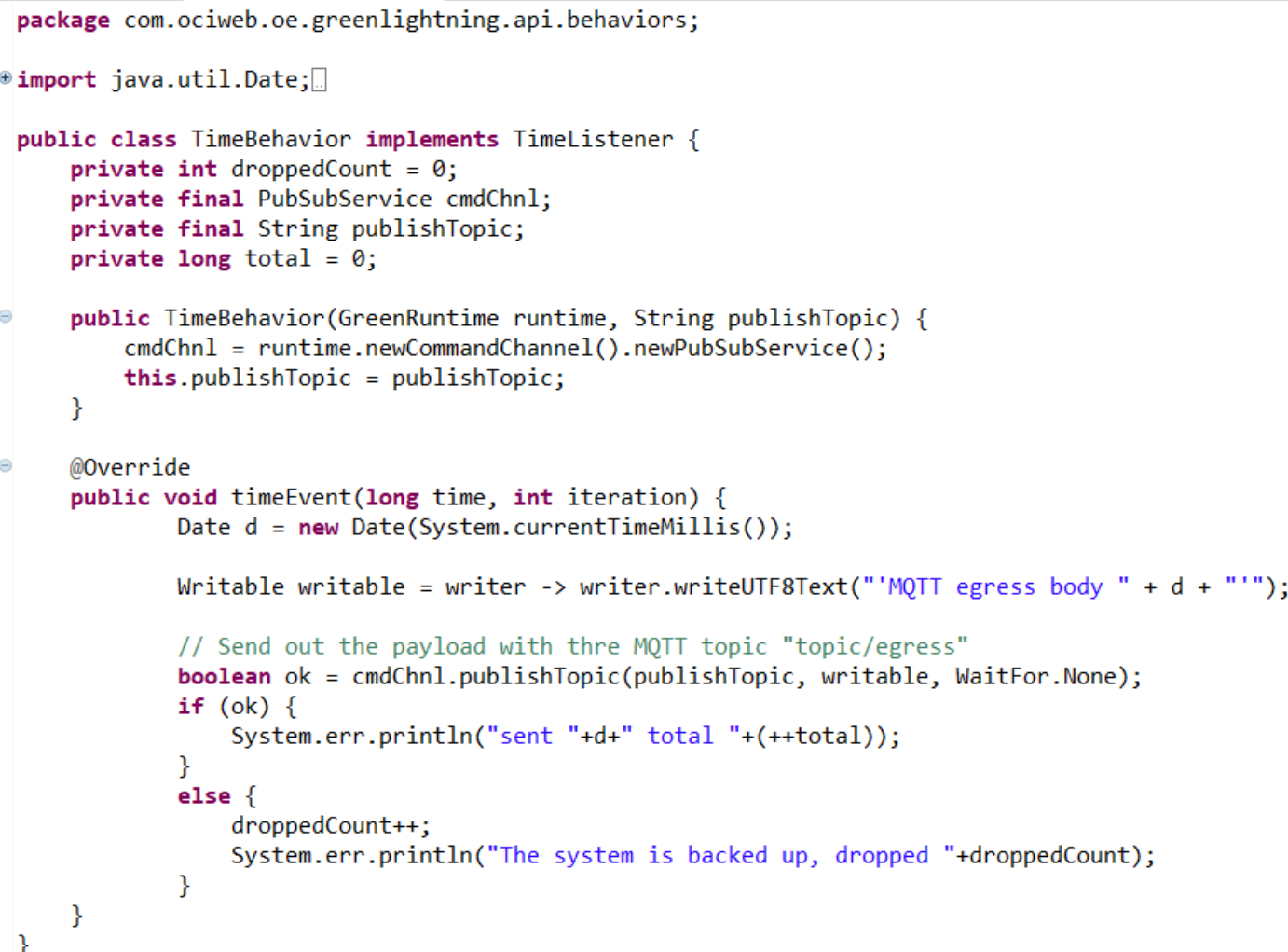
* + 1. Make a new project called MQQTClient.
    2. This project will use three Behavior classes: IngressBehavior, EgressBehavior, and TimeBehavior. Make a behavior class folder inside src and make these classes.
    3. EgressBehavior will implement PubSubMethodListener and will have a Boolean method receiveTestTopic, taking a CharSequence topic and a ChannelReader payload. It will print when it gets a topic and payload then return true. The class will look like this:



* + 1. IngressBehavior will also implement PubSubMethodListener. It will have a command variable for the PubSubService and a String for publishing the topic. Additionally, implement a boolean method for handling the reception of an MQQT message, taking the same parameters as receiveTestTopic. The payload should call readUTFOfLength to the console to read the message payload and output it to System.out, and create a mqqtPayload variable of type Writable to writeUTF to create an on-demand MQQT payload writer. Lastly, on the ‘localtest’ topic publish the MQQT payload using cmd.publishTopic. The finished class will resemble this:



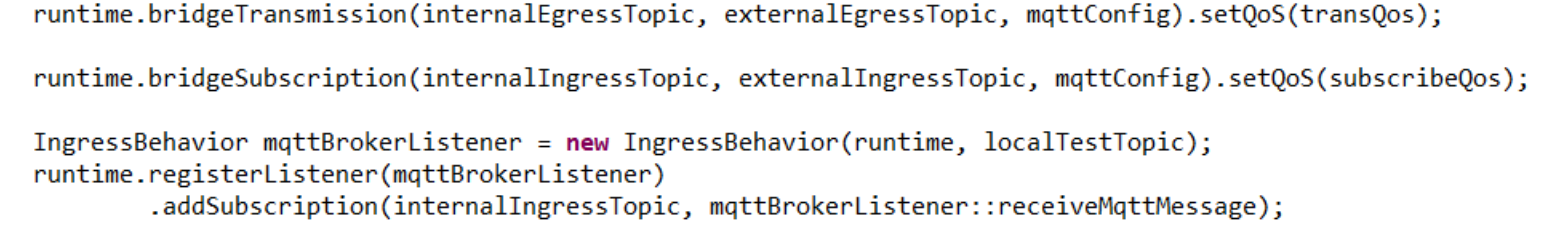
* + 1. The last behavior class to implement is TimeBehavior.java. As before, it implements TimeListener so it will need a timeEvent method. This method will need a command channel for PubSubService, a String publishTopic, and will use a droppedCount int and a long total variable. \
    2. Now, inside of the timeEvent method, make a variable that gets the current time as well as a Writable using a writer that writes ‘MQQT egress body’ and the time to create a payload with the string encoded timestamp.
    3. To finish this method off, make a Boolean that is evaluated by the command channel publishing the topic. Follow it with an if statement on that Boolean, printing a success’ contents or the failure with the dropped count (increment droppedCount before). See below:



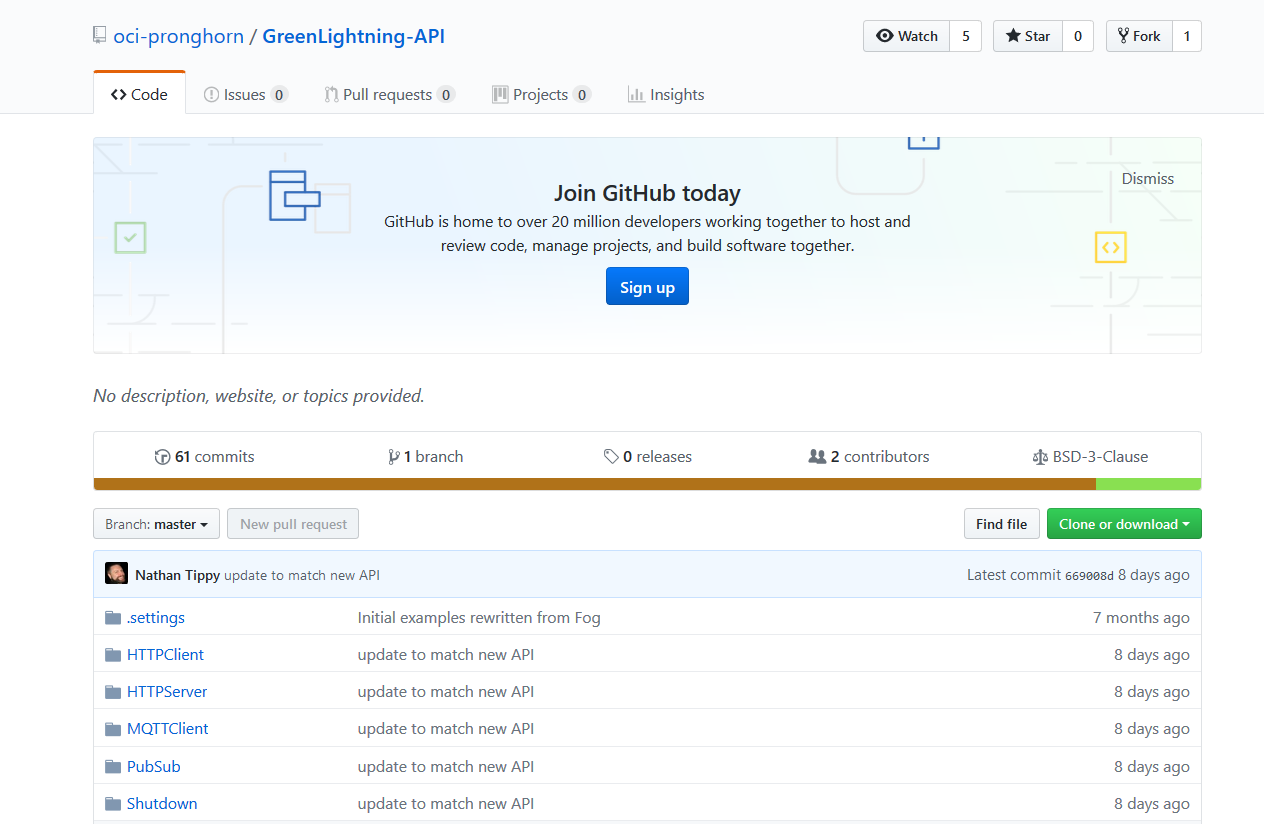
* + 1. Now, to implement MQQTClient.java, start by adding a MQQTBridge variable.
    2. Add internal and external egress topics as well as a local test topic into the behavior. For example, a internal egress topic would look like “internal/topic/egress” (but external/internal topic translation is not necessary).
    3. Now, inject the timer that publishes topic/egress. Make a TimeBehavior that will serve as a topic producer, and add a time pulse listener to the runtime with the object you just made as the argument, like this:



* + 1. After that, convert the internal topic/egress to external for MQQT. This is done with runtime.bridgeTransmission.
    2. The next step is to subscribe to the MQTT topic/ingress, which was created by mosquitto\_pub above. Call bridgeSubscription on runtime.
    3. Now, listen to internal/topic/ingress and publish localtest, and register a listener on the mqqtBrokerListener. These lines:



# Example Projects

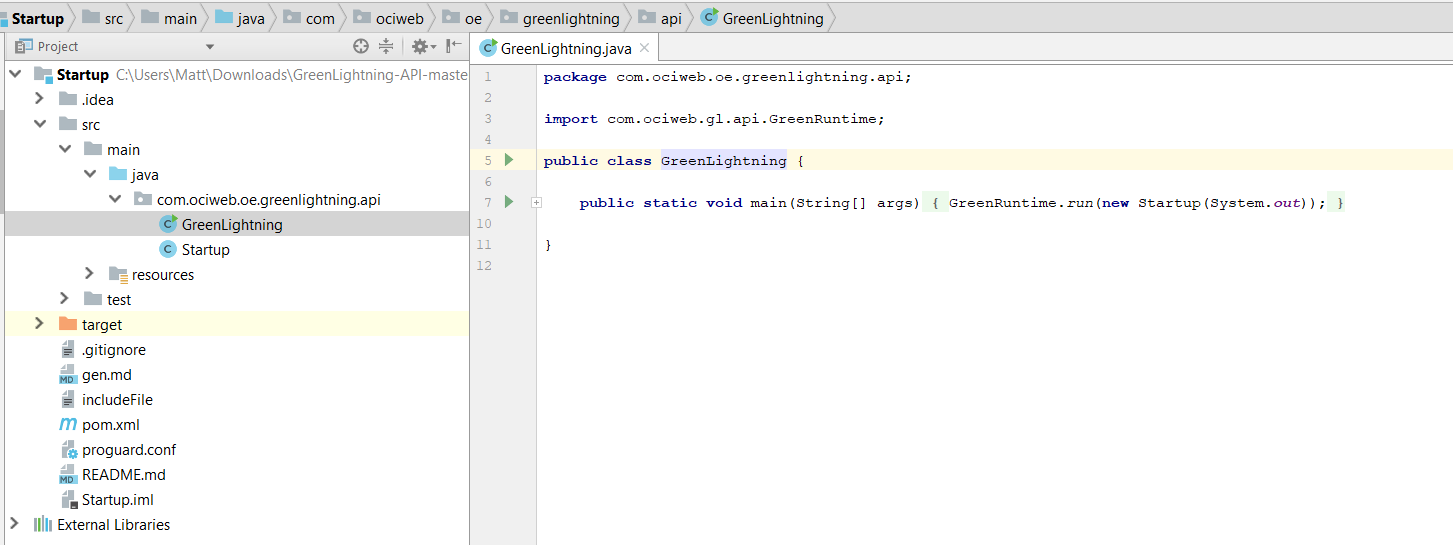
<https://github.com/oci-pronghorn/GreenLightning-API>

Click the green “Clone or download” button and select download as zip to download all the examples, then unzip the folder and it into your IDE.

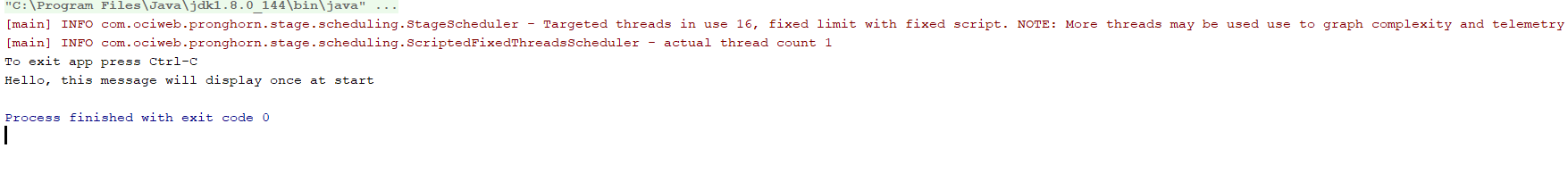
# 1) Startup

<https://github.com/oci-pronghorn/GreenLightning-API/tree/master/Startup/src/main/java/com/ociweb/oe/greenlightning/api>

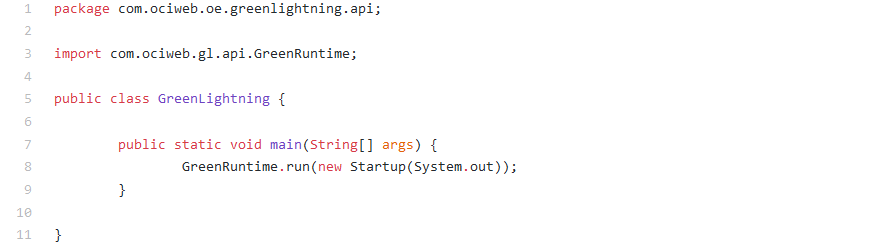
Using IntelliJ, you can run the app by clicking the green arrow to the left of main.



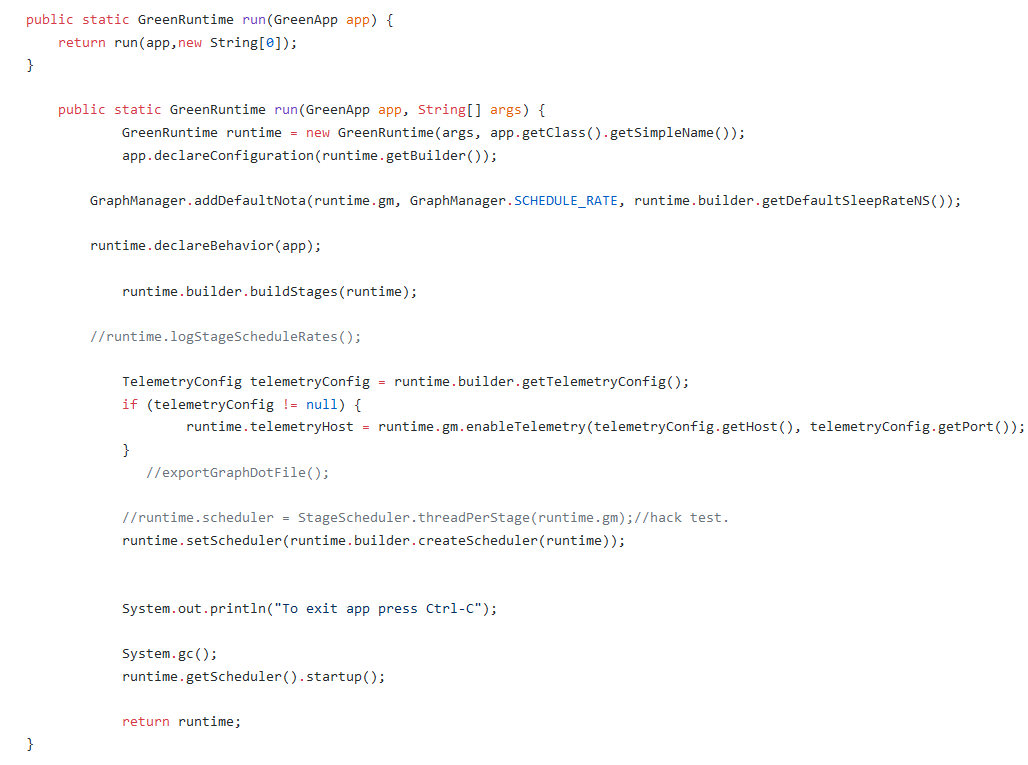
Console output:



GreenLightning.java



GreenLightning programs have main in their GreenLightning.java file. This is where the program begins by using GreenRuntime. GreenRuntime has the method run which starts the program (line 8). Below is the code being used in GreenRuntime’s run method.



Startup.java



When executed, the above code will send the string ```"Hello, this message will display once at start"`` as soon as the program begins running. NOTE: while it was not performed here, if a transducer uses a startup method, then the startup listener of the transducer will execute before the startup method in the behavior class. Also, if multiple transducers use startup a method, do not worry about an order, it will be done automatically.

GreenLightning apps also have a Behavior file. This app demonstrates how a StartupListener works. To use a StartupListener, you need a Startup method (line 15).

The AppendableProxy being used is the console, in other words, it gives the ability to print to the console.

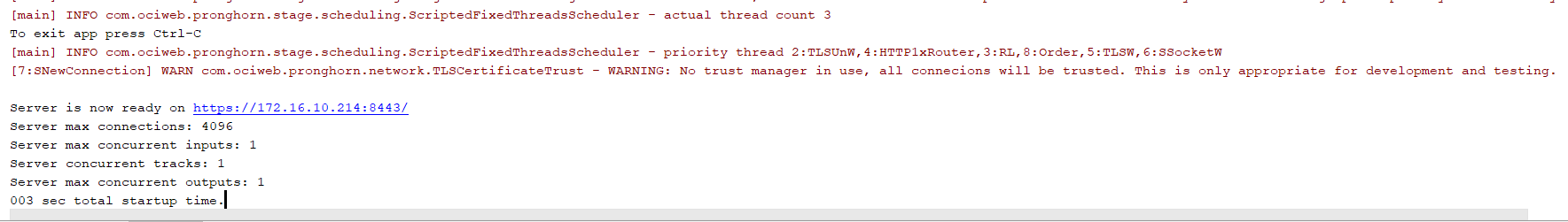
The behavior of the application is declared in the method at line 26. This app calls addStartupListener on the runtime which was declared in the GreenLightning.java file. Once the app is started, because of the StartupListener, it will print the message in the append on line 29.

# 2) Shutdown

<https://github.com/oci-pronghorn/GreenLightning-API/tree/master/Shutdown/src/main/java/com/ociweb/oe/greenlightning/api>

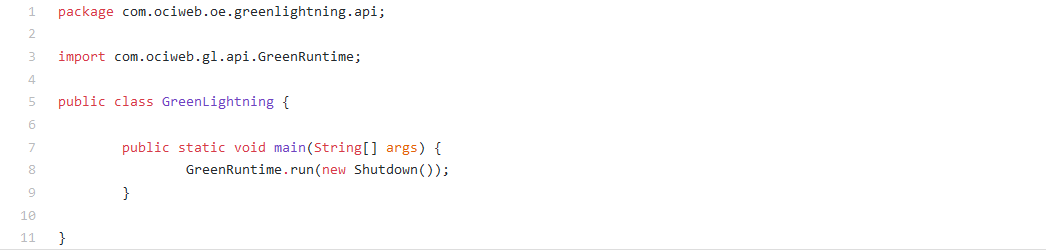
This class is a simple demonstration of how to use the Shutdown(). This demonstration uses allows for shutdown of a device.

Console output:



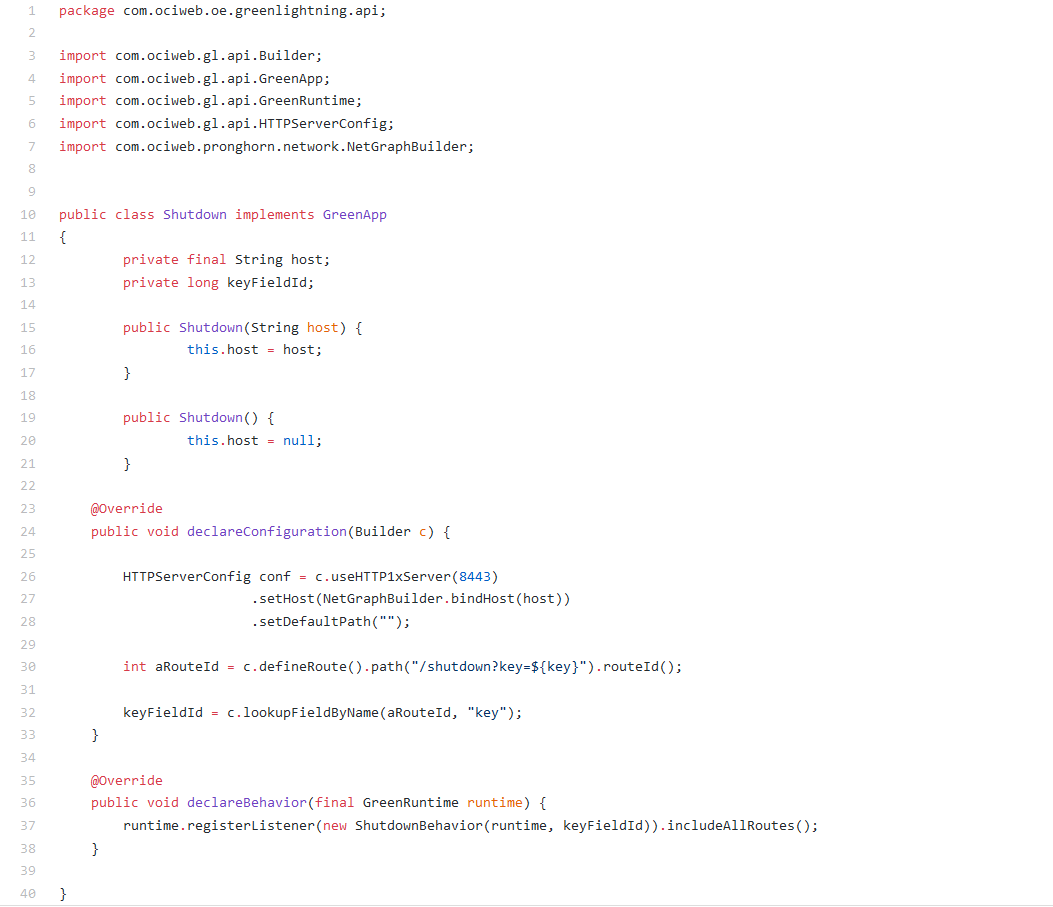
\*The link is expired.

GreenLightning.java



This example’s GreenLightning file is almost the same as the previous, except for the name of the runtime, which is now Shutdown.

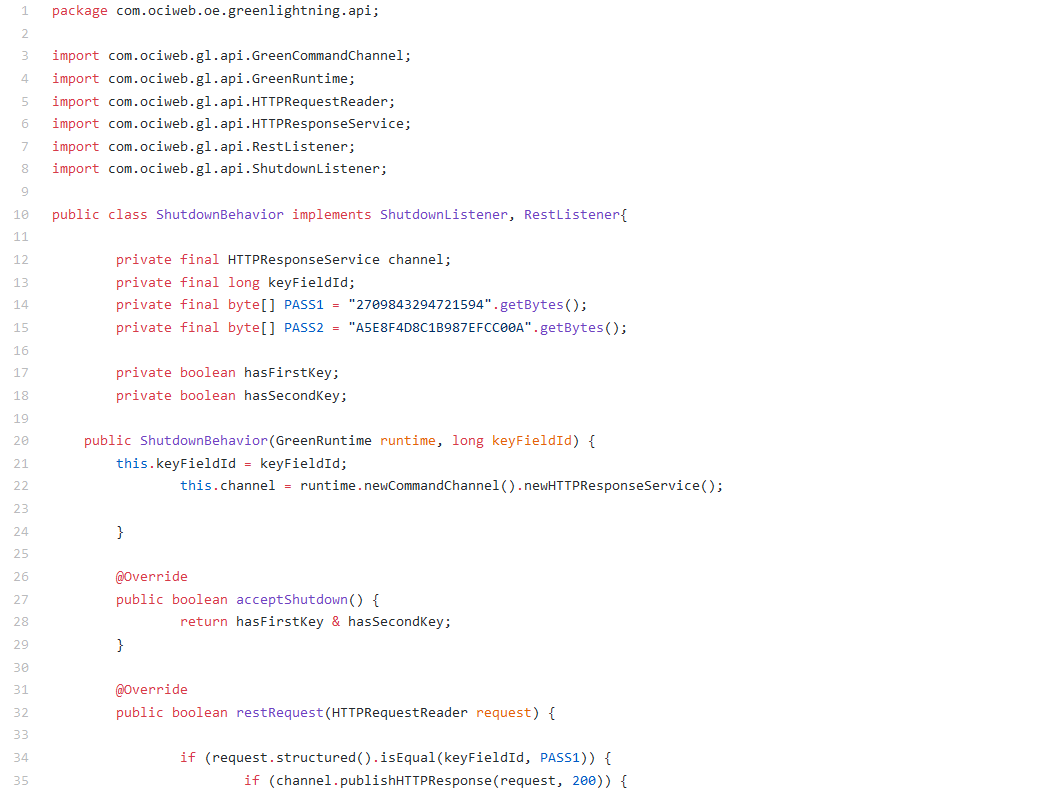
Shutdown.java

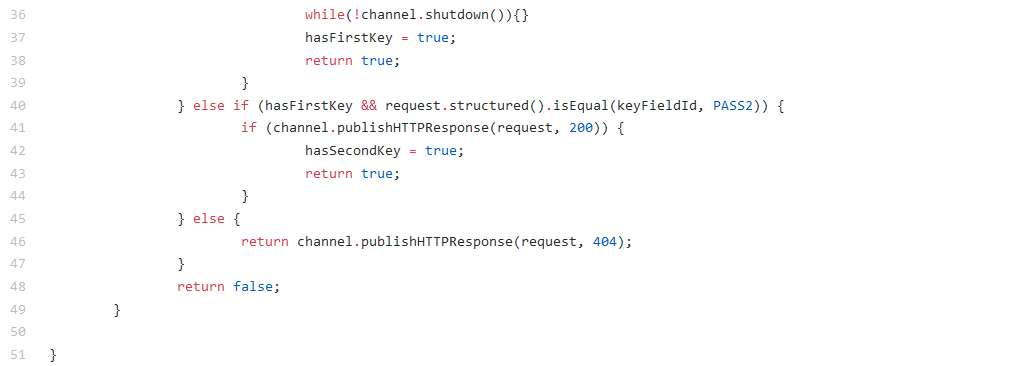


In addition to declareBehavior, this app has declareConfiguration. The declareConfiguration method takes a Builder. A Builder is a GreenLightning class that is used to make objects. The code for Builder can be read here: <https://github.com/oci-pronghorn/GreenLightning/blob/master/src/main/java/com/ociweb/gl/api/Builder.java>.

A configuration for an HTTP server needs a host, path, and server, which can be seen on line 26.

ShutdownBehavior.java



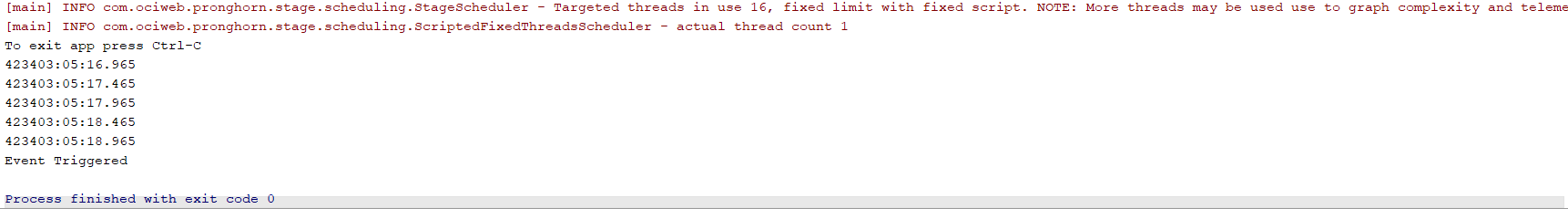


# 3) Timer

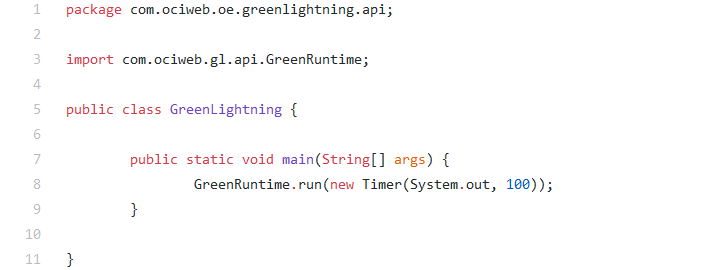
<https://github.com/oci-pronghorn/GreenLightning-API/tree/master/Timer/src/main/java/com/ociweb/oe/greenlightning/api>

The following sketch will demonstrate two simple uses of the addTimeListener() method.

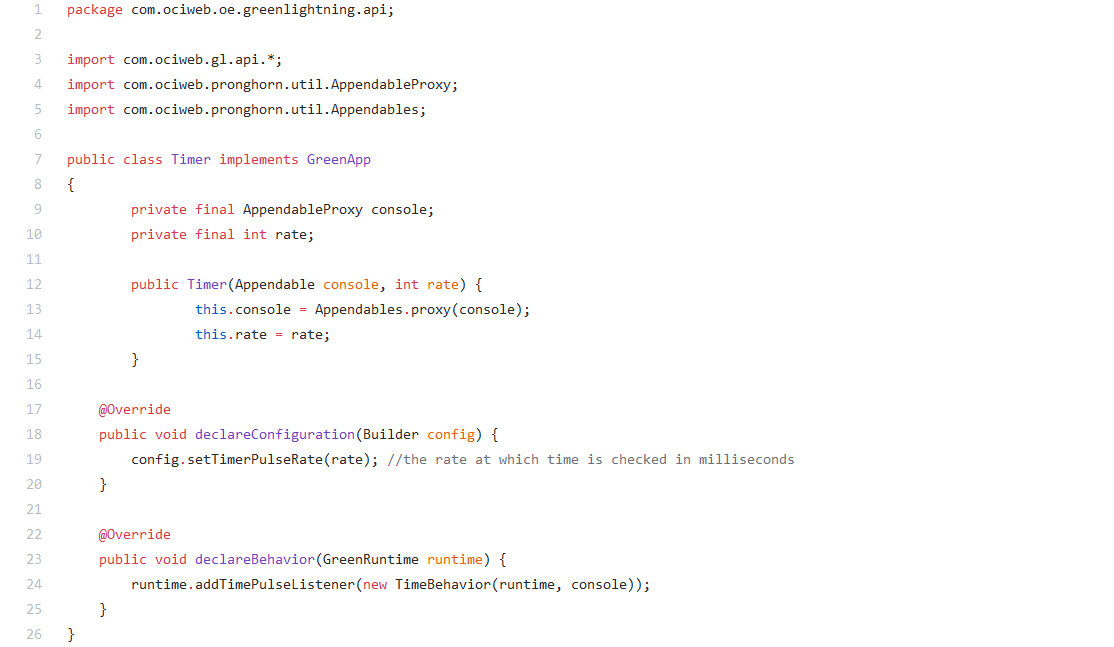
Console output:



GreenLightning.java



Timer.java



TimeBehavior.java



The first demo in this code uses the addTimeListener() method to print out the string "clock" at the top of every minute, regardless of when the program was started. The second demo uses the addTimeListener() method to print out the string "clock" at an interval of one minute since the start of the program. You can change the interval length by changing timeInterval.

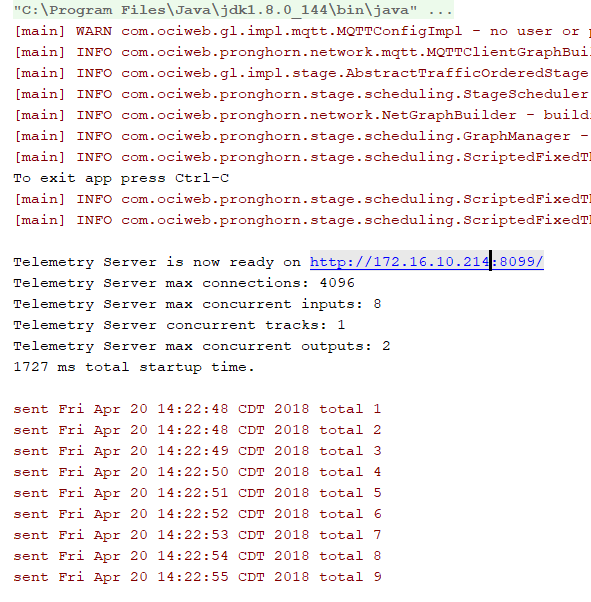
# 4) MQTTClient

<https://github.com/oci-pronghorn/GreenLightning-API/tree/master/MQTTClient/src/main/java/com/ociweb/oe/greenlightning/api>

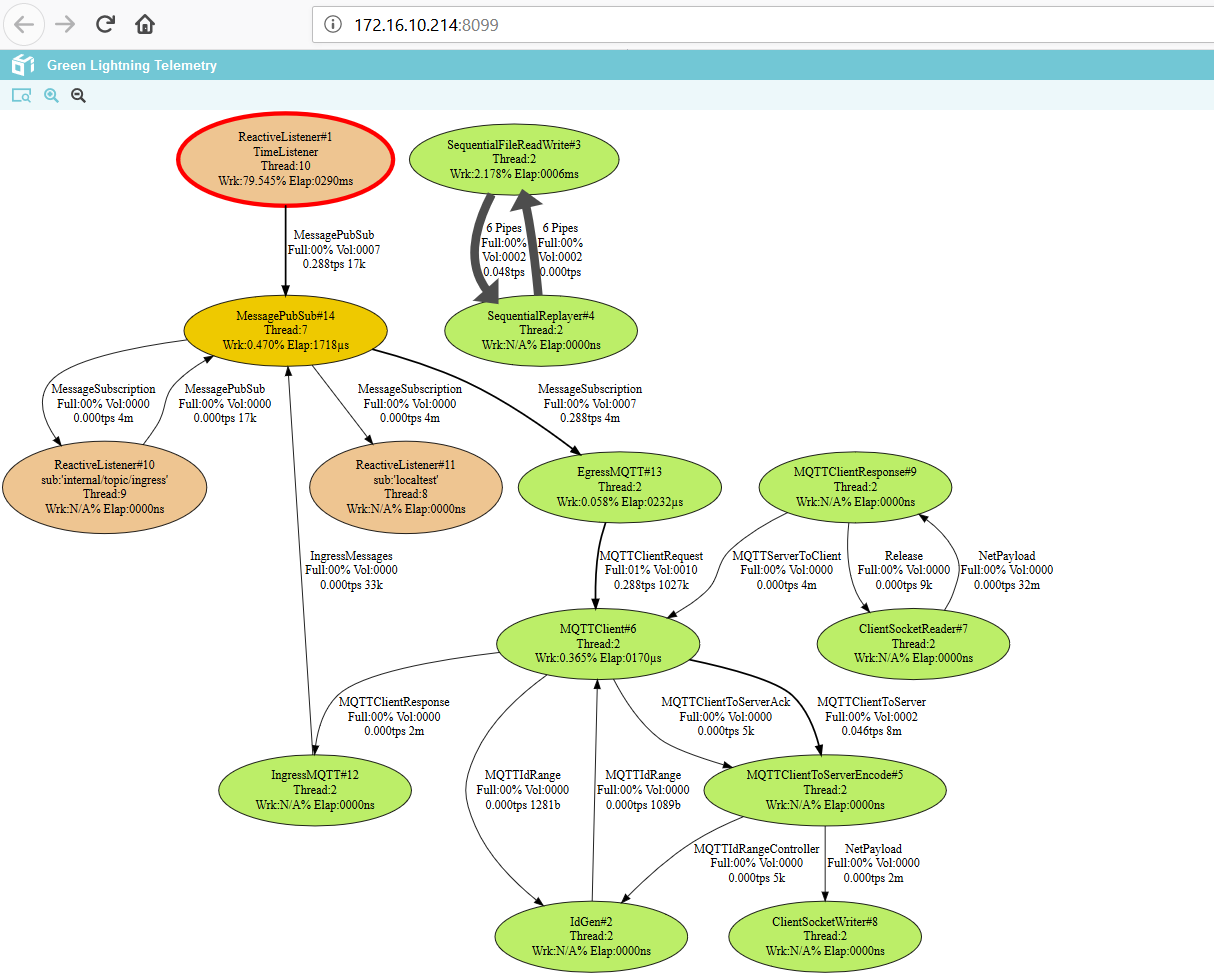
The following sketch will demonstrate a basic demo for using a MQTT.

This class is a simple demonstration of MQTT (Message Queue Telemetry Transport). A lightweight messaging protocol, it was initially designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. This demo uses Mosquitto as a message broker, which means that the messages that are published will go through Mosquitto, which will send them to and subscribers of the topic.

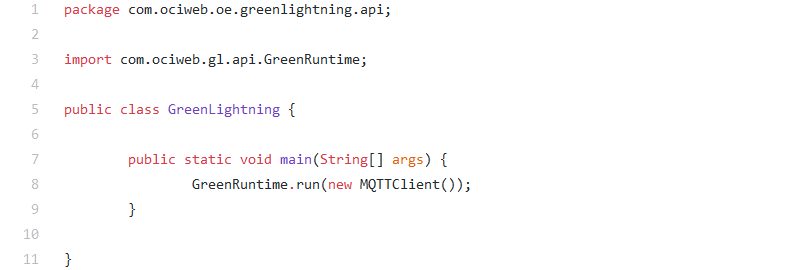
Console output:



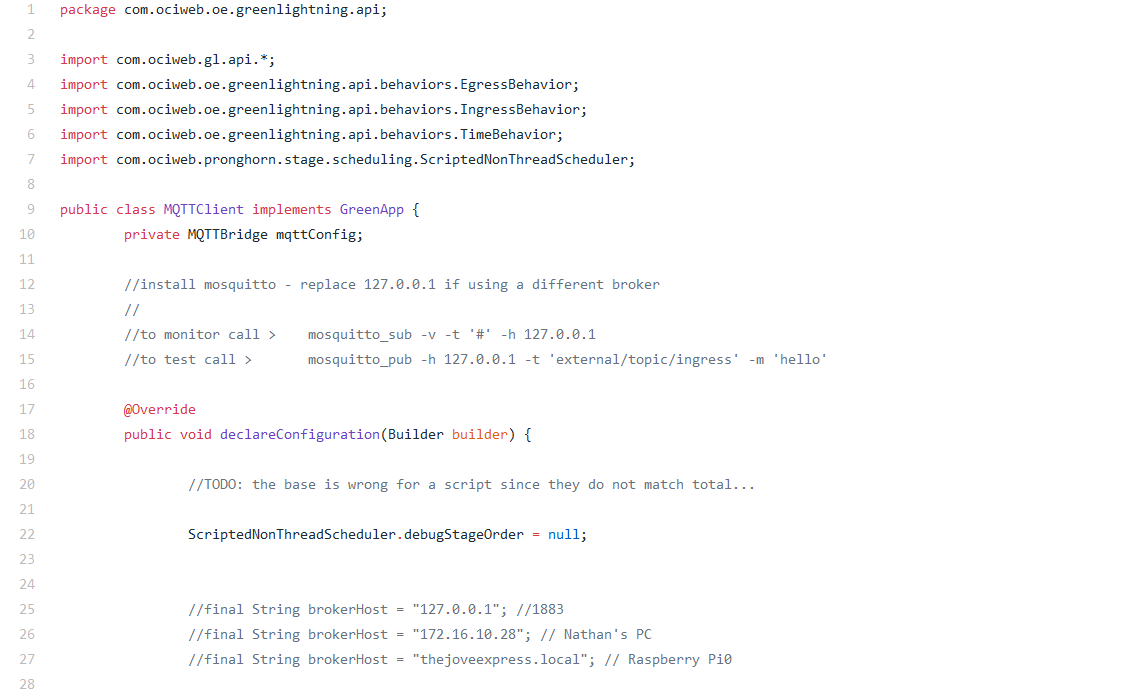
Telemetry:

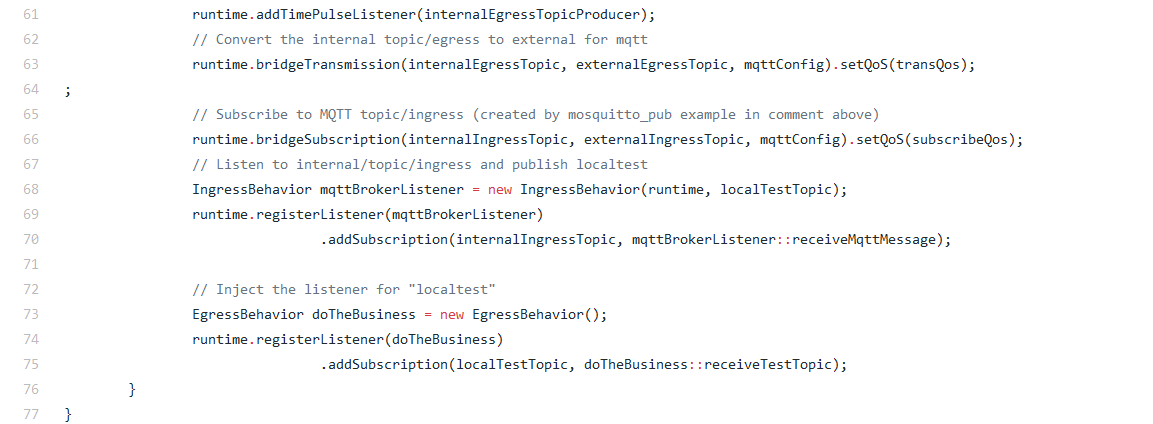


GreenLightning.java



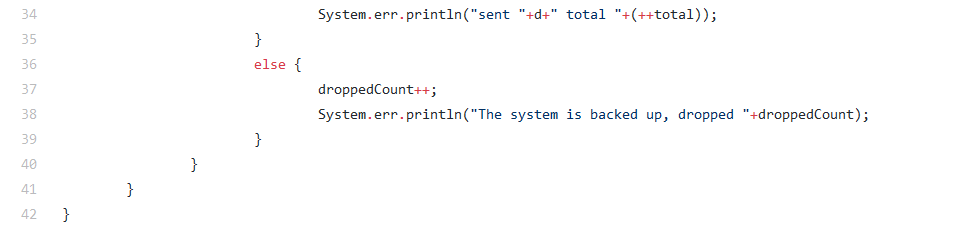
MQTTClient.java



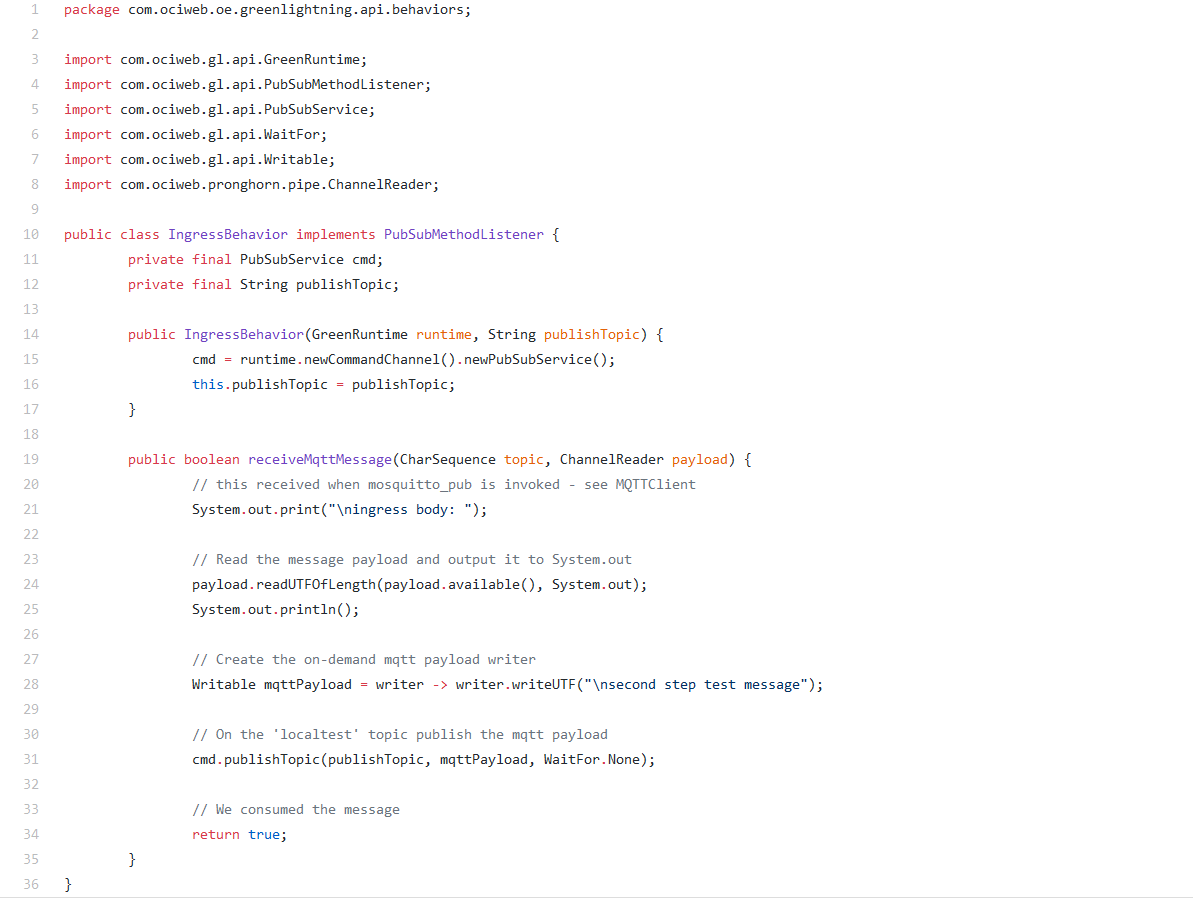


Behaviors

TimeBehavior.java



IngressBehavior.java



EgressBehavior.java

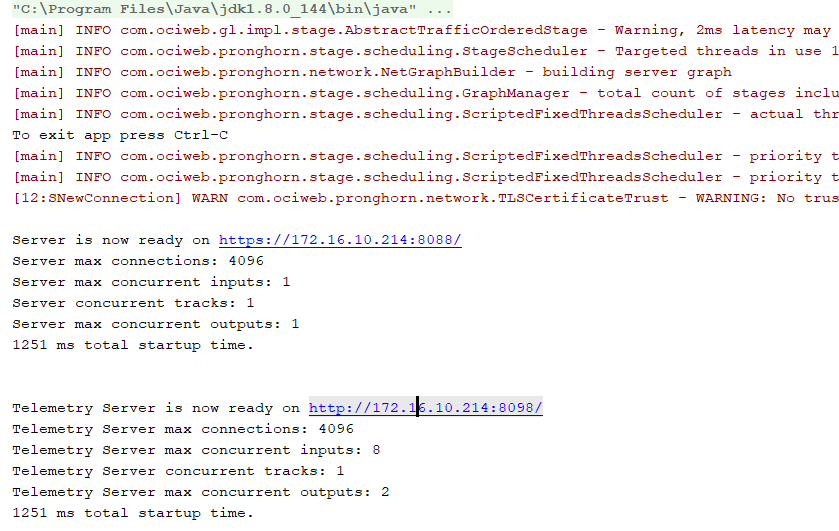


# 5) HTTPServer

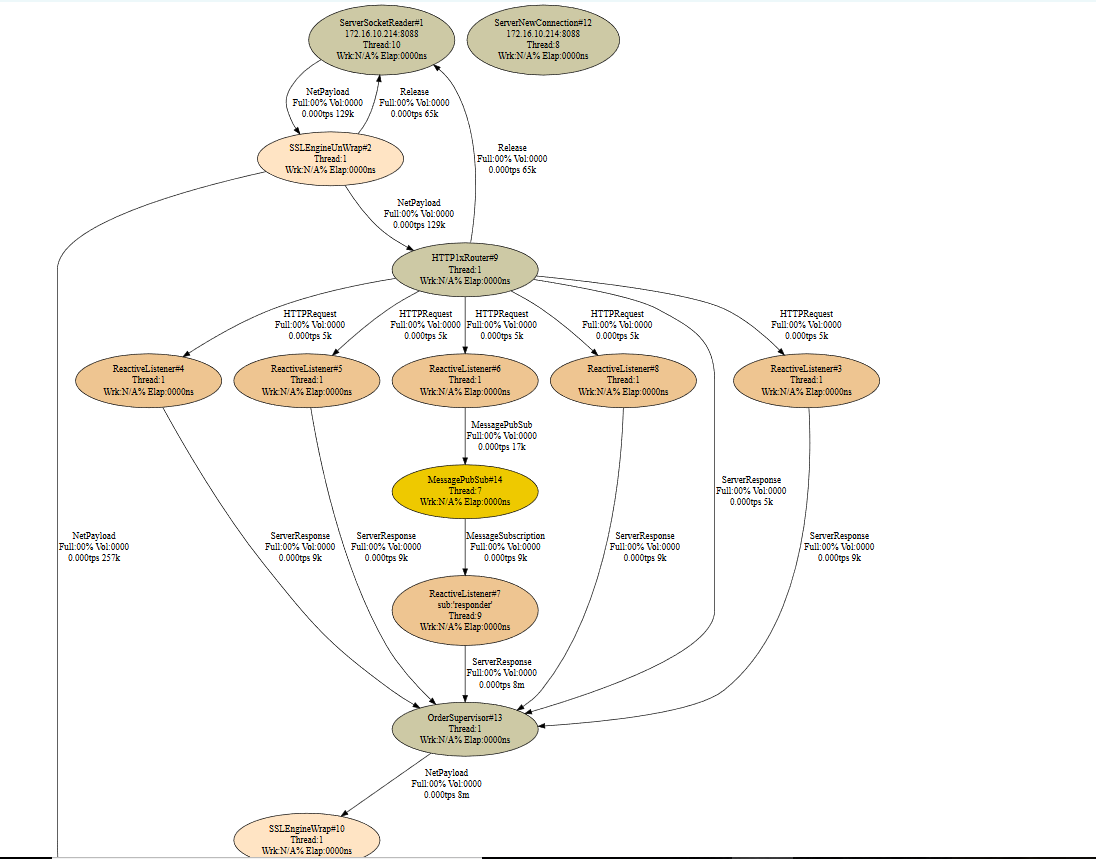
<https://github.com/oci-pronghorn/GreenLightning-API/tree/master/HTTPServer/src/main/java/com/ociweb/oe/greenlightning/api>

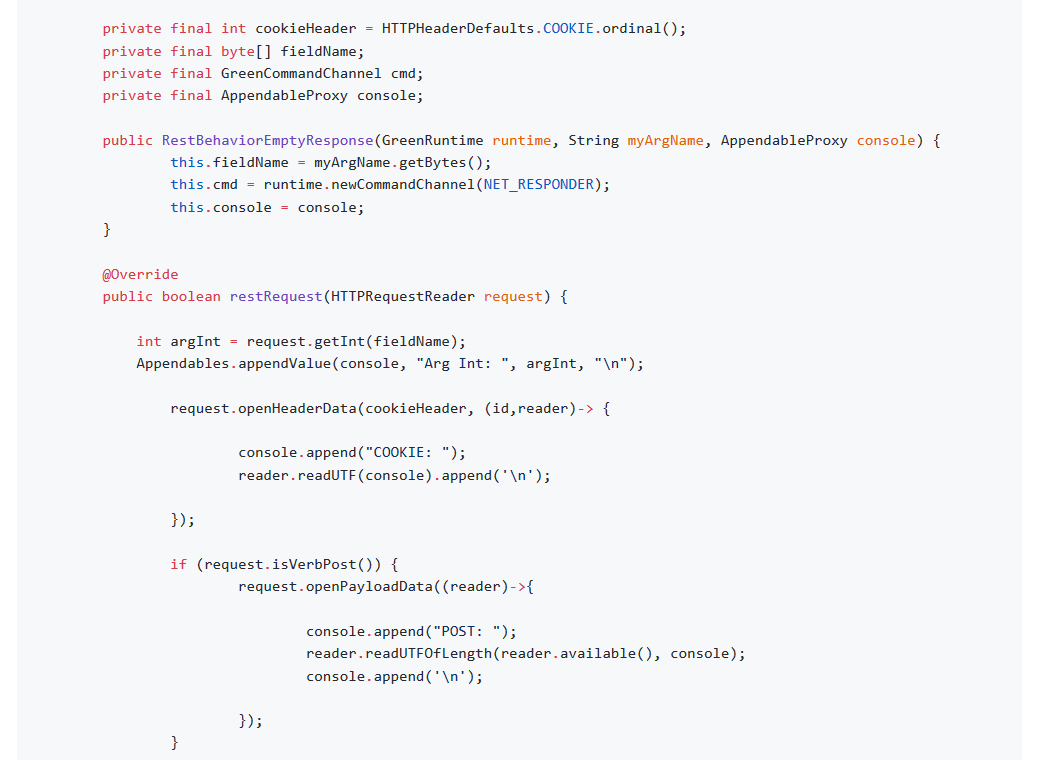
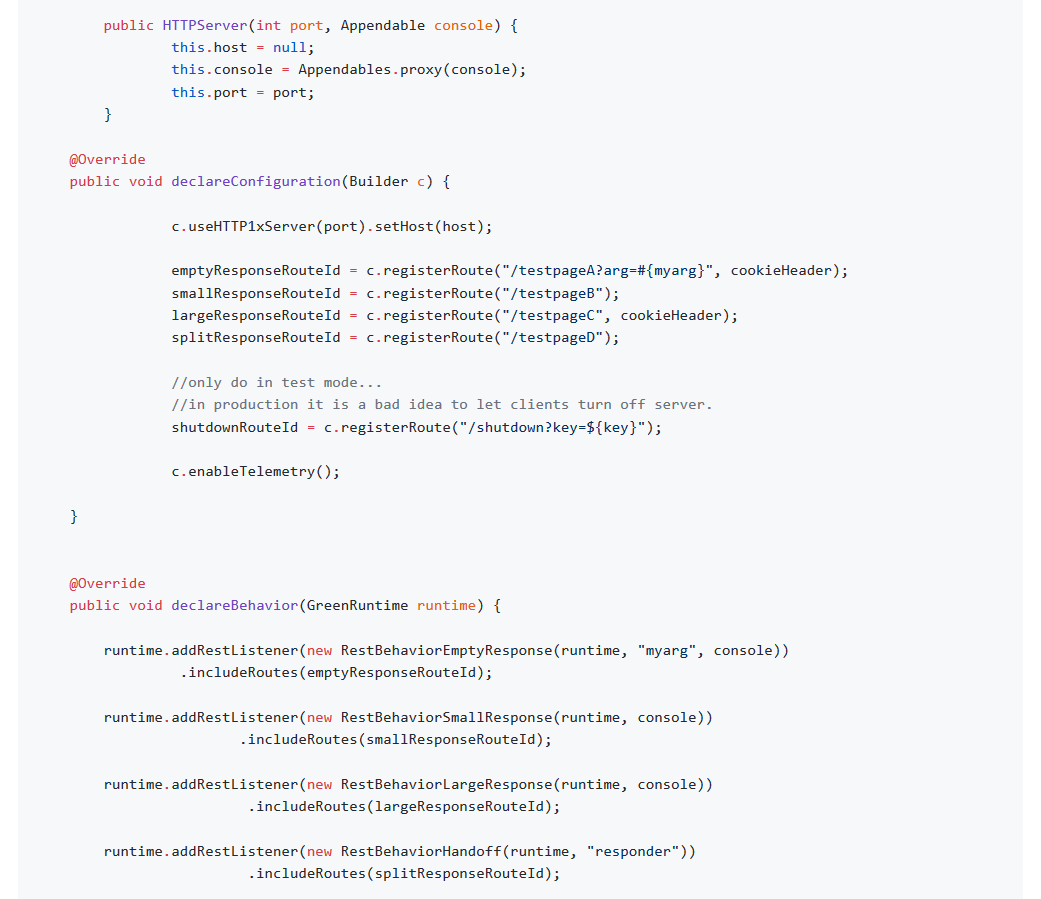
This class is a simple demonstration of HTTPServer. HTTP Server will listen for an HTTP Client to send it a request. It will try to process the request, however, if it is unable to process it at that time, it will be reattempted later.

Console output:



Telemetry:





# 6) HTTPClient

<https://github.com/oci-pronghorn/GreenLightning-API/tree/master/HTTPClient/src/main/java/com/ociweb/oe/greenlightning/api>

