UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño y prototipación de una vivienda inteligente con Arduino y Java

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Serrano Ferriz, David

Tutor: Fons Cors, Joan Josep

2014/2015

Resumen

Este proyecto trata sobre el diseño e implementación de un **sistema domótico** mediante el uso de una placa **Arduino** y un servidor para el control de la vivienda en **Java**. También consta de una **interfaz web** para interactuar con la vivienda accesible vía **REST**. Se ha realizado un prototipo mediante una maqueta de una vivienda unifamiliar.

Palabras clave: Arduino, Java, domótica, IoT, REST.

Abstract

This Project it is about the design and the implementation of an **automated system** along with the use of an **Arduino** plate and a server for the control of the dwelling in **Java**. It also has an **interface web** in order to have the total control of the house with **REST**. It has been created a prototype with a single family home mock-up.

Keywords: Arduino, Java, home automation, IoT, REST.

Tabla de contenidos

1. Introducción

Este documento es la memoria del Trabajo Fin de Grado realizado por David Serrano Ferriz, alumno de la Universidad Politécnica de Valencia.

En este primer capítulo comentaremos la motivación para la realización de este proyecto, y los objetivos que se pretenden alcanzar.

1.1. Motivación

Desde mi punto de vista, pienso que la informática es un campo muy amplio donde cada vez existen más tecnologías con diversos patrones y estándares. Por ello se hace necesario conseguir que todas las nuevas tecnologías se puedan comunicar entre ellas y de este modo lograr un objetivo de forma conjunta. Pienso que es un reto que la mayoría de informáticos tienen que abordar día a día.

A partir de la asignatura de "Integración de aplicaciones" que cursé en cuarto de carrera, fue un gran impulso para la realización de este proyecto.

Al mismo tiempo, el mundo de la domótica siempre me ha gustado porque gracias a su tecnología y evolución conseguimos aumentar nuestro confort de vida en nuestras viviendas y facilitar nuestras tareas cotidianas.

El realizar este trabajo supone el enfrentarse a diferentes problemáticas que tenemos que hacer frente en cada capa y en la comunicación e interacción entre ellas.

Todas estas razones son las que me han impulsado a realizar este Trabajo Fin de Grado.

1.2. Objetivo del proyecto

Este proyecto tiene como objetivo principal diseñar un **sistema domótico** para una vivienda inteligente y hacer un prototipo de ese diseño mediante una maqueta en la que se represente una vivienda unifamiliar.

Se va a optar por ofrecer un sistema domótico con una funcionalidad muy básica. Concretamente se pretende poder controlar la iluminación, ver la temperatura y humedad y un sencillo control de alarma para la detección de intrusos. Para controlar todos estos aspectos se pretende crear una interfaz web.

En el proyecto, se desarrollarán los siguientes módulos:

- **Arduino**: se encargara del control físico de cada componente eléctrico o electrónico.
- **Servidor Java**: se encargará del control de la vivienda domótica, de la comunicación con Arduino y la comunicación con la interfaz web.
- **Interfaz web** que se desarrollará con HTML y CSS.

Una vez finalizado el proyecto se pretender haber conseguido varios aspectos:

- **Aplicar** los diferentes **conocimientos** que se han adquirido a lo largo del Grado en Informática.
- **Experimentar** con nuevas tecnologías, que aunque no se han visto en el Grado, sí que es necesario que con la experiencia y los conocimientos adquiridos nos podamos enfrentar a ellas sin ningún tipo de problema.
- Trabajar con herramientas, técnicas y tecnologías que se utilizar en los **sistemas domóticos actualmente**.

Por último, haciendo referencia los conceptos **WoT** (Web of Things) e **IoT** (Internet of Things) se pretende que ya no solo las personas nos conectemos a Internet, sino en este caso cualquier sensor, actuador o dispositivo esté conectado a Internet.

2. Lenguajes, tecnologías, componentes y herramientas utilizadas

2.1. Lenguajes

Para este proyecto se ha utilizado diferentes lenguajes de programación. A continuación explicamos cada uno de ellos.

2.1.1. Java SE

El lenguaje Java se desarrolló por la empresa Sun Microsystems, la cual se hizo muy famosa por el eslogan "The network is the computer" ("La red es la computadora"). En 1991 desarrolló la implementación de referencia para los compiladores, máquinas virtuales y librerías, siendo en 1995 cuando se publicó por primera vez. En 2009 la compañía fue adquirida por Oracle Corporation.

Las principales características que han hecho de Java un lenguaje tan potente son:

- **Simple**: ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y confusas de éstos.
- **Orientado a objetos**: Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo.
- **Distribuido**: Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.
- **Robusto**: Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.
- **Arquitectura neutral**: Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.
- **Seguro**: las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo.
- **Portable**: Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo.

- **Interpretado**: El intérprete Java (sistema run-time) puede ejecutar directamente el código objeto.
- **Multithreaded**: Al ser multihilo, Java permite muchas actividades simultáneas en un programa.
- **Dinámico**: Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución.

En este caso utilizaremos la versión **Java SE** (Java Standart Edition). Esta es la versión estándar de la plataforma, es decir, la que la mayoría usa para desarrollar sus aplicaciones de escritorio o web. Fue la originalmente desarrollada por Sun.

2.1.2. HTML

HTML (HyperText Markup Language) es un **lenguaje de marcado** para la elaboración de páginas web. Es un estándar a cargo de W3C, organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web, sobre todo en lo referente a su escritura e interpretación.

El lenguaje HTML basa su filosofía de desarrollo en la referenciación. Por ejemplo, para añadir un elemento externo a la página (imagen, vídeo, script, entre otros.), este no se incrusta directamente en el código de la página, sino que se hace una referencia a la ubicación de dicho elemento mediante texto. De este modo, la página web contiene sólo texto mientras que recae en el navegador web (interpretador del código) la tarea de unir todos los elementos y visualizar la página final.

Al ser un estándar, HTML busca ser un lenguaje que permita que cualquier página web escrita en una determinada versión, pueda ser interpretada de la misma forma (estándar) por cualquier navegador web actualizado.

2.1.3. CSS

CSS es el acrónimo de Cascading Style Sheets (hojas de estilo). Describe la **apariencia** y el **formato** de un documento con marcas (como HTML). CSS permite la separación del contenido (HTML) de su presentación.

Maneja elementos como el diseño, colores, fuentes, etc. CSS permite asociar reglas con los elementos que aparecen en un documento HTML. Estas reglas controlan cómo debe ser renderizado el contenido de dichos elementos.

2.1.4. jQuery

jQuery es un **framework JavaScript** que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM (Document Object Model), manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web. Actualmente es la librería JavaScript más utilizada, es gratuita, de código abierto (bajo licencia MIT y GPL v2) y muy ligera.

jQuery, al igual que otras bibliotecas, ofrece una serie de **funcionalidades basadas en JavaScript** que de otra manera requerirían de mucho más código, es decir, con las

funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio.

jQuery nos permite:

- Acceder a elementos en un documento.
- Modificar la apariencia de una web.
- Alterar el contenido de un documento.
- Responder a una interacción del usuario.
- Animar cambios en un documento.
- Recoger información del servidor sin refrescar la página.

2.1.5. PHP

PHP es un lenguaje de programación que se **ejecuta en el lado del servidor**, originalmente diseñado para el desarrollo web de contenido dinámico.

Fue uno de los primeros lenguajes de programación del lado del servidor que se podían incorporar directamente en el documento HTML en lugar de llamar a un archivo externo que procese los datos. El código es interpretado por un servidor web con un módulo de procesador de PHP que genera la página Web resultante.

PHP ha evolucionado por lo que ahora incluye también una interfaz de línea de comandos que puede ser usada en aplicaciones gráficas independientes. Puede ser usado en la mayoría de los servidores web al igual que en casi todos los sistemas operativos y plataformas.

2.1.6. Arduino

Arduino se programa mediante el uso de un **lenguaje propio** basado en el lenguaje de programación de alto nivel Processing que es similar a C++. También es posible utilizar otros lenguajes de programación ya que Arduino usa la transmisión serial de datos, la cuál es soportada por la mayoría de los lenguajes, y para los que no soportan el formato serie de forma nativa, es posible utilizar software intermediario.

2.2. Herramientas de desarrollo utilizadas

Para este proyecto se ha utilizado varias herramientas de desarrollo para facilitar la programación que se ha tenido que llevar a cabo. En concreto hemos usado dos, el IDE de Arduino para programar y cargar el código en la placa Arduino utilizada, y Eclipse IDE para el resto de lenguajes utilizados.

2.2.1. Eclipse IDE

Eclipse es una plataforma de desarrollo compuesto por un conjunto de herramientas de código abierto multiplataforma. Posee un **IDE genérico** por lo que sirve para diferentes lenguajes, siendo Java uno de los más usados.

Las principales características que lo hacen tan popular son las perspectivas, editores y vistas que ofrece para permitirnos trabajar de una forma optima. También ofrece una fácil gestión de proyectos permitiendo la interacción entre ellos. Eclipse también nos proporciona un depurador de código muy potente, de uso fácil e intuitivo.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

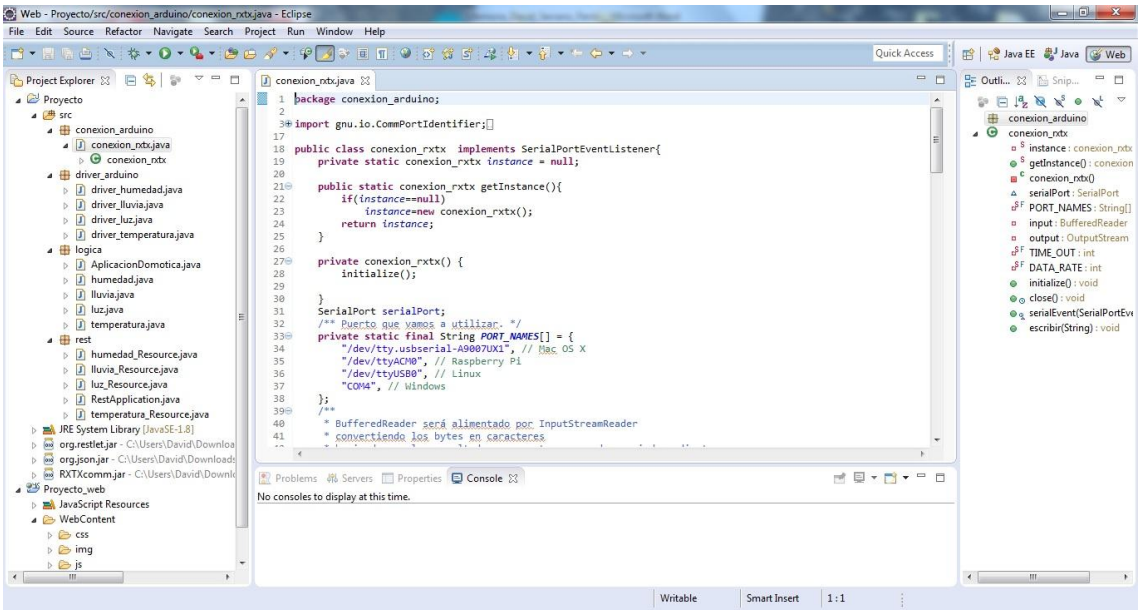

Ilustración 1. Interfaz Eclipse IDE

2.2.2. IDE Arduino

Arduino tiene su propia plataforma de desarrollo en la cual podemos desarrollar el programa. Para ello nos tenemos que descargar de su página oficial la última versión del IDE Arduino. También necesitaremos la última versión de Java Runtime Enviroment (J2RE). Por último instalaremos un controlador FTDI USB para la placa de Arduino que vayamos a utilizar.

La interfaz del IDE de Arduino es muy sencilla. Tiene en la parte superior un menú como la mayoría de programas en el que se pueden configurar diferentes aspectos como el puerto que vamos a usar y el tipo de placa Arduino.

Justo abajo tiene, tiene un botón para compilar el código en busca de fallos (botón con forma de tick) y otro botón para cargar el código en la placa Arduino (botón con forma de fecha hacia la derecha).

En la parte central de la interfaz es donde se desarrolla el código, y por último, en la parte inferior la aplicación nos muestra diferente información cuando compilamos el código o lo cargamos a la placa, desde los posibles fallos hasta el espacio que ocupa en la placa Arduino.

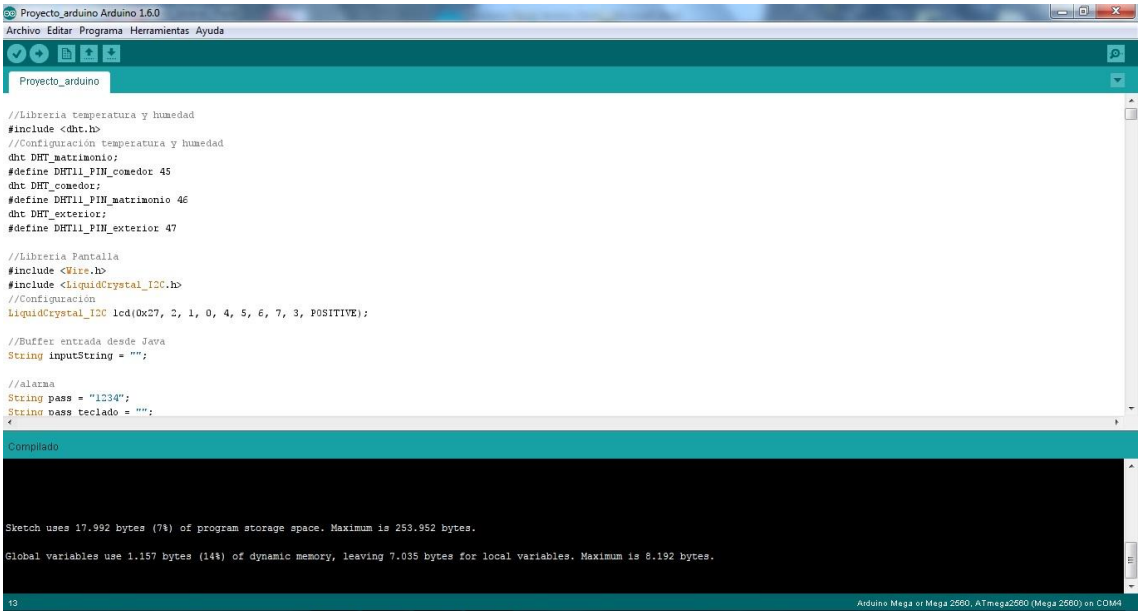

Ilustración 2. Interfaz Arduino IDE

2.3. Tecnologías y formatos de datos utilizados

2.3.1. Ajax

Ajax es una técnica de desarrollo web para crear **aplicaciones interactivas.** Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la **comunicación asíncrona con el servidor** en segunda plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que significa aumentar la interactividad, velocidad y usabilidad de las aplicaciones.

Decimos que Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. JavaScript es el lenguaje interpretado (scripting language) en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.

Proporciona una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores ya que está basado en estándares abiertos como JavaScript y Document Object Model (DOM).

2.3.2. REST

REST (Representational State Transfer) es un **estilo de arquitectura** de software para **sistemas hipermedias distribuidos** tales como la Web. El término fue introducido en la tesis doctoral de Roy Fielding en 2000, quien es uno de los principales autores de la especificación de HTTP.

REST se refiere a una colección de principios para el diseño de arquitecturas en red. Estos principios resumen como los recursos son definidos y diseccionados. El término frecuentemente es utilizado en el sentido de describir a cualquier interfaz que transmite datos específicos de un domino sobre HTTP sin una capa adicional, como hace SOAP. Estos dos significados pueden chocar o incluso solaparse. Es posible diseñar un sistema software de gran tamaño de acuerdo con la arquitectura propuesta por Fielding sin utilizar HTTP o sin interactuar con la Web. Así como también es posible diseñar una simple interfaz XML+HTTP que no sigue los principios REST, y en cambio seguir un modelo RPC (Remote Procedure Call).

Cabe destacar que REST no es un estándar, ya que es tan solo un estilo de arquitectura. Aunque REST no es un estándar, está basado en estándares:

- **HTTP**
- **URL**
- **Representación** de los **recursos**: XML/HTML/GIF/JPEG/...
- **Tipos MIME**: text/xml, text/html, ...

El estilo de arquitectura subyacente a la Web es el modelo REST. Los objetivos de este estilo de arquitectura se listan a continuación:

- **Escalabilidad** de la interacción con los componentes. La Web ha crecido exponencialmente sin degradar su rendimiento. Una prueba de ellos es la variedad de clientes que pueden acceder a través de la Web: estaciones de trabajo, sistemas industriales, dispositivos móviles,...
- **Generalidad de interfaces**. Gracias al protocolo HTTP, cualquier cliente puede interactuar con cualquier servidor HTTP sin ninguna configuración especial. Esto no es del todo cierto para otras alternativas, como SOAP para los Servicios Web.
- Puesta en **funcionamiento independiente**. Este hecho es una realidad que debe tratarse cuando se trabaja en Internet. Los clientes y servidores pueden ser puestas en funcionamiento durante años. Por tanto, los servidores antiguos deben ser capaces de entenderse con clientes actuales y viceversa. Diseñar un protocolo que permita este tipo de características resulta muy complicado. HTTP permite la extensibilidad mediante el uso de las cabeceras, a través de las URIs, a través de la habilidad para crear nuevos métodos y tipos de contenido.
- **Compatibilidad** con componentes intermedios. Los más populares intermediaros son varios tipos de proxys para Web. Algunos de ellos, las caches, se utilizan para mejorar el rendimiento. Otros permiten reforzar las políticas de seguridad: firewalls. Y por último, otro tipo importante de intermediarios, gateway, permiten encapsular sistemas no propiamente Web. Por tanto, la compatibilidad con intermediarios nos permite reducir la latencia de interacción, reforzar la seguridad y encapsular otros sistemas. REST logra satisfacer estos objetivos aplicando cuatro restricciones:
- **Identificación de recursos y manipulación de ellos a través de representaciones**. Esto se consigue mediante el uso de **URIs**. HTTP es un protocolo centrado en URIs. Los recursos son los objetos lógicos a los que se le envían mensajes. Los recursos no pueden ser directamente accedidos o modificados. Más bien se trabaja con representaciones de ellos. Cuando se

utiliza un método PUT para enviar información, se coge como una representación de lo que nos gustaría que el estado del recurso fuera. Internamente el estado del recurso puede ser cualquier cosa desde una base de datos relacional a un fichero de texto.

- **Mensajes autodescriptivos**. REST dicta que los mensajes HTTP deberían ser tan descriptivos como sea posible. Esto hace posible que los intermediarios interpreten los mensajes y ejecuten servicios en nombre del usuario. Uno de los modos que HTTP logra esto es por medio del uso de varios métodos estándares, muchos encabezamientos y un mecanismo de direccionamiento. Por ejemplo, las cachés Web saben que por defecto el comando GET es cacheable (ya que es side-effect-free) en cambio POST no lo es. Además saben como consultar las cabeceras para controlar la caducidad de la información. HTTP es un protocolo sin estado y cuando se utiliza adecuadamente, es posible es posible interpretar cada mensaje sin ningún conocimiento de los mensajes precedentes. Por ejemplo, en vez de logearse del modo que lo hace el protocolo FTP, HTTP envía esta información en cada mensaje.
- **Hipermedia** como un mecanismo del estado de la aplicación. El estado actual de una aplicación Web debería ser capturada en uno o más documentos de hipertexto, residiendo tanto en el cliente como en el servidor. El servidor conoce sobre el estado de sus recursos, aunque no intenta seguirle la pista a las sesiones individuales de los clientes. Esta es la misión del navegador, el sabe como navegar de recurso a recurso, recogiendo información que el necesita o cambiar el estado que él necesita cambiar.

Los métodos HTTP más importantes son **PUT, GET, POST** y **DELETE**. Ellos suelen ser comparados con las operaciones asociadas a la tecnología de base de datos, operaciones CRUD: CREATE, READ, UPDATE, DELETE. Otras analogías pueden también ser hechas como con el concepto de copiar-y-pegar (Copy&Paste). Todas las analogías se representan en la siguiente tabla:

Acción	HTTP	SQL	Copy&Paste	Unix Shell
Create	PUT	Insert	Pegar	>
Read	GET	Select	Copiar	<
Update	POST	Update	Pegar después	>>
Delete	DELETE	Delete	Cortar	Del/rm

Tabla 1. Analogías REST

Las acciones (verbos) CRUD se diseñaron para operar con datos atómicos dentro del contexto de una transacción con la base de datos. REST se diseña alrededor de transferencias atómicas de un estado más complejo, tal que puede ser visto como la transferencia de un documento estructurado de una aplicación a otra.

El protocolo HTTP separa las nociones de un servidor y un navegador. Esto permite a la implementación cada uno variar uno del otro, basándose en el concepto cliente/servidor. Cuando utilizamos REST, HTTP no tiene estado. Cada mensaje contiene toda la información necesaria para comprender la petición cuando se combina el estado en el recurso. Como resultado, ni el cliente ni el servidor necesita mantener ningún estado en la comunicación. Cualquier estado mantenido por el servidor debe ser modelado como un recurso.

2.3.3. JSON

JSON(JavaScript Object Notation) es un **formato de datos** muy ligero basado en un subconjunto de la sintaxis de JavaScript: literales de matrices y objetos. Como usa la sintaxis JavaScript, las definiciones JSON pueden incluirse dentro de archivos JavaScript y acceder a ellas sin ningún análisis adicional como los necesarios con lenguajes basados en XML.

2.4. Componentes Hardware

2.4.1. Arduino Mega 2560

El Arduino Mega 2560 es una **placa electrónica** basada en el Atmega2560. Puede ser alimentado a través de la conexión USB o con una fuente de alimentación externa. La fuente de alimentación se selecciona automáticamente. Tiene 256 KB de memoria flash para almacenar código (de los cuales 8 KB se utiliza para el cargador de arranque), 8 KB de SRAM y 4 KB de EEPROM.

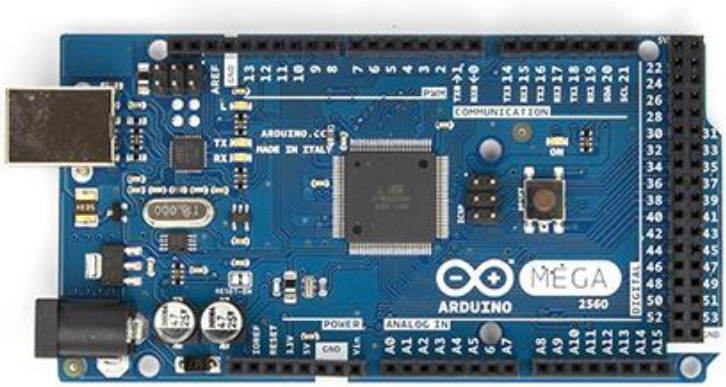

Ilustración 3. Arduino MEGA 2560

Summary

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

Ilustración 4. Características Arduino Mega 2560

2.4.2. Sensores

2.4.2.1. Temperatura y humedad

Para medir la temperatura y la humedad vamos a usar el sensor **DHT11** ya que es uno de los más económicos del mercado y nos ofrece las dos lecturas con un solo sensor. El mayor inconveniente que podemos encontrar es su poca precisión, pero en nuestro proyecto lo que buscamos es conseguir conectar cualquier objeto a Internet. Las características que nos ofrece son las siguientes:

Overview:

Item	Measurement Range	Humidity Accuracy	Temperature Accuracy	Resolution	Package
DHT11	20-90%RH 0-50 ℃	±5%RH	±2℃	1	4 Pin Single Row

Ilustración 5. Características sensor temperatura y humedad DHT11

Ilustración 6. Sensor DHT11

2.4.2.2. Lluvia

La detección de lluvia la vamos a realizar mediante un sensor de lluvia que nos devuelve un valor analógico del cual obtendremos un valor entre 0 y 1023.

Ilustración 7. Sensor lluvia

2.4.2.3. Luminosidad

Para el control de la luminosidad hemos utilizado una LDR (Light-Dependent Resistor). Una **LDR** o fotorresistor es un componente electrónico cuya resistencia varía en función de la luz. Utilizamos una entrada analógica en Arduino para comprobar su valor y ver la luminosidad actual.

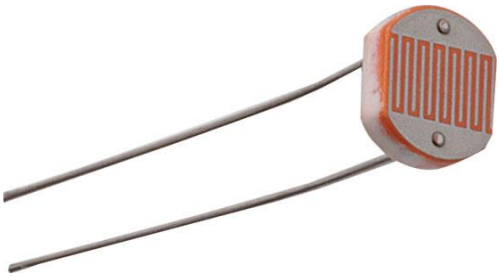

Ilustración 8. Resistencia LDR

2.4.2.4. Detector de presencia

Para la detección de presencia hemos usado el sensor **HC-SR501**. Este sensor nos ofrece dos resistencias variables de calibración. Con la primera (Ch1) podemos establecer el tiempo que se va a mantener activa la salida del sensor entre medio segundo y doscientos segundos. Con la segunda (RL2) nos permite establecer la distancia de detección que puede variar entre 3-7m.

The HC-SR501 Infrared Motion Sensor Features:

- Input Voltage: 4.5V - 20V
- Current Draw: <50µA
- Digital Output: 3.3V (High)
- Digital Output: 0V (Low)
- Working Temperature: -15°C to 70°C
- Delay Time: 0.5 - 200 Seconds
- Sensing Angle: 100° Cone
- Range 5m - 7m
- Dimensions
  - Sensor Lens Diameter: 23mm
  - Length: 24.03mm
  - Width: 32.34mm
  - Height (with lens): 24.66mm
  - Centre screw hole distance: 28mm
  - Screw hole diameter: 2mm (M2)

Ilustración 9. Características del sensor de detección de presencia

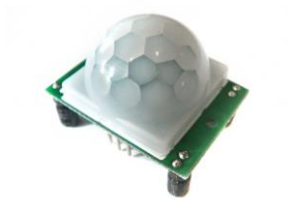

Ilustración 10. Sensor detector de presencia

2.4.3. Actuadores

En este proyecto hay dos actuadores físicos los cuales necesitan la **intervención humana** para interactuar con ellos y para que estos cambien alguno o algunos estados de la vivienda domótica.

2.4.3.1. Teclado

Para la activación y desactivación de la alarma vamos a utilizar un **teclado 4x4** en forma de matriz. Nos permitirá tanto activar la alarma, como introducir el pin y desactivarla. Las características del teclado son las siguientes:

Features:

- A 16 button matrix arranged keypad
- Made of a thin, flexible membrane material with an adhesive backing
- Weight: 7.5 grams
- Keypad dimensions: 77mm x 77mm x 1mm (3" x 3" x 0.035")
- Length of cable + connector: 85mm
- 8-pin 0.1" pitch connector

Ilustración 11. Teclado

Ilustración 12. Teclado 4x4

2.4.3.2. Conmutador

En nuestra maqueta pondremos una serie de conmutadores los cuales nos permitirá manualmente encender y apagar las luces. Alguno de los puntos de iluminación se pueden controlar desde dos o más conmutadores diferentes.

Ilustración 13. Conmutador

2.4.4. Pantalla LCD

Hemos utilizado una pantalla LCD para Arduino de 2 x 16 con caracteres blancos y retroiluminación azul. Cuenta con un potenciómetro para ajustar el contraste de la pantalla.

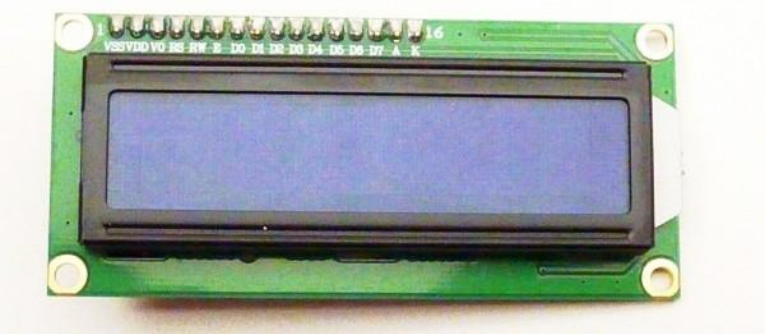

Ilustración 14. Pantalla LCD para Arduino

3. Análisis de una vivienda inteligente con Arduino y Java

En este capítulo hemos realizado un **análisis** de los **requisitos** que tiene que cumplir la vivienda inteligente y los problemas que se tienen que abordar al tratarse de un sistema en tiempo real.

3.1. Descripción del Caso de Estudio

Para el Proyecto Fin de Grado nos piden diseñar un sistema domótico para una vivienda unifamiliar.

El objetivo principal es controlar las luces y observar la temperatura desde cualquier dispositivo. Para ello también es necesario crear una interfaz web para poder interactuar con la vivienda. Se permite añadir más funcionalidades.

Un punto de luz se tiene que poder encender desde la interfaz web o desde un lugar físico. Como en la mayoría de viviendas la iluminación funciona mediante interruptores y conmutadores es necesario que el diseño se haga mediante estos evitando el uso de pulsadores.

Uno de los requisitos es saber la temperatura en el interior de la vivienda y en el exterior. También es necesario tener un sensor para saber si está lloviendo. Esta información deberá estar en la interfaz web.

Para la interacción con la vivienda se tiene que realizar mediante una red local, sin conexión al exterior, por lo que no hará falta de ningún proceso de autentificación para poder controlar la vivienda.

3.2. Análisis del sistema requerido

El principal problema que se aborda en una vivienda inteligente es que se trata de un **sistema en tiempo real**, el cual está en constante cambio y del que no se tiene un control total, ya que el mundo físico no siempre se puede controlar como se desea.

En un sistema de una vivienda inteligente ocurren constantemente eventos procedentes del mundo físico el cual incluye cualquier tipo de dispositivo, o desde el propio sistema para el control del mundo físico mediante los diferentes actuadores. Para almacenar y procesar toda esta información hemos decidido abordar el problema sin utilizar bases de datos ya que se trata de un sistema en tiempo real.

Otro problema abordado es que necesitamos hacer una representación digital del mundo físico. Para ello vamos a hacer uso de **Arduino**, el cual nos proporciona la **abstracción del mundo físico** ya que se encarga de controlar todos los actuadores y sensores.

Aunque en nuestra **vivienda** va a ser totalmente **autónoma**, necesitamos un servidor que se encargue del control. Este servidor se ha realizado mediante el lenguaje de programación Java, el cual almacenara el estado actual de cada elemento de la vivienda domótica.

Por último, tenemos un servidor web que es el encargado de mostrar una **interfaz web** con el estado de la vivienda, y con el cual se puede interactuar para encender o apagar puntos de iluminación.

En definitiva, tendremos tres capas principales: **capa física**, **capa de control**, y **capa de presentación** o interfaz de usuario.

A continuación mostramos un esquema del sistema de la vivienda inteligente.

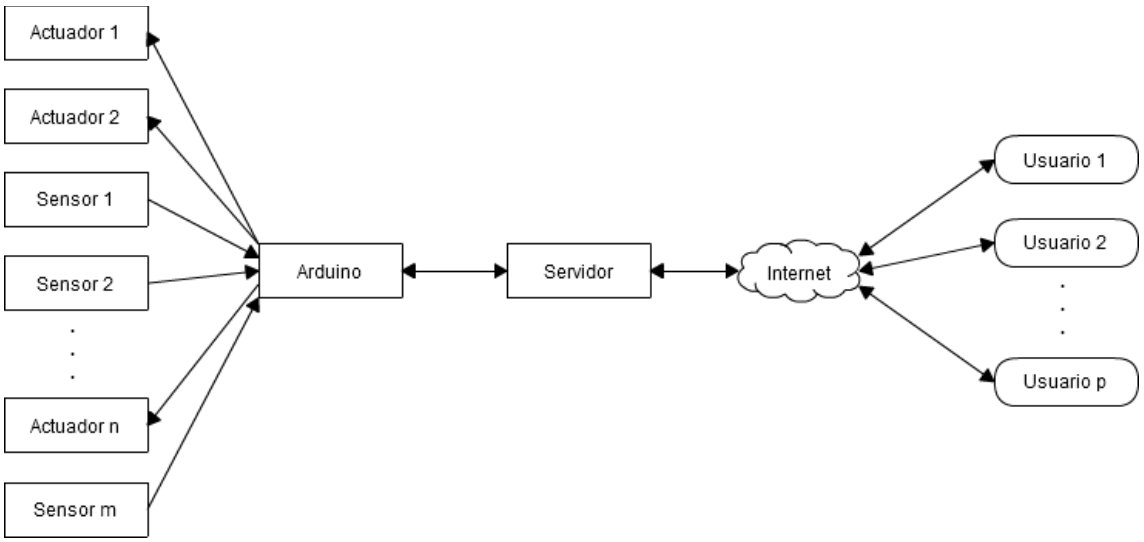

Ilustración 15. Esquema del sistema.

3.3. Capa física

La capa física la componen todos los **dispositivos físicos** de la vivienda, entre los que se encuentra Arduino que es el encargado de controlar directa o indirectamente al resto. Arduino va a ser el encargado de llevar el mundo físico al mundo digital.

El objetivo es el control de la temperatura y la humedad mediante diferentes sensores en cada una de las habitaciones principales. También se controlara en el exterior. Se dispone de unos sensores de movimiento para el control de presencia en diferentes lugares de la vivienda. En el exterior controlaremos la luminosidad y la lluvia mediante dos sensores. La vivienda dispone de interruptores para controlar la iluminación, añadiendo nosotros unos relés para controlarla de forma remota o automática mediante algún automatismo.

La vivienda también dispone de un teclado y una pantalla LCD para el control de una alarma para la detección de intrusos.

3.4. Capa de control

La capa de control está formada por **el servidor Java**. Este procesa cada estado de la vivienda domótica almacenando su estado en variables. En nuestro caso al tratarse de una vivienda autónoma, no se realizara ningún tipo de acción sobre ella. Aun así, esta parte de capa de control es imprescindible para tener una representación digital de la vivienda y posteriormente poder transmitir esa información a la capa de presentación.

Esta capa es la encargada de mandar las instrucciones a Arduino cuando desde la interfaz web se pida apagar o encender una luz.

3.5. Capa de presentación

Para la creación de la **interfaz web** deberíamos hacer en primer lugar un diseño el cual este enfocado dependiendo el uso y las necesidades del usuario final. Para lograr esto habría recolectar información de los usuarios y sus necesidades, definir las personas a partir de la información recolectada. En segundo lugar crear el diseño de la interfaz, y hacer una evaluación del diseño de dicha interfaz, y en caso de no satisfacer los criterios establecidos previamente volver al realizar el diseño hasta que los requisitos se cumplan. Por último se implementaría y se evaluaría para comprobar que cumple todos los criterios de usabilidad y requisitos establecidos, y si no es así volver a diseñar la interfaz y repetir todos los pasos.

En nuestro caso no vamos a seguir todo este proceso. Simplemente vamos a crear una sencilla interfaz con el único objetivo de que tenga la funcionalidad necesaria para poder interactuar con nuestra vivienda domótica, hacienda que todas las tecnologías estudiadas y aplicadas en este proyecto se comuniquen entre ellas y cumplan su objetivo.

Para la implementación interfaz web se ha optado por utilizar **HTML**, **CSS** y **jQuery** porque son lenguajes de cliente, es decir, se ejecutan en el ordenador o dispositivo del cliente, con lo cual no sobrecarga el servidor como ocurre con PHP, el cual es un lenguaje de servidor que se tiene que ejecutar cada vez que el cliente hace una petición pasándole a este la página en HTML.

A pesar de esta decisión de diseño, nos hemos visto obligados a usar **PHP** porque jQuery bloquea las peticiones hacía REST ya que están estas no están en el mismo dominio que el servidor web. A este problema de jQuery se le denomina **CORS**. PHP hace de intermediario y es este el que hace la petición a REST. Hay otra solución la cual consiste en habilitar el CORS en Java, pero para esto en Java hay que hacer tres peticiones en la API Restlet con lo cual entendemos que consume más recursos, ya que con PHP solo supone una llamada más, y no tres.

Un aspecto importante en el diseño de la interfaz web es como se iba a abordar como **mantener actualizada** en la pantalla del un usuario el **estado actual** de la vivienda domótica, principalmente los puntos de iluminación. Había varias posibilidades. La primera era poner un **botón** y que el usuario tuviera que darle para actualizar la página y así tener los valores reales. El inconveniente de esta solución es que se necesitaba la interacción del usuario. La segunda solución y la que se implemento fue que cada

intervalo de tiempo (entre un segundo y dos minutos) dependiendo del objeto se actualizara de forma dinámica en la página web sin que el usuario tuviera que interactuar. La tercera opción era la actualización mediante la tecnología **WebSocket**. Esto permitía crear un canal de comunicación bidireccional mediante un único socket TCP. Aunque esta solución es la mejor, no se implanto porque entendíamos que el proyecto ya era lo suficientemente largo como para añadir una tecnología más.

3.6. Comunicación entre capa física y capa de control

Con respecto a la parte física, la comunicación entre Arduino y el servidor Java se ha realizado mediante el **puerto serie**. Hay otras alternativas como la comunicación por Ethernet o Wifi, pero en nuestro proyecto no será necesaria ya que Arduino está muy próximo al servidor. Si no fuera así, sería recomendable utilizar Ethernet o Wifi en el caso de no poder llevar físicamente un cable de un sitio al otro.

Para la interacción de mensajes entre ambas capas hemos decidido crear un **lenguaje propio** para que ambas capas puedan mandar y recibir mensajes ya que Arduino no proporciona ningún lenguaje predefinido para mandar los estados de los dispositivos físicos.

3.7. Comunicación entre capa de control y capa de presentación

La comunicación entre ambos servidores es una de las problemáticas más difíciles que he tenido que abordar en este proyecto porque existen muchas alternativas para ello, pero ninguna que hayamos visto en profundidad en la carrera y la mayoría ni las conocía o solo había oído hablar de ellas.

En primer lugar hay que definir que lenguajes vamos a usar en ambas partes, en nuestro caso como para el servidor centralizado vamos a usar Java nos quedaba definir que lenguaje vamos a utilizar en el servidor web. Una alternativa es usar JSP (JavaServer Pages) o JSF (JavaServer Faces) ya que ambos ofrecen una API para la comunicación con Java.

En nuestro caso queremos una interfaz en HTML y jQuery, con lo cual tenemos que explorar otras alternativas. Las dos mejores opciones que encontramos fueron Servlets y **REST**. Nos decantamos por la última ya que es una tecnología más actual y se ajustaba al concepto de IoT y WoT.

Hay varias tecnologías actualmente que montan un servicio REST como puede ser Restlet o Jersey. Nosotros nos hemos decantado por la primera ya que nos ofrece una API muy potente y fácil de implementar. La API de **Restlet** se puede descargar de su página oficial.

4. Diseño

A lo largo de este capítulo hemos abordado todos los temas de **diseño**, desde la elaboración de plano de la vivienda unifamiliar sobre el cual haremos el prototipo de la vivienda unifamiliar, hasta el diseño de las diferentes capas de las cuales consta nuestro proyecto.

4.1. Vivienda unifamiliar

El primer paso ha sido realizar un plano de una vivienda unifamiliar. Con la ayuda de **AutoCAD** hicimos un diseño sencillo en el que teníamos un comedor-cocina, una entrada, dos baños y una habitación de matrimonio, el cual mostramos a continuación.

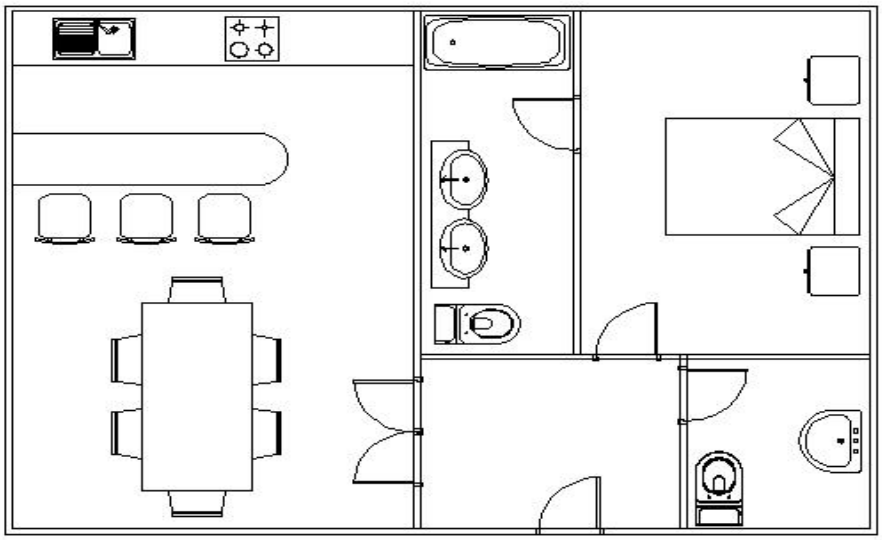

Ilustración 16. Plano vivienda unifamiliar

4.1.1. Sensores y actuadores

En la vivienda hay tres sensores de **presencia** en el comedor, entrada y habitación de matrimonio. Tenemos tres sensores de **temperatura** y **humedad** situados en el comedor, en la habitación de matrimonio y en el exterior. Por último, tenemos también en el exterior un sensor de **luminosidad** y otro de **lluvia**.

Cada habitación tiene un **punto de iluminación** controlado por interruptores. Seis **relés** conectados al sistema de iluminación permiten que Arduino tenga el control de cada punto.

Para terminar, tenemos una **pantalla LCD** y un **teclado** para el control de la alarma junto con los sensores de presencia antes nombrados.

4.2. Arduino

La capa física va a ser controlada por **Arduino**, el cual nos permite convertir el mundo **físico en digital**. Cada sensor o actuador tiene su propia librería en Arduino que se encarga del control de estos, permitiéndonos a nosotros centrarnos en recolectar la información digital que nos proporciona y mandar dicha información cuando sea necesario a la capa de control, en nuestro caso el servidor Java.

A continuación mostramos el diagrama de flujo el cual tiene que procesar Arduino en su bucle principal, es decir, de forma continua.

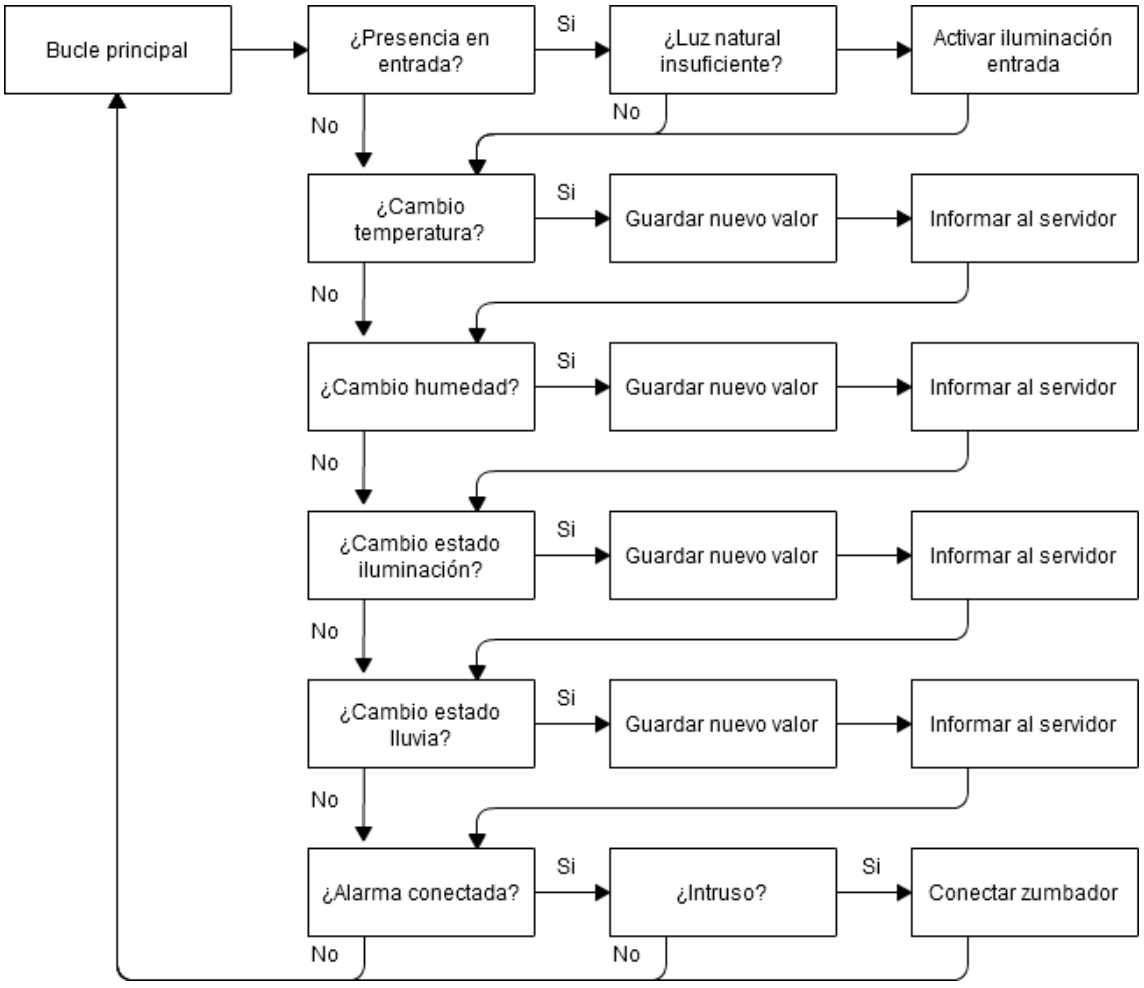

Ilustración 17. Diagrama de flujo. Bucle principal Arduino.

Arduino también se encarga de **procesar** todas las **instrucciones** procedentes del servidor Java mediante el túnel que hemos creado a través del puerto serie y el lenguaje que nos hemos definido para permitir esta comunicación.

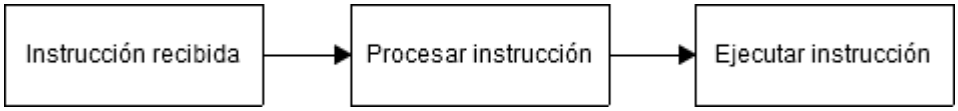

Ilustración 18. Diagrama de flujo. Arduino recibe instrucción.

4.2.1. Distribución de pines

A continuación adjuntamos una tabla donde especificamos cada pin de la placa Arduino para que se ha utilizando, si es digital o analógico, y si hace función de entrada o de salida.

Nº Pin	Función	Digital/Analógico	Salida/Entrada
A0	Sensor luminosidad	Analógico	Entrada
A15	Sensor lluvia	Analógico	Entrada
22	Teclado Fila 0	Digital	Entrada
23	Teclado Fila 1	Digital	Entrada
24	Teclado Fila 2	Digital	Entrada
25	Teclado Fila 3	Digital	Entrada
26	Teclado Columna 0	Digital	Entrada
27	Teclado Columna 1	Digital	Entrada
28	Teclado Columna 2	Digital	Entrada
29	Teclado Columna 3	Digital	Entrada
30	Luz matrimonio	Digital	Entrada
31	Luz aseo matrimonio	Digital	Entrada
32	Luz aseo	Digital	Entrada
33	Luz entrada	Digital	Entrada
34	Luz cocina	Digital	Entrada
35	Luz comedor	Digital	Entrada
36	Luz matrimonio	Digital	Salida
37	Luz aseo matrimonio	Digital	Salida
38	Luz aseo	Digital	Salida
39	Luz entrada	Digital	Salida
40	Luz cocina	Digital	Salida
41	Luz comedor	Digital	Salida
42	Sensor presencia matrimonio	Digital	Entrada
43	Sensor presencia entrada	Digital	Entrada
44	Sensor presencia comedor	Digital	Entrada
45	Sensor DHT11 comedor	Digital	Entrada
46	Sensor DHT11 matrimonio	Digital	Entrada
47	Sensor DHT11 exterior	Digital	Entrada
48	Buzzer alarma	Digital	Salida

SDA20	Pantalla LCD	-	Salida
SCL21	Pantalla LCD	-	Salida

Tabla 2. Distribución pines Arduino

4.3. Comunicación Arduino-Java

La comunicación entre Arduino y Java se ha realizado mediante un túnel por el puerto serie. Para ello hemos creado un **lenguaje propio** que nos permita transmitir la información entre ellos.

Desde Arduino a Java se mandara información del estado de cada elemento, con lo cual utilizaremos el formato **JSON**, indicando el dispositivo y el estado o valor.

{[dispositivo],[estado/valor]}

Un ejemplo de comunicación es así:

{luz_comedor,true}
{tem_comedor,25.0}

A continuación mostramos una tabla con todos los dispositivos y el tipo variable que tienen como representación digital.

Dispositivo	Tipo
luz_comedor	Boolean
luz_cocina	Boolean
luz_matrimonio	Boolean
luz_aseomatrimonio	Boolean
luz_aseo	Boolean
luz_entrada	Boolean
tem_comedor	Double
tem_matrimonio	Double
tem_exterior	Double
hum_comedor	Double
hum_matrimonio	Double
hum_ exterior	Double
lluvia	Boolean

Tabla 3. Formato de datos Arduino -> Java

Para la comunicación de Java hacía Arduino mandaremos **instrucciones**. El fin de cada una de ellas se indica con un punto y coma. El formato es el siguiente:

[instrucción],[parámetro];

Un ejemplo de comunicación es así:

encender_luz,comedor;
estado_temp,comedor;
estado_lluvia;

La siguiente tabla muestra todas las instrucciones y los posibles parámetros que puede recibir. Los parámetros están separados por comas en el caso de que existan diferentes parámetros para una misma instrucción.

Instrucción	Parámetros
encender_luz	comedor, cocina, aseo, aseomatrimonio, matrimonio, entrada
apagar_luz	comedor, cocina, aseo, aseomatrimonio, matrimonio, entrada
estado_temp	comedor, matrimonio, exterior
estado_hum	comedor, matrimonio, exterior
estado_lluvia	-
estado_luz	comedor, cocina, aseo, aseomatrimonio, matrimonio, entrada

Tabla 4. Instrucciones Java->Arduino

4.4. Servidor Java

El diseño del servidor Java lo hemos abordado dividiéndolo en tres paquetes. El primero de ellos es el **túnel** hacia Arduino. El segundo es el **driver** que hemos creado para el control de mensajes con Arduino. Y por último, la capa de **lógica** de negocio.

El primer paquete se encarga de crear el túnel con Arduino mediante el puerto serie. Hemos creado un buffer de entrada y otro de salida para la comunicación.

El segundo paquete está formado por los drivers que se encargan de mandar las instrucciones mediante el lenguaje que hemos creado, y procesar los JSON recibidos desde Arduino. Hemos creado cuatro drivers para controlar la luz, humedad, temperatura y lluvia respectivamente. Para que cada driver sea único hemos utilizado el patrón singleton.

Por último, en la capa de lógica de negocio hemos creado cuatro clases correspondientes a la luz, humedad, temperatura y lluvia las cuales tendrán el valor actual de variable y los correspondientes métodos para cambiar y controlar cual estado de la vivienda domótica. Para la creación de estas clases también se ha seguido un patrón singleton.

En esta capa también tendremos la clase que contiene el *main* la cual es la encargada de inicializar el resto de clases.

En la siguiente imagen podemos observar el diagrama de clases del servidor Java.

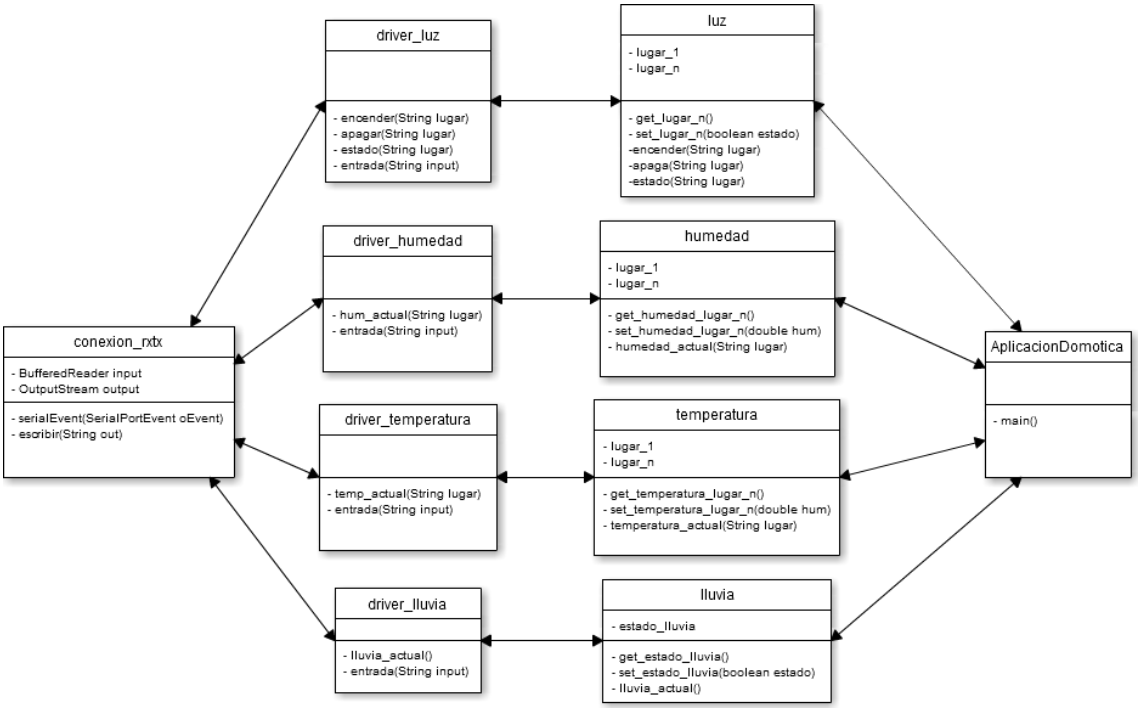

Ilustración 19. Diagrama de clases

4.5. REST

Lo primero que se ha hecho es una tabla que incluya todas las peticiones REST que posteriormente implementaremos, indicando el método de petición (GET o POST), los parámetros que se envían y la respuesta esperada.

URL del recurso	Método	Parámetros	Respuesta
dispositivo/luces	GET	-	Estados iluminación en formato JSON
dispositivo/luz/cocina	POST	Nuevo estado luz (true/false)	Ok
dispositivo/luz/comedor	POST	Nuevo estado luz (true/false)	Ok
dispositivo/luz/aseomatrimonio	POST	Nuevo estado luz (true/false)	Ok
dispositivo/luz/matrimonio	POST	Nuevo estado luz (true/false)	Ok
dispositivo/luz/aseo	POST	Nuevo estado luz (true/false)	Ok

dispositivo/luz/entrada	POST	Nuevo estado luz (true/false)	Ok
dispositivo/humedades	GET	-	Estados humedad en formato JSON
dispositivo/temperaturas	GET	-	Estados temperatura en formato JSON

Tabla 5. URLs de recursos REST

Para la **implementación** del servicio **REST** lo hemos realizado sobre el **servidor Java**, al cual vamos a incorporar un paquete con cinco clases. Tenemos cuatro clases para los recursos de luz, lluvia, temperatura y humedad, y una última clase donde creamos la aplicación REST la cual se encarga de procesar cada llamada y ejecutar cada recurso.

La inicialización de la aplicación REST se efectúa desde la clase *AplicacionDomotica*. A continuación mostramos como quedaría el diagrama de clases perteneciente a la aplicación REST.

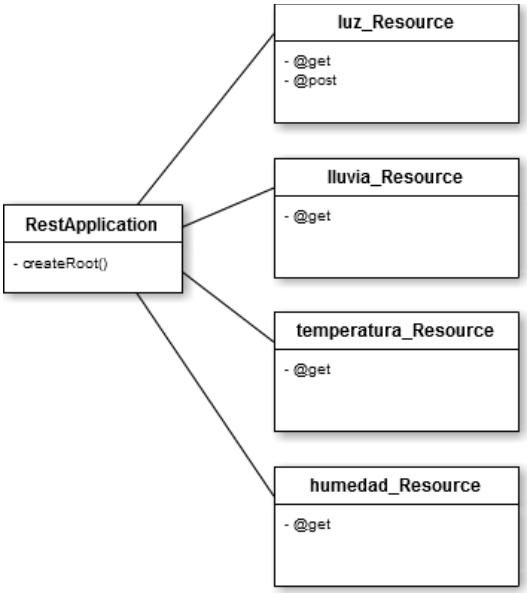

Ilustración 20. Diagrama de clases REST

4.6. Interfaz web

Para la interfaz web, como hemos comentado en el apartado de análisis, vamos a realizar un diseño muy básico, en el que se muestre la temperatura, humedad, lluvia y estado de la iluminación. La única interacción que va a existir es apagar y encender cada punto de iluminación.

Solo vamos a tener una pantalla en el que se muestre toda la información. La dividimos en dos columnas. La columna de la derecha mostramos la fecha actual, la temperatura y humedad exterior, y el estado de la lluvia. En la columna de la izquierda tenemos el plano de la vivienda en el que pondremos bombillas para representar los puntos de

iluminación a las cuales se les puede pulsar para cambiar el estado. También indicaremos la temperatura y humedad del comedor-cocina y de la habitación de matrimonio.

Con respecto a la **parte dinámica** la cual gestionara jQuery la podemos diferenciar en dos partes. La primera de ellas es cuando el usuario presione una bombilla para apagarla o encenderla. La segunda parte trata sobre la actualización automática que hemos realizado del estado de la vivienda, es decir, del estado de la iluminación, temperatura, humedad y lluvia.

Cuando un usuario pulse una bombilla se activa en jQuery el método correspondiente el cual realiza una petición REST con el nuevo estado. Posteriormente cambiamos el estado de la bombilla. Es importante saber que la petición REST tiene una respuesta de "OK" si la petición se ha realizado correctamente, es decir, si le ha llegado la petición al servidor. Esto no implica que la luz cambie de estado como se deseaba. Nosotros asumimos que ha funcionado correctamente, ya que posteriormente lo comprobamos como decidimos en la fase de análisis.

El diagrama de flujo correspondiente al pulsar una bombilla es el siguiente:

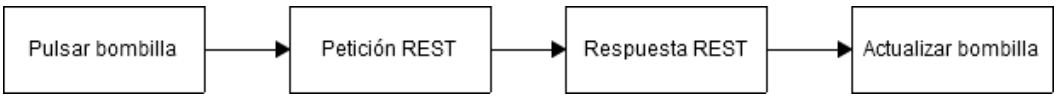

Ilustración 21. Diagrama de flujo. Pulsar bombilla

Como comentamos en la fase de análisis, para mantener la interfaz web actualizada hemos optado por actualizarla cada cierto intervalo de tiempo de forma automática. El flujo que sigue jQuery es el siguiente:

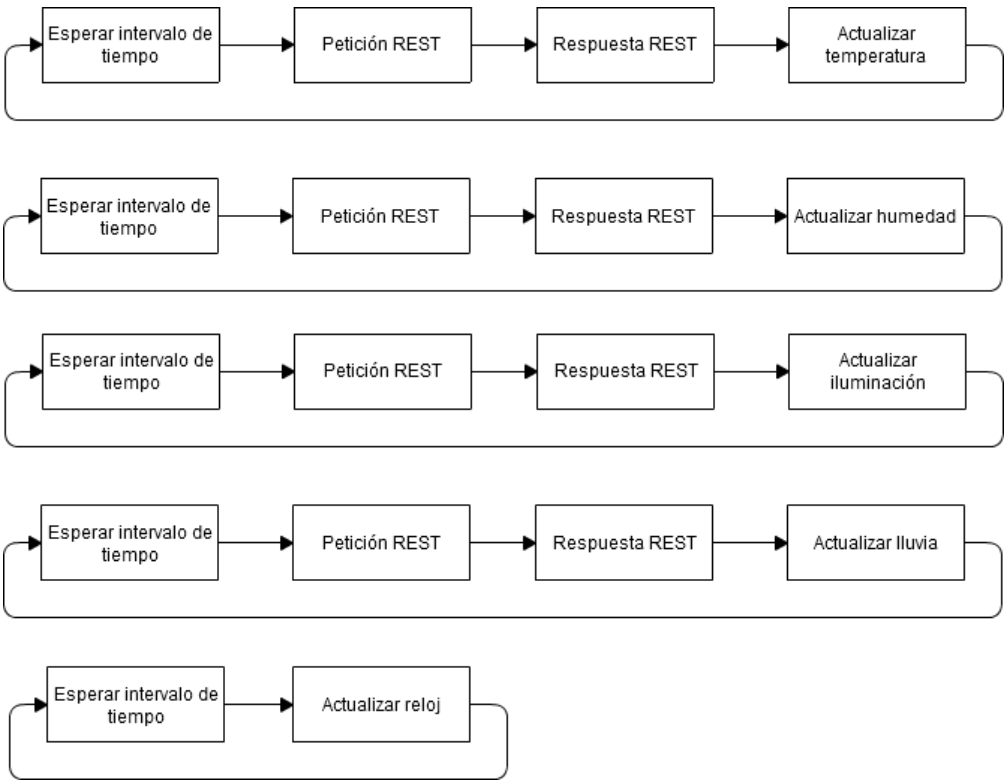

Ilustración 22. Diagrama de flujo. Actualización interfaz jQuery

Cuando la página se carga por primera vez, todos los elementos se encuentran actualizados. A partir de ese momento, cada elemento se actualiza con diferentes intervalos dependiendo de la probabilidad que tienen de un cambio de valor en el tiempo, es decir, una bombilla puede cambiar el estado cada pocos segundos porque por ejemplo vas recorriendo la casa en un momento determinado, pero la temperatura o humedad no va a variar en un par de segundos por normal general. A continuación mostramos una tabla con los tiempos de actualización.

Elemento a actualizar	Intervalo de tiempo entre actualización (segundos)
Iluminación	5
Temperatura	60
Humedad	60
Lluvia	120
Fecha y hora	5

Tabla 6. Intervalos de tiempo entre actualización jQuery

5. Implementación

La implementación la hemos dividido en tres bloques principales. El primero de ellos es Arduino, el segundo es el servidor Java el cual incluye REST, y el tercero y último es la interfaz web desarrollada en HTML.

5.1. Arduino

A continuación vamos a describir los pasos que hemos llevado a cabo para implementar el modulo de Arduino siguiendo el diseño elaborado en el capitulo anterior.

5.1.1. Configuración IDE Arduino

En primer lugar lo que tenemos que hacer cuando vamos a desarrollar un programa en Arduino es indicarle que placa vamos a utilizar, ya que sin esto luego no se puede compilar y cargar el código en la placa Arduino. En nuestro caso como utilizamos la placa **Mega2560** hacemos lo siguiente: Herramientas -> Placa -> Arduino Mega or Mega 2560.

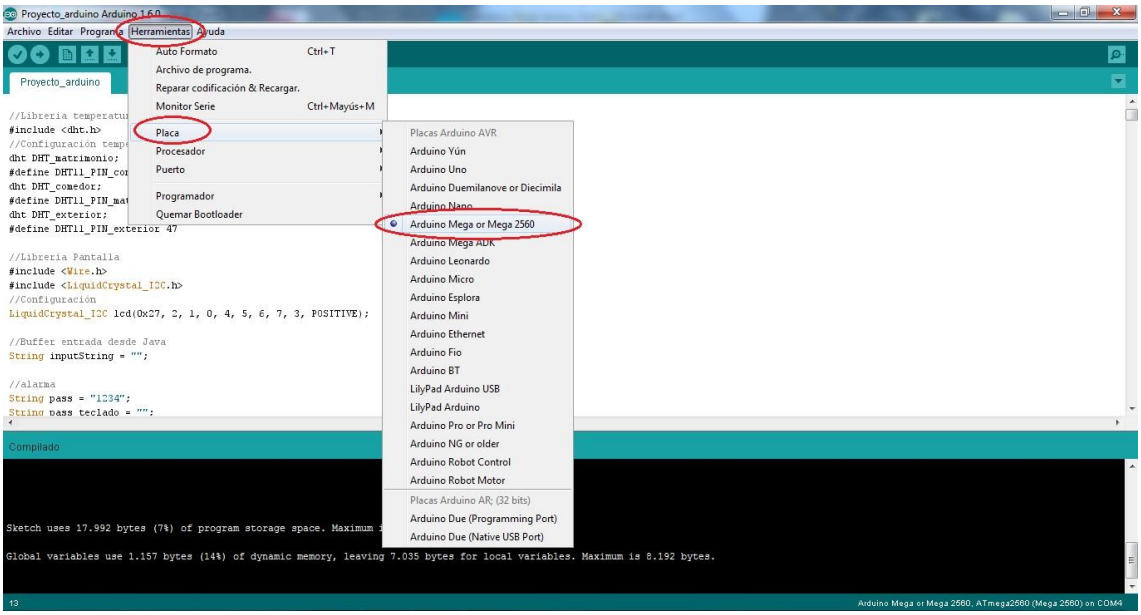

Ilustración 23. Selección placa Arduino

5.1.2. Importación de clases

Como ya ocurre con otros lenguajes como Java, en Arduino también se incluyen las librerías que vayamos a utilizar al inicio del programa.

Para importar la clase que necesitemos tenemos que añadirla al **workspace** de Arduino donde estamos trabajando, que por defecto se guarda en la siguiente ruta cuando usamos Windows: *C:\Users\usuario\Documents\Arduino\libraries*

Desde la interfaz de Arduino podemos añadir la librería dándole a Programa -> Importar Librería -> Añadir librería.

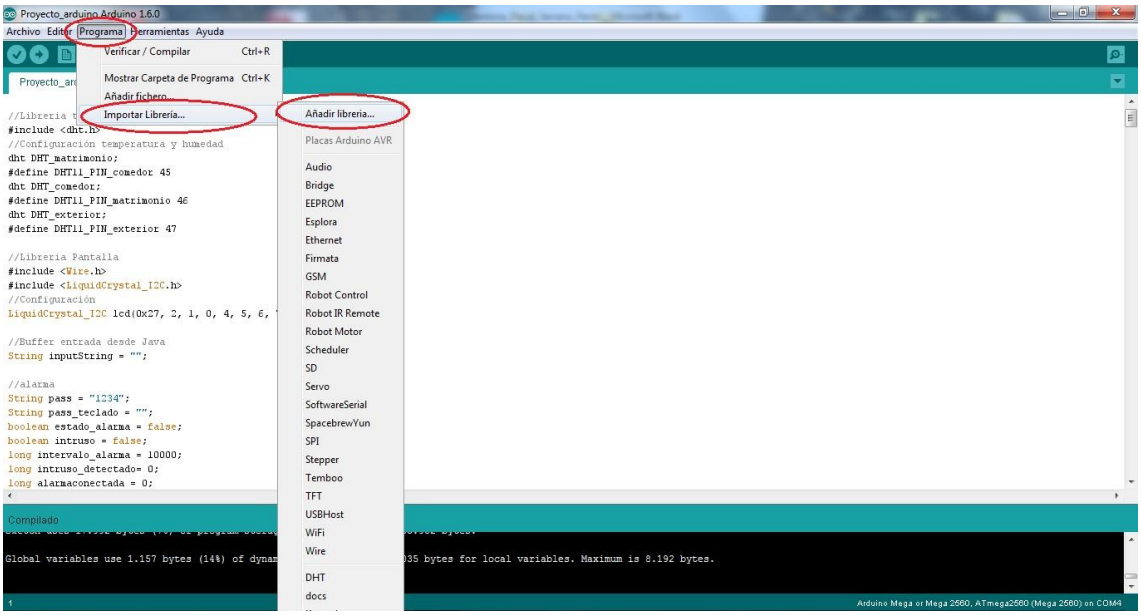

Ilustración 24. Añadir librería en Arduino

Por último, en el programa solo tendremos que añadir los nombres de las librerías. En nuestro programa usaremos las siguientes librerías:

- Temperatura y humedad para el sensor DHT11:
  #include <dht.h>
- Pantalla LCD
  #include <Wire.h>
  #include <LiquidCrystal_I2C.h>
- Teclado
  #include <Keypad.h>

Todas las librerías utilizadas se pueden descargar de la página oficial de Arduino.

5.1.3. Declaración de variables

A continuación detallamos las variables globales que hemos utilizado.

Para la temperatura y la humedad tenemos que declarar la variable de tipo *dht* que controle cada sensor, y el pin digital que estamos utilizando. También declaramos las variables de tipo *double* donde almacenaremos el valor de la última medida de la temperatura y humedad. Además tendremos variables para controlar el tiempo entre dos medidas.

```
dht DHT_matrimonio;
#define DHT11_PIN_matrimonio 46
long tiempo_ultima_temp = 0;
long tiempo_ultima_hum = 0;
double temp_matrimonio_actual;
double temp_matrimonio;
```

Para la **pantalla LCD** también hemos creado una variable global de tipo *LiquidCrystal_I2C*:

```
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);
```

Para la **comunicación** de Java hacia Arduino vamos usar una variable de tipo *String* que nos haga de buffer:

```
String inputString = "";
```

Para la **iluminación** hemos creado variables globales constantes de tipo *int* para almacenar el pin que vamos a usar. Las hemos nombrado con el prefijo *ed* para las entradas digitales, y el prefijo *sd* para las salidas digitales. También almacenamos el último valor tomado para el cual usamos el prefijo *st*.

```
const int ed_luz_matrimonio = 30;
const int sd_luz_matrimonio = 36;
int st_luz_matrimonio;
```

Para los **sensores** de **presencia** almacenamos el pin que vamos a utilizar en una variable constante de tipo *int*:

```
const int presencia_matrimonio = 42;
```

Para la configuración del **teclado** vamos a crear una variable tipo *char* para almacenar la última tecla pulsada. Creamos dos variables constantes de tipo *byte* para almacenar el número de columnas y filas. También almacenamos en dos *arrays* de tipo *byte* los números de pines utilizados. En una *array* de tipo *char* añadimos los diferentes caracteres de los que está formado el teclado. Por último, creamos la variable global que llamamos *teclado* de tipo *Keypad*:

```
char tecla;
const byte teclado_filas = 4;
const byte teclado_columnas =4;
byte teclado_pines_Filas[teclado_filas]={22,23,24,25};
byte teclado_pines_Columnas[teclado_columnas]={26,27,28,29};
//Teclado
char teclado_teclas[teclado_filas][teclado_columnas]={
        {'1','2','3','A'},
        {'4','5','6','B'},
        {'7','8','9','C'},
        {'*','0','#','D'},
};

Keypad teclado = Keypad(makeKeymap(teclado_teclas),teclado_pines_Filas,
teclado_pines_Columnas, teclado_filas, teclado_columnas);
```

5.1.4. Configuración inicial

La **configuración inicial** se realiza en la función *setup()*. Esta función es necesaria que se encuentre en el programa, incluso si no se tuviera que realizar ninguna configuración inicial. En nuestro caso, si que tenemos que hacerla, la cual detallamos a continuación:

- Inicializar comunicación serie:
  Serial.begin(9600);
- Inicializar la pantalla LCD:
  lcd.begin(16,2);
- Inicializar sensor:
  DHT11. DHT_lugar.read11(DHT11_PIN_lugar);
- Definir el uso de los pines digitales:
  pinMode(ed_luz_lugar, INPUT);
  pinMode(sd_luz_lugar, OUTPUT);

Además de esta inicialización imprescindible, vamos a indicar en la pantalla LCD que la alarma esta desconectada, y si hay alguna luz encendida la apagaremos. Esto último es para prevenir que si hay un corte de luz y se apaga Arduino, cuando vuelva a funcionar no se quede ninguna luz encienda produciendo un consumo innecesario.

```
apagar_luz("todas");
lcd.setCursor(0,0);
lcd.write("Alarma desconectada");
```

5.1.5. Bucle principal

El bucle principal en Arduino es la función *loop()*. Esta se ejecuta después de la función *setup()* y la inicialización de variables. Esta función se ejecuta de forma indefinida, con lo cual es donde se llama a todos los eventos o procesos que se encargaran de regular y gestionar todo el sistema.

En nuestro caso la función *loop()* se la encargada de llamar a siete eventos:

```
iluminacion_entrada();
teclado_pulsado();
alarma();
lluvia();
cambios_estado_luz();
cambio_temperatura();
cambio_humedad();
```

5.1.6. Control cambios de temperatura

La temperatura la vamos a comprobar cada minuto, para ello comprobamos la última vez que tomamos la medida:

```
if(millis() > tiempo_ultima_temp + 60000 ){...}
```

Para comprobar la temperatura guardamos en una variable la temperatura actual y la comparamos con la última medida. Si la temperatura no es igual la actualizamos y mandamos al servidor el nuevo valor:

```
temp_comedor_actual=DHT_comedor.temperature;
        if(abs(temp_comedor_actual - temp_comedor) > 0.1 ){
            temp_comedor = temp_comedor_actual;
            Serial.print("{temp_comedor:");
            Serial.print(temp_comedor);
            Serial.println("}");
        }
```

5.1.7. Control cambios de humedad

Para el control de la humedad lo primero que haremos es comprobar la última vez que se midió la humedad, para que entre cada medida transcurra un tiempo superior a un minuto y medio:

```
if(millis() > tiempo_ultima_hum + 9000 ){...}
```

Posteriormente si es necesario comprobar la humedad, guardaremos el valor actual en una variable y lo compararemos con el valor tomado en la última medida. Si es diferente actualizaremos el valor en Arduino y mandaremos el nuevo valor al servidor:

```
hum_comedor_actual=DHT_comedor.humidity;
        if(abs(hum_comedor_actual - hum_comedor) > 0.1 ){
            hum_comedor = hum_comedor_actual;
            Serial.print("{hum_comedor:");
            Serial.print(hum_comedor);
            Serial.println("}");
        }
```

5.1.8. Control cambios de lluvia

Para controlar si hay un cambio en el estado de lluvia comprobamos el valor actual que nos devuelve el sensor de lluvia para saber si actualmente está lloviendo o no, y lo comparamos con el valor almacenado en Arduino. Si es diferente actualizamos el valor y mandamos al servidor el nuevo valor del estado de lluvia.

El sensor de lluvia devuelve un valor entre 0 y 1023, dependiendo de la cantidad de agua. Nosotros tenemos que decidir cuándo consideramos que está lloviendo y cuando no, por eso se añade una variable (rango_lluvia) para acotar y poder extrapolarlo a un valor binario.

```
if(!estado_lluvia){
            if(analogRead(sensor_lluvia) > rango_lluvia + 50){
        estado_lluvia = true;
        Serial.println("{lluvia:true}");
    }
```

```
} else {
        if(analogRead(sensor_lluvia) < rango_lluvia){
    estado_lluvia = false;
   Serial.println("{lluvia:false}");
  }
}
```

5.1.9. Control alarma

Para el control de alarma lo hemos realizado mediante dos métodos. El primero de ellos se encarga de **controlar** el **teclado**. Este el encargado tanto de activar la alarma cuando se pulsa la letra 'A' o desactivarla cuando se pulsa la letra 'D' después de añadir el pin correspondiente de cuatro cifras en nuestro caso.

```
void teclado_pulsado(){
 tecla=teclado.getKey();
 if(tecla != NO_KEY){
  digitalWrite(buzzer,HIGH);
  delay(10);
  digitalWrite(buzzer,LOW);
  if(tecla == 'A' && estado_alarma==false){
   lcd.clear();
   lcd.setCursor(0,0);
        lcd.write("Alarma conectada");
        lcd.setCursor(0,1);
        estado_alarma=true;
        alarmaconectada = millis();
       }else if(tecla == 'B'){
        lcd.clear();
  lcd.setCursor(0,0);
        if(estado_alarma){ lcd.write("Alarma conectada");
            }else{ lcd.write("Alarma desconectada");
        }
        lcd.setCursor(0,1);
        pass_teclado="";
 }else if(tecla != 'D'){
  lcd.write('*');
        pass_teclado += tecla;
 }else if(tecla == 'D'){
        if(pass.equals(pass_teclado)){
            estado_alarma=false;
            pass_teclado="";
            intruso=false;
            digitalWrite(buzzer,LOW);
            lcd.clear();
   lcd.setCursor(0,0);
         lcd.write("Alarma desconectada");
         lcd.setCursor(0,1);
        }
```

```
        }else{

        }
    }
}
```

El segundo método es el encargado de **detectar intrusos** y **hacer sonar** la **alarma** en caso de que se note presencia y la alarma no se desconecta en un intervalo de diez segundos. Además cuando se conecta la alarma y han pasado los diez segundos para salir de la vivienda todas las luces se apagan para evitar un consumo innecesario en caso de que se nos olvidara apagar algún punto de iluminación.

```
void alarma(){
    if(millis() - alarmaconectada > intervalo_alarma){
        if(estado_alarma){
            if(!intruso){
              apagar_luz("todas");
                    if(digitalRead(presencia_comedor)== 1 ||
digitalRead(presencia_entrada)==1                          ||
digitalRead(presencia_matrimonio)==1){
                        intruso = true;
                        intruso_detectado=millis();
                        }
                } else {
                    if(millis()  -  intruso_detectado  >
intervalo_alarma) digitalWrite(buzzer, HIGH);
                }
            }
        }
}
```

5.1.10. Control cambios de iluminación

Para cada punto de iluminación comprobamos la entrada digital con el valor almacenado en Arduino. Si el valor es diferente guardamos el nuevo valor y llamamos a la función *estado_luz("lugar");* para que esta sea la encargada de informar al servidor del nuevo valor.

```
if(st_luz_matrimonio != digitalRead(ed_luz_matrimonio)){
    st_luz_matrimonio=digitalRead(ed_luz_matrimonio);
    estado_luz("matrimonio");
}
```

5.1.11. Control iluminación automática entrada

Para la iluminación de la entrada en vez de encenderla de forma manual, tenemos instalado un sensor de luminosidad y aprovechando el sensor de presencia que utilizamos para la detección de intrusos vamos a hacer que cuando no haya suficiente iluminación y se detecte la presencia de alguien la luz se encienda automáticamente.

También se apagara automáticamente cuando no detecte nadie en un periodo de tiempo de cinco segundos.

```
void iluminacion_entrada(){
    if(luz_entrada_auto){
        if(digitalRead(presencia_entrada)== 1){
            if(analogRead(sensor_luminosidad) < rango_iluminacion){
                tiempo_luz_entrada_ultimo=millis();
                encender_luz("entrada");
            }
        }else{
            if((millis() - tiempo_luz_entrada_ultimo) > tiempo_luz_entrada) apagar_luz("entrada");
        }

    }
}
```

5.1.12. Control mensajes desde servidor

Para la comunicación Java→ Arduino vamos a utilizar la función *serialEvent()*, la cual ya está definida en Arduino. Esta función no se invoca en el bucle principal como hacemos en el resto, si no que Arduino la ejecuta después del bucle principal si hay datos en el buffer de entrada.

Cuando Arduino ha detectado datos, la función *serialEvent()* se ejecuta y esta se encarga de concatenar cada byte hasta que tiene la instrucción completa, la cual está delimitada por un punto y coma. La instrucción es pasada como parámetro a un método que hemos creado (“*leido()*”) el cual procesa la instrucción y sus posibles parámetros para finalmente ejecutar dicha instrucción.

5.1.13. Estado actual lluvia

La función se llama *getLluvia()* e imprime en el puerto serial el valor de la variable *estado_lluvia* en formato JSON. Si está lloviendo imprime true, y si de lo contrario no está lloviendo imprime el valor false.

```
Serial.print("{lluvia:");
Serial.print(estado_lluvia);
Serial.println("}");
```

5.1.14. Estado actual iluminación

Esta función se encarga de mandar al servidor por el puerto serie el estado actual de algún punto de iluminación. Se le pasa como parámetro en formato *String* el lugar de donde se quiere saber estado de la iluminación. Responde al servidor con el estado en formato JSON.

```
void estado_luz(String lugar){

        String estado = "{luz_";
        if(lugar.equals("matrimonio")) estado += lugar + ":" +
digitalRead(ed_luz_matrimonio) + "}";
        else if(lugar.equals("aseomatrimonio")) estado += lugar + ":" +
digitalRead(ed_luz_aseomatrimonio) + "}";
        else if(lugar.equals("aseo")) estado += lugar + ":" +
digitalRead(ed_luz_aseo) + "}";
        else if(lugar.equals("entrada")) estado += lugar + ":" +
digitalRead(ed_luz_entrada) + "}";
        else if(lugar.equals("cocina")) estado += lugar + ":" +
digitalRead(ed_luz_cocina) + "}";
        else if(lugar.equals("comedor")) estado += lugar + ":" +
digitalRead(ed_luz_comedor) + "}";
        else {
            Serial.println("Error en lectura estado_luz");
            exit(0);
        }
        Serial.println(estado);
    }
```

5.1.15. Estado actual humedad

Esta función devuelve al servidor en formato JSON el estado actual de la humedad. Para ello utilizamos el parámetro de entrada para saber de que lugar necesitamos la humedad y luego mediante *DHT_matrimonio.humidity* obtenemos la humedad actual.

```
if(lugar.equals("matrimonio")){
  Serial.print("{hum_matrimonio:");
  Serial.print(DHT_matrimonio.humidity,1);
  Serial.println("}");
 }
```

5.1.16. Estado actual temperatura

La función *temperatura(lugar)* recibe como parámetro el lugar del cual tenemos que devolver la temperatura. Con DHT_*matrimonio.temperature* obtenemos la temperatura y se la mandamos por el puerto serie al servidor.

```
if(lugar.equals("matrimonio")){
  Serial.print("{temp_matrimonio:");
  Serial.print(DHT_matrimonio.temperature,1);
  Serial.println("}");
 }
```

5.1.17. Apagar luz

Para apagar una luz se llama a la función *apagar_luz(String lugar)* pasándole como parámetro el lugar donde hay que apagar la luz. Mediante una secuencia *if..else*

averiguamos que luz hay que apagar. Posteriormente comprobamos si la luz está encendida leyendo la entrada digital correspondiente mediante la función *digitalRead(pin)* siendo pin el número de la entrada digital. Si el valor es alto, significa que está encendida y procedemos a apagarla. Para apagar usamos la función *digitalWrite(pin,valor*) donde el pin es el número de la salida digital, y valor es el valor que nosotros vamos a cambiar, en nuestro caso como estaba encendida, hay que poner el valor contrario. Un ejemplo de apagar una luz sería así:

```
if(lugar.equals("aseo")){
        if(digitalRead(ed_luz_aseo)==HIGH){
            digitalWrite(sd_luz_aseo, !digitalRead(sd_luz_aseo));
        }
}
```

Por último, una vez apagada la luz, **añadimos una pausa** mediante el método *delay(ms)*, siendo ms los milisegundos de la pausa. Nosotros hemos añadido una pausa de 100 milisegundos. Esto es necesario ya que existe una **latencia** de aproximadamente 80 milisegundos desde que Arduino cambia el valor de la salida digital hasta que el valor en la entrada digital cambia. Dicha latencia se debe principalmente al **relé**, ya que como se trata de un mecanismo mecánico el cambio no es instantáneo.

Si no hiciésemos esta pausa, el problema que existe es que Arduino cambia el valor de la salida digital para apagar la luz, comprueba si la luz está apagada, y como no le ha dado tiempo debido a la latencia vuelve a cambiar la salida digital entrando en un bucle infinito.

Este problema apareció al realizar las pruebas en la maqueta. No ocurría de forma continua, con lo que averiguar el por qué la iluminación no funcionaba correctamente llevo mucho tiempo de investigación, nuevos diseños e implementaciones del mismo código. Esto demuestra la dificulta que supone llevar cualquier proyecto al mundo físico o real, ya que aumenta de forma exponencial las variables que hacen que lo que en un laboratorio funcione, luego no funcione en un sistema real.

5.1.18. Encender luz

La función encargada de encender la luz llamada encender_luz(String lugar) es casi idéntica a la función de apagar una luz. El única cambio que existe es cuando se comprueba si la luz está apagada, en la cual ahora hay que comprobar que tiene un valor bajo (false) para que si es así encenderla. Un ejemplo de su implementación es el siguiente:

```
if(lugar.equals("aseo")){
        if(digitalRead(ed_luz_aseo)==HIGH){
            digitalWrite(sd_luz_aseo, !digitalRead(sd_luz_aseo));
        }
}
```

También debemos usar la función **delay(ms)** para hacer una pausa, tal como hemos explicado en el apartado anterior.

5.2. Servidor Java

En este apartado hemos realizado una descripción de la implementación.

5.2.1. Túnel Arduino - Java

La comunicación entre Arduino y Java la controlara la clase conexion_rxtx, que realizara la función de **túnel** entre ambos. Para la inicialización de esta clase usamos un **patrón singleton**, con el cual conseguimos que solo se pueda instanciar una vez la clase.

```
private static conexion_rxtx instance = null;
public static conexion_rxtx getInstance(){
	if(instance==null)
		instance=new conexion_rxtx();
	return instance;
}
```

Utilizaremos la **librería RXTX** la cual utilizamos para implementar la clase *SerialPortEventListener* que nos facilitara la comunicación con Arduino mediante el puerto serie.Tenemos dos buffer, uno de entrada y otro de salida. El método *serialEvent(SerialPortEvent oEvent)* es el encargado de procesar el buffer de entrada el cual mandara la información recibida a su driver correspondiente. El método *escribir(String out)* recibirá de los drivers los datos para enviar a Arduino.

```
public synchronized void serialEvent(SerialPortEvent oEvent) {
	if (oEvent.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
	  try {
	String inputLine=input.readLine();

	if(inputLine.contains("luz_")){
 driver_luz.getInstance().entrada(inputLine);
	}else if(inputLine.contains("temp_")){
 driver_temperatura.getInstance().entrada(inputLine);
	}else if(inputLine.contains("hum_")){
 driver_humedad.getInstance().entrada(inputLine);
	}else if(inputLine.contains("lluvia")){
 driver_lluvia.getInstance().entrada(inputLine);
	}else{
   System.out.println("Error lectura. Conexión_rxtx:");
   System.out.println(inputLine);
	}
	} catch (Exception e) {
   System.err.println(e.toString());
	}
      }
}

 public void escribir(String out){
	byte[] array = out.getBytes();
	try{
	output.write(array);
	}catch (Exception e){}
 }
```

5.2.2. Drivers

Tenemos **cuatro clases** que son nuestros drivers correspondiente a la luz, humedad, temperatura y lluvia. Para que cada driver sea único hemos utilizado el patrón singleton. Un ejemplo de la instancia de uno de los drivers es:

```
public class driver_luz {
 private static driver_luz instance = null;
 public static driver_luz getInstance(){
       if(instance==null)
             instance=new driver_luz();
       return instance;
 }
…
}
```

5.2.2.1. Driver luz

El driver luz consta de cuatro métodos que utilizando el lenguaje definido para la comunicación entre Arduino y Java permite la interacción entre ellos.

```
public void encender(String lugar){
 conexion_rxtx.getInstance().escribir("encender_luz,"+lugar+";");
}

public void apagar(String lugar){
 conexion_rxtx.getInstance().escribir("apagar_luz,"+lugar+";");
}

public void estado(String lugar){
 conexion_rxtx.getInstance().escribir("estado_luz,"+lugar+";");
}

public void entrada(String input){
   if(input.contains("comedor"))
luz.getInstance().set_comedor(input.contains("1"));
   else if(input.contains("cocina"))
luz.getInstance().set_cocina(input.contains("1"));
   else if(input.contains("aseomatrimonio"))
luz.getInstance().set_aseomatrimonio(input.contains("1"));
   else if(input.contains("aseo"))
luz.getInstance().set_aseo(input.contains("1"));
   else if(input.contains("matrimonio"))
luz.getInstance().set_matrimonio(input.contains("1"));
   else if(input.contains("entrada"))
luz.getInstance().set_entrada(input.contains("1"));
   else System.out.println("Error driver lectura luz");
         }
}
```

5.2.2.2. Driver temperatura

Esta clase está compuesta por dos métodos. El primero de ellos realiza una petición a Arduino para saber la temperatura actual de uno de los sensores. El segundo es el encargado de procesar el JSON que recibe por el túnel creado desde Arduino a Java.

```
 public void temp_actual(String lugar){
        conexion_rxtx.getInstance().escribir("temperatura,"+lugar+";");
 }

 public void entrada(String input){

        if(input.contains("comedor"))
temperatura.getInstance().set_temp_comedor(Double.parseDouble(input.subs
tring(input.indexOf(":")+1, input.length()-1)));
        else if(input.contains("matrimonio"))
temperatura.getInstance().set_temp_matrimonio(Double.parseDouble(input.s
ubstring(input.indexOf(":")+1, input.length()-1)));
        else if(input.contains("exterior"))
temperatura.getInstance().set_temp_exterior(Double.parseDouble(input.sub
string(input.indexOf(":")+1, input.length()-1)));
        else System.out.println("Error driver lectura temperaura");
 }
```

5.2.2.3. Driver humedad

Este driver también está compuesto por dos métodos que se encargan de pedir la humedad actual, y posteriormente de procesarla cuando llegue desde Arduino.

```
 public void hum_actual(String lugar){
        conexion_rxtx.getInstance().escribir("humedad,"+lugar+";");

 }

 public void entrada(String input){
        if(input.contains("comedor"))
humedad.getInstance().set_hum_comedor(Double.parseDouble(input.substring
(input.indexOf(":")+1, input.length()-1)));
        else if(input.contains("matrimonio"))
humedad.getInstance().set_hum_matrimonio(Double.parseDouble(input.substr
ing(input.indexOf(":")+1, input.length()-1)));
        else if(input.contains("exterior"))
humedad.getInstance().set_hum_exterior(Double.parseDouble(input.substrin
g(input.indexOf(":")+1, input.length()-1)));
        else System.out.println("Error driver lectura temperaura");
 }
```

5.2.2.4. Driver lluvia

Por último, el driver correspondiente a controlar el estado de la lluvia, consta de un método para pedir el estado actual de la lluvia, y otro para procesar el estado de la lluvia cuando nos llega por el túnel creado.

```
 public void lluvia_actual(){
        conexion_rxtx.getInstance().escribir("estado_lluvia;");
 }

 public void entrada(String input){
        if(input.contains("true"))
lluvia.getInstance().set_estado_lluvia(true);
        else if(input.contains("false"))
lluvia.getInstance().set_estado_lluvia(false);
```

```
        else System.out.println("Error driver lectura lluvia");

    }
```

5.2.3. Capa lógica

El paquete de lógica de negocio está compuesto por **cinco clases**, las cuatro correspondientes a la luz, humedad, temperatura y lluvia, y la clase *AplicacionDomotica* que contiene el *main* siendo la encargada de inicializar el resto de clases.

5.2.3.1. Clase principal

En el main de nuestro programa vamos a inicializar las cuatro clases. Como utilizamos el patrón Singleton no necesitamos guardar ninguna variable de cada clase.

```
luz.getInstance();
humedad.getInstance();
temperatura.getInstance();
lluvia.getInstance();
```

5.2.3.2. Clase luz

La clase luz almacena el estado de cada punto de iluminación mediante variables *Boolean*. En nuestro caso tenemos seis variables de este tipo. Para cada variable tenemos un método *get* y otro *set* con los cuales se consulta el estado y se cambia respectivamente. En la inicialización del método se pregunta a Arduino el estado actual de cada punto de iluminación.

```
package logica;

import driver_arduino.driver_luz;

public class luz {
 private boolean comedor;
 private boolean cocina;
 private boolean aseo;
 private boolean matrimonio;
 private boolean aseomatrimonio;
 private boolean entrada;

 private static luz instance = null;

 public static luz getInstance(){
        if(instance==null)
                instance=new luz();
        return instance;
 }

 private luz(){
        estado("comedor");
        estado("cocina");
        estado("aseo");
        estado("matrimonio");
```

```
        estado("aseomatrimonio");
        estado("entrada");
    }

    public void encender(String lugar){
        driver_luz.getInstance().encender(lugar);
    }

    public void apagar(String lugar){
        driver_luz.getInstance().apagar(lugar);
    }

    public void estado(String lugar){
        driver_luz.getInstance().estado(lugar);
    }

    public boolean get_comedor(){
        return comedor;
    }

    public boolean get_cocina(){
        return cocina;
    }

    …. [Código omitido]

    public void set_comedor(boolean estado){
        comedor=estado;
    }

    public void set_cocina(boolean estado){
        cocina=estado;
    }

    …. [Código omitido]
}
```

5.2.3.3. Clase temperatura

En esta clase tenemos cuatro variables tipo *double* donde guardamos el valor de la temperatura de los diferentes lugares. Para acceder a su valor usamos el método *get*, y para cambiar su valor usamos el método *set*. En la inicialización de la clase llamaremos al método *temp_actual()* para que Arduino nos mande su valor actual.

```
package logica;

import driver_arduino.driver_temperatura;

public class temperatura {
    private double temp_comedor;
    private double temp_matrimonio;
    private double temp_exterior;

    private static temperatura instance = null;
    public static temperatura getInstance(){
        if(instance==null)
            instance=new temperatura();
```

```
        return instance;
 }

 private temperatura(){
        temp_actual("comedor");
        temp_actual("matrimonio");
        temp_actual("exterior");
 }

 public void temp_actual(String lugar){
        driver_temperatura.getInstance().temp_actual(lugar);
 }

 public double get_temp_comedor(){
        return temp_comedor;
 }
 …. [Código omitido]
 public void set_temp_comedor(double temp){
        temp_comedor=temp;
 }
 …. [Código omitido]
}
```

5.2.3.4. Clase humedad

Tenemos tres variables tipo *double* donde almacenamos los valores de la humedad. También disponemos de los métodos para consultar su estado y cambiarlos. Por último tenemos el método *hum_actual()* para pedir a Arduino la humedad actual cuando se inicialice la clase por primera vez.

```
package logica;

import driver_arduino.driver_humedad;

public class humedad {
 private double hum_comedor;
 private double hum_matrimonio;
 private double hum_exterior;

 private static humedad instance = null;
 public static humedad getInstance(){
        if(instance==null)
               instance=new humedad();
        return instance;
 }

 private humedad(){
        hum_actual("comedor");
        hum_actual("matrimonio");
        hum_actual("exterior");
 }

 public void hum_actual(String lugar){
        driver_humedad.getInstance().hum_actual(lugar);
 }

 public double get_hum_comedor(){
```

```
        return hum_comedor;
 }
 …. [Código omitido]

 public void set_hum_comedor(double hum){
        hum_comedor=hum;
 }
 …. [Código omitido]
}
```

5.2.3.5. Clase lluvia

La clase lluvia consta de una variable tipo *Boolean* que almacena el estado de la lluvia actual. Mediante los métodos *get* y *set* se consulta y se cambia su estado respectivamente. El método *lluvia_actual()* es el encargado de preguntar a Arduino el estado actual de la lluvia.

```
package logica;

import driver_arduino.driver_lluvia;

public class lluvia {
 private boolean estado_lluvia;

 private static lluvia instance = null;

 public static lluvia getInstance(){
        if(instance==null)
               instance=new lluvia();
        return instance;
 }

 private lluvia(){
        lluvia_actual();
 }

 public void lluvia_actual(){
        driver_lluvia.getInstance().lluvia_actual();
 }

 public void set_estado_lluvia(boolean estado){
        estado_lluvia = estado;
 }

 public boolean get_estado_lluvia(){
        return estado_lluvia;
 }
}
```

5.3. REST (Restlet)

Para la implementación, lo primero que tenemos que hacer es **inicialización** del servicio REST, el cual lo hacemos en *main*.

```
     // Create a new Component.
component component = new Component();
```

```
        // Add a new HTTP server listening on port 8182.
component.getServers().add(Protocol.HTTP, 8182);

        // Attach the sample application.
component.getDefaultHost().attach("/dispositivo",
new RestApplication());

        // Start the component.
 component.start();
```

En la clase *RestApplication* creamos el **router** donde se procesara cada petición REST.

```
package rest;

import org.restlet.*;
public class RestApplication extends Application {
    /**
     * Creates a root Restlet that will receive all incoming calls.
     */
    @Override
    public synchronized Restlet createInboundRoot() {
        // Create a router Restlet that routes each call to a new
instance.
        Router router = new Router(getContext());

        // Defines router
        router.attach("/luz/{lugar}", luz_Resource.class);
        router.attach("/luces", luz_Resource.class);
        router.attach("/humedades", humedad_Resource.class);
        router.attach("/temperaturas", temperatura_Resource.class);
        router.attach("/lluvia", lluvia_Resource.class);

        return router;
    }
}
```

Para el recurso de **luz** vamos a crear una clase llamada *luz_Resourse* que se encargara de procesar las peticiones GET y POST.

```
package rest;

import logica.luz;
import org.json.*;
import org.restlet.resource.*;

public class luz_Resource extends ServerResource {
    @Get
    public String represent() {
        JSONObject json = new JSONObject();
        try {
              json.put("luz_comedor", luz.getInstance().get_comedor());
              json.put("luz_aseomatrimonio",
luz.getInstance().get_aseomatrimonio());
              json.put("luz_matrimonio",
luz.getInstance().get_matrimonio());
```

```
                json.put("luz_aseo", luz.getInstance().get_aseo());
                json.put("luz_cocina", luz.getInstance().get_cocina());
                json.put("luz_entrada", luz.getInstance().get_entrada());
            } catch (JSONException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            return json.toString();
        }

        @Post
        public void cambio_luz(String payload){
            if(payload.contains("true"))
luz.getInstance().encender(getAttribute("lugar"));
            else if(payload.contains("false"))
luz.getInstance().apagar(getAttribute("lugar"));
        }
}
```

Para los recursos de **humedad**, creamos la clase *humedad_Resource*. Este recurso solo aceptará peticiones GET.

```
package rest;

import logica.humedad;
import org.json.*;
import org.restlet.resource.*;

public class humedad_Resource extends ServerResource {
    @Get
    public String represent() {
        JSONObject json = new JSONObject();
        try {
                json.put("hum_comedor",
humedad.getInstance().get_hum_comedor());
                json.put("hum_matrimonio",
humedad.getInstance().get_hum_matrimonio());
                json.put("hum_exterior",
humedad.getInstance().get_hum_exterior());
        } catch (JSONException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        return json.toString();
    }
}
```

La **temperatura** la controlara la clase *temperature_Resource*. Esta clase solo acepta peticiones GET.

```
package rest;

import logica.temperatura;
import org.json.*;
import org.restlet.resource.*;

public class temperatura_Resource extends ServerResource {
```

```
    @Get
    public String represent() {
        JSONObject json = new JSONObject();
        try {
               json.put("temp_comedor",
temperatura.getInstance().get_temp_comedor());
               json.put("temp_matrimonio",
temperatura.getInstance().get_temp_matrimonio());
               json.put("temp_exterior",
temperatura.getInstance().get_temp_exterior());

        } catch (JSONException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
        }
        return json.toString();
    }
}
```

Por último, la **lluvia** es controlada por la clase *lluvia_Resource*. Este recurso solo aceptara peticiones GET ya que solo se le podrá solicitar información.

```
package rest;

import logica.lluvia;
import org.json.*;
import org.restlet.resource.*;

public class lluvia_Resource extends ServerResource {

    @Get
    public String represent() {
        JSONObject json = new JSONObject();
        try {
               json.put("estado_lluvia",
lluvia.getInstance().get_estado_lluvia());
        } catch (JSONException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
        }
        return json.toString();
    }
}
```

Con esto acabamos toda la implementación en Java. En la siguiente imagen podemos observar todas las clases creadas y librerías utilizadas para la implementación de servidor Java y REST.

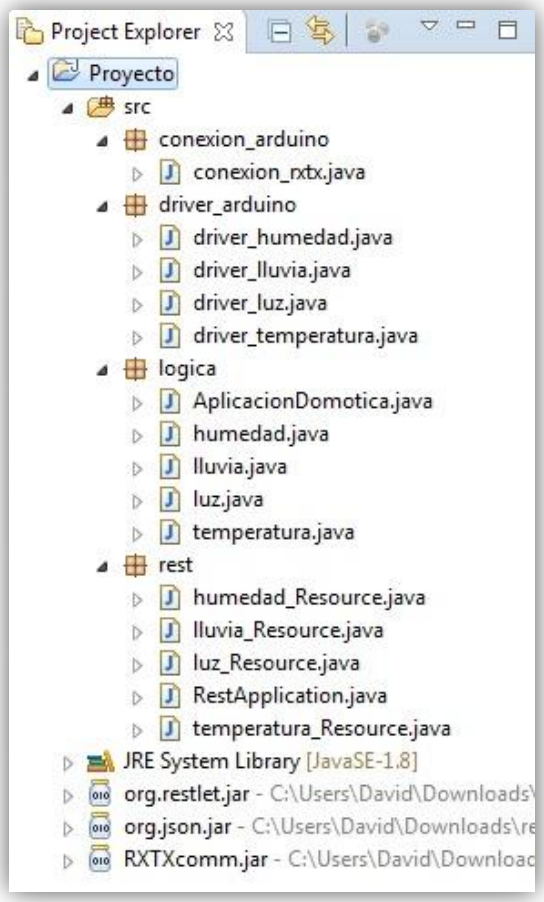

Ilustración 25. Clases y librerías en Eclipse IDE

5.4. Interfaz Web

Para la implementación de la interfaz web hemos hecho dos apartados. En primer lugar hemos abordado la parte estática la cual está desarrollada con **HTML** y **CSS**. Posteriormente hemos desarrollado la parte dinámica en **jQuery**, haciendo también uso de PHP para poder hacer las peticiones a REST.

5.4.1. Parte estática: HTML y CSS

Vamos a estructurar la pantalla en dos zonas verticales. A la derecha ponemos la hora y la fecha, la temperatura y humedad exterior, y si está lloviendo. Y a la izquierda ponemos el plano de la vivienda. El plano lo ponemos de fondo:

```
<div id=iluminacion style="background: white url('img/Vivienda-
Model.png') no-repeat center center; background-size: 100% 100%;">
```

Sobre el plano de la vivienda vamos sobreponiendo cada bombilla que representara cada punto de iluminación de la vivienda. Tenemos dos bombillas, una apagada y otra encendida dependiendo del estado actual en el que se encuentre. También ponemos en la temperatura y la humedad en la habitación de matrimonio y en la cocina-comedor.

Cada uno de estos elementos los dispondremos sobre *div* diferente y los etiquetaremos con su identificador correspondiente:

```
<div id=matrimonio><a href="#"><img alt="" src="img/luz_off.png"
title=""></a></div>
```

Mediante las **hojas de estilo** vamos ajustando cada elemento donde corresponde, todo ello con porcentajes para que cuando se redimensione la ventana cada elemento siga estando en su lugar correspondiente y el plano se ajuste al tamaño máximo disponible que ofrezca la ventana en ese momento.

```
#matrimonio img{
    width: 60px;
}
#matrimonio {
    height: 30%;
    width: 30%;
    text-align: center;
    float: left;
}
```

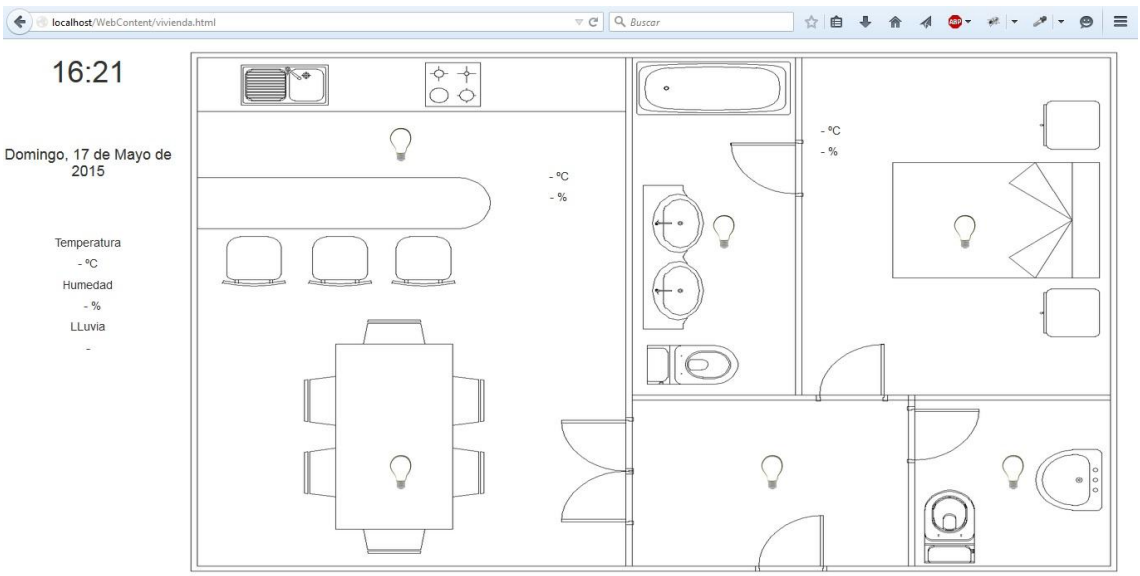

Ilustración 26. Interfaz web.

5.4.2. Parte dinámica: jQuery y PHP

Para controlar la hora lo primero que hemos hecho es definir en jQuery dos variables globales para almacenar todos los **días** y **meses**.

```
var dia_semana = [
"Domingo",
"Lunes",
"Martes",
"Mi&eacute;rcoles",
"Jueves",
"Viernes",
```

```
"S&aacute;bado"
];

var mes = [
"Enero",
"Febrero",
"Marzo",
"Abril",
"Mayo",
"Junio",
"Julio",
"Agosto",
"Septiembre",
"Octubre",
"Noviembre",
"Diciembre"
];
```

Posteriormente hemos creado una función para actualizar la hora cambia la hora que se mostraba por la hora actual. La **hora y fecha** que muestra es la que tiene el ordenador o dispositivo que estemos utilizando.

```
function actualizar(){
 var fecha = new Date();
       $("#hora h1").replaceWith("<h1>"+fecha.getHours() + ":" +
 (fecha.getMinutes()<10?'0':'') + fecha.getMinutes()+"</h1>");
       $("#fecha h4").replaceWith("<h4>"+dia_semana[fecha.getDay()]+",
 "+fecha.getDate()+" de "+mes[fecha.getMonth()]+" de
 "+fecha.getFullYear()+"</h4>");
}
```

Por último, hacemos que se llame de **forma periódica** a la función que hemos creado para ir actualizando la hora.

```
$(document).ready(function(){
 actualizar();
 setInterval(actualizar,5000);
});
```

Cuando hacemos click sobre una bombilla jQuery detecta sobre que bombilla hemos pulsado para ejecutar el código correspondiente. Lo primero que hacemos es comprobar si la bombilla está encendida o apagada. Hacemos una petición **POST** a al url de REST correspondiente indicando si tiene que encender o apagarse la luz. Seguidamente cambiamos la bombilla de estado.

```
$(document).ready(function(){
   $("#aseomatrimonio a").click(function(){
         if(($("#aseomatrimonio a img").attr("src")) ==
"img/luz_on.png"){
     $.post("http://localhost:8182/dispositivo/luz/aseomatrimonio",
     {
       estado:"false"
     });
     $("#aseomatrimonio a img").attr("src","img/luz_off.png");
   }else{
         $.post("http://localhost:8182/dispositivo/luz/aseomatrimonio",
```

```
                        {
                          estado:"true"
                        });
                        $("#aseomatrimonio a
img").attr("src","img/luz_on.png");
    }
    });
  });
```

Cambiamos la bombilla en el instante que le damos para que el usuario no observe la latencia que hay entre que pulsa la bombilla, le llega la instrucción a Arduino para cambiar el estado, y vuelve el cambio de estado. Lo que ocurre es que si le damos y por cualquier problema no se cambia la luz, en la interfaz sí que aparecería cambiada. Pero en cuanto se volviera a actualizar el estado de la iluminación volvería la luz a su estado real.

Hemos creado una función que se encarga de actualizar todas las bombillas. Esta función hace una petición **GET** a REST para saber el estado actual.

```
$.get_luz = function(){
  var xmlhttp;

    if (window.XMLHttpRequest){// code for IE7+, Firefox, Chrome, Opera,
Safari
      xmlhttp=new XMLHttpRequest();
    }else{// code for IE6, IE5
      xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
    }

        xmlhttp.onreadystatechange=function(){
              if (xmlhttp.readyState==4 && xmlhttp.status==200){
                    var json = jQuery.parseJSON(xmlhttp.responseText);
                    $("#comedor a img").attr("src",
(json.luz_comedor==true) ? "img/luz_on.png" : "img/luz_off.png");
                    $("#aseo a img").attr("src", (json.luz_aseo==true)
? "img/luz_on.png" : "img/luz_off.png");
                    $("#aseomatrimonio a img").attr("src",
(json.luz_aseomatrimonio==true) ? "img/luz_on.png" : "img/luz_off.png");
                    $("#cocina a img").attr("src",
(json.luz_cocina==true) ? "img/luz_on.png" : "img/luz_off.png");
                    $("#entrada a img").attr("src",
(json.luz_entrada==true) ? "img/luz_on.png" : "img/luz_off.png");
                    $("#matrimonio a img").attr("src",
(json.luz_matrimonio==true) ? "img/luz_on.png" : "img/luz_off.png");
              }
        }
    xmlhttp.open("GET","php/get_luz.php",true);
    xmlhttp.send();
}
```

Como hemos comentado antes, todas las peticiones GET que hacemos tenemos que hacer a través de **PHP** como pasarela, ya que jQuery no acepta peticiones a dominios diferentes. La petición para el estado de la iluminación es así:

```
<?php
        $response ="";
```

```
        $response =
file_get_contents("http://localhost:8182/dispositivo/luz");
        echo $response;
    ?>
```

Para la temperatura, humedad y lluvia también hemos creado tres funciones que se encargan de actualizarlas respectivamente.

```
$.get_temperatura = function(){
 var xmlhttp;

   if (window.XMLHttpRequest){// code for IE7+, Firefox, Chrome, Opera,
Safari
     xmlhttp=new XMLHttpRequest();
   }else{// code for IE6, IE5
     xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
   }

       xmlhttp.onreadystatechange=function(){
            if (xmlhttp.readyState==4 && xmlhttp.status==200){
                 var json = jQuery.parseJSON(xmlhttp.responseText);
                 $("#temp_exterior").replaceWith("<h5
id=temp_exterior>"+json.temp_exterior+" &#186;C</h5>");
                 $("#temp_comedor").replaceWith("<h5
id=temp_comedor>"+json.temp_comedor+" &#186;C</h5>");
                 $("#temp_matrimonio").replaceWith("<h5
id=temp_matrimonio>"+json.temp_matrimonio+" &#186;C</h5>");
                 }
       }
   xmlhttp.open("GET","php/get_temperatura.php",true);
   xmlhttp.send();

}
$.get_humedad = function(){
 var xmlhttp;

   if (window.XMLHttpRequest){// code for IqE7+, Firefox, Chrome,
Opera, Safari
     xmlhttp=new XMLHttpRequest();
   }else{// code for IE6, IE5
     xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
   }

       xmlhttp.onreadystatechange=function(){
            if (xmlhttp.readyState==4 && xmlhttp.status==200){
                 var json = jQuery.parseJSON(xmlhttp.responseText);
                 $("#hum_exterior").replaceWith("<h5
id=hum_exterior>"+json.hum_exterior+" %</h5>");
                 $("#hum_comedor").replaceWith("<h5
id=hum_comedor>"+json.hum_comedor+" %</h5>");
                 $("#hum_matrimonio").replaceWith("<h5
id=hum_matrimonio>"+json.hum_matrimonio+" %</h5>");
                 }
       }
   xmlhttp.open("GET","php/get_humedad.php",true);
   xmlhttp.send();

}
```

```
$.get_lluvia = function(){
 var xmlhttp;

   if (window.XMLHttpRequest){// code for IE7+, Firefox, Chrome, Opera,
Safari
     xmlhttp=new XMLHttpRequest();
   }else{// code for IE6, IE5
     xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
   }

        xmlhttp.onreadystatechange=function(){
              if (xmlhttp.readyState==4 && xmlhttp.status==200){
                    var json = jQuery.parseJSON(xmlhttp.responseText);
                    if(json.estado_lluvia==true)
$("#lluvia_exterior").replaceWith("<h5 id=lluvia_exterior>Si</h5>");
                    else if(json.estado_lluvia==false)
$("#lluvia_exterior").replaceWith("<h5 id=lluvia_exterior>No</h5>");
                    else  $("#lluvia_exterior").replaceWith("<h5
id=lluvia_exterior>-</h5>");

              }
        }
   xmlhttp.open("GET","php/get_lluvia.php",true);
   xmlhttp.send();
}
```

Estas funciones también necesitan hacer su petición GET mediante PHP para solucionar el problema del CORS igual que hemos hecho anteriormente.

Por último, cuando carguemos por primera vez la página web jQuery hará las llamadas a las funciones que se encargan de actualizar la luz, temperatura, humedad y lluvia para que están se muestren. Luego haremos que estas funciones se ejecuten cada cierto intervalo para que la página web se vaya actualizando de forma dinámica sin intervención del usuario.

```
$(document).ready(function(){
   $.get_luz();
   $.get_temperatura();
   $.get_humedad();
   $.get_lluvia();
   setInterval($.get_luz,5000);
   setInterval($.get_humedad,60000);
   setInterval($.get_temperatura,60000);
   setInterval($.get_lluvia,60000);
});
```

6. Desarrollo prototipo

La maqueta tiene un tamaño de un DIN A2. Sobre esto se dispondrá la vivienda unifamiliar con un tamaño de un DIN A3, y el espacio restante se utilizara para poner la placa Arduino y todos los sensores, actuadores y dispositivos que formen parte de la vivienda domótica.

6.1. Proceso de construcción de la maqueta

La maqueta se va a construir mediante **marquetería**. Se utiliza un grosor de 5 milímetros. Las paredes tendrán una altura de 5 centímetros. Se pondrán unos tacos de madera para elevar la base y poder poner la mayoría del cableado por la parte inferior.

Para las paredes y los tacos se utilizan punchas y adhesivo para que se sujete de forma firme sobre el tablero principal. Sobre el tablero se hacen unos agujeros y hendiduras para poner la pantalla LCD y el teclado.

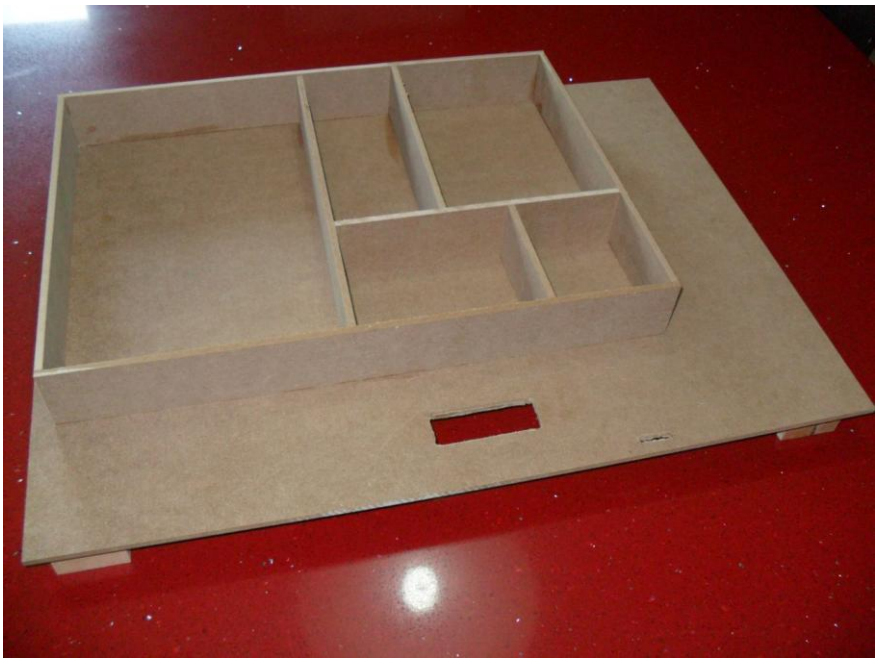

Ilustración 27. Construcción carpintería

Se coloca cada punto de iluminación con sus respectivos conmutadores y se pone el cableado correspondiente. Para la unión de estos se usa soldadura de estaño.

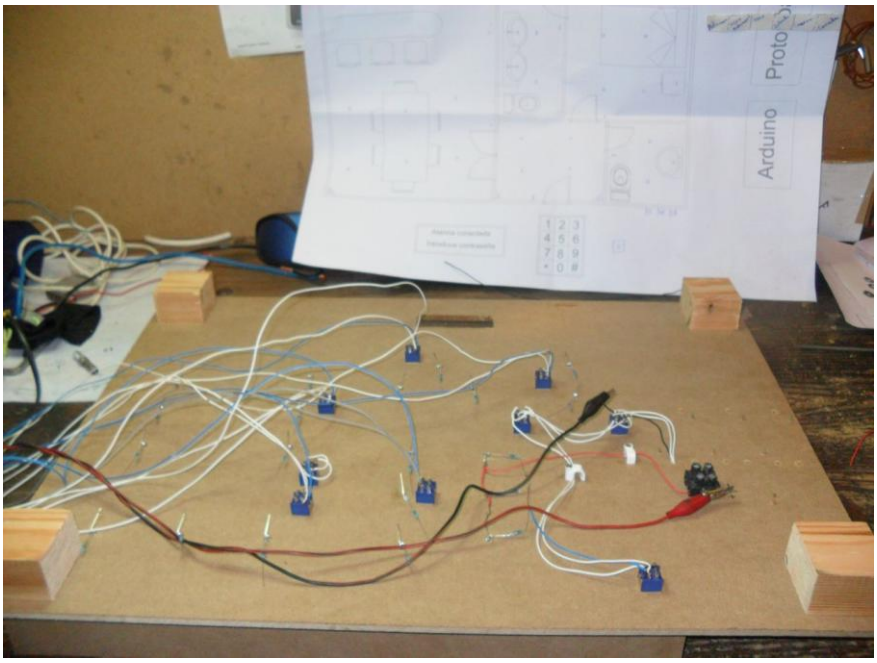

Ilustración 28. Proceso de cableado de la maqueta 1

Posteriormente colocamos los relés que nos permitirá mediante Arduino interactuar con cada punto de iluminación, y además harán que en caso de que Arduino deje de funcionar, la iluminación siga funcionando correctamente ya que los relés simplemente hacen la función de un conmutador. Si hubiéramos utilizados pulsadores, y Arduino deja de funcionar, perderíamos el control de los puntos de iluminación.

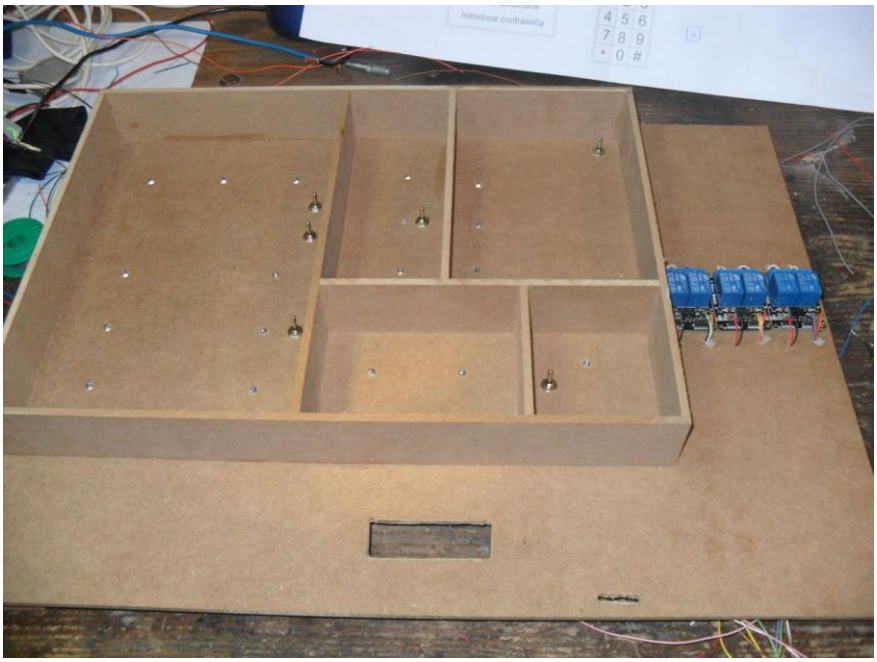

Ilustración 29. Proceso de cableado de la maqueta 3

Por último, añadimos todos los sensores, la pantalla LCD, el teclado, el zumbador y la placa Arduino.

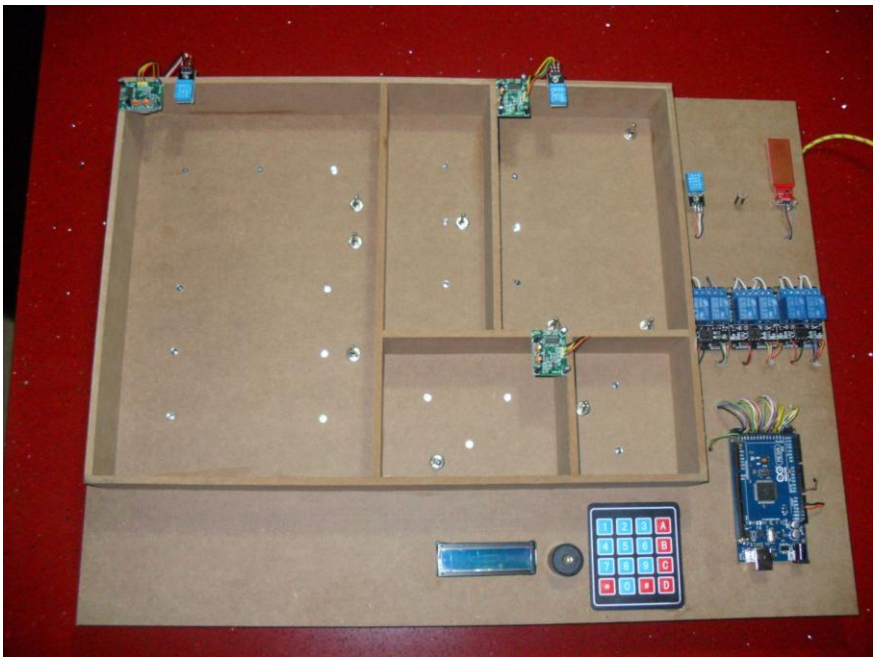

Ilustración 30. Maqueta finalizada 1

Acabamos de pasar todo el cableado que une cada sensor o actuador con Arduino.

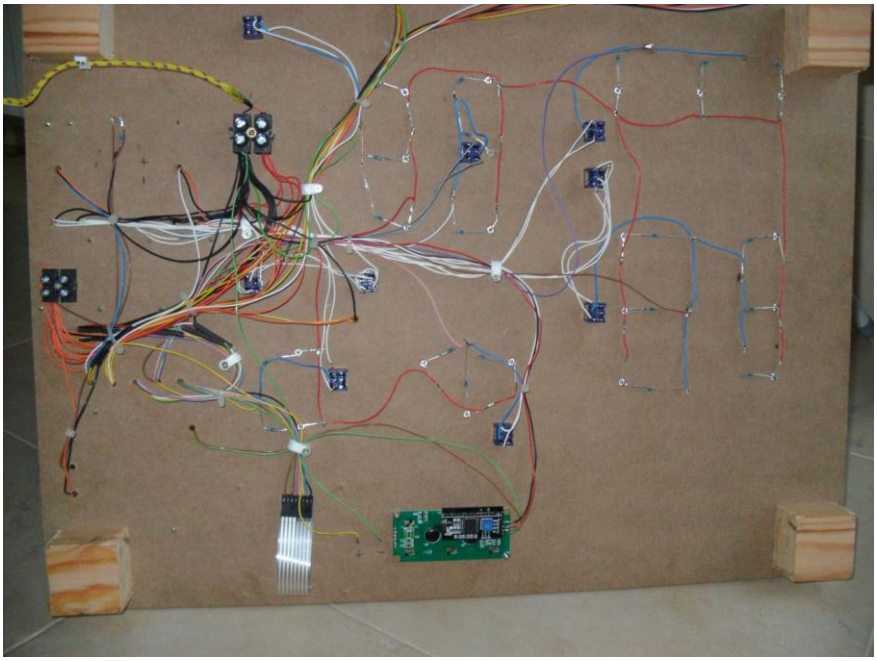

Ilustración 31. Maqueta finalizada 2

7. Propuestas de mejora

El sistema de una vivienda inteligente siempre estará abierto a nuevas mejoras y cambios con el objetivo de mejorar la confortabilidad de los usuarios.

En nuestro diseño se han cubierto una parte muy pequeña de todas las funcionalidades que una vivienda domótica puede tener hoy en día, con lo que cada añadido supondría una mejora para el sistema.

Con respecto a mi diseño, cambiaria varios aspectos. El primero de ellos, y ya comentado anteriormente, sería realizar una **interfaz web y móvil** que cumpla las necesidades de los usuarios siguiendo las técnicas y patrones necesarias para cumplir todos los requisitos.

La segunda mejora tiene que ver con el servidor Java. Una vez realizado el diseño e implementación me he dado cuenta de que no se realizó un diseño correcto. Actualmente la clase luz se encarga de controlar todas las luces. En cambio, un diseño más correcto sería que cada punto de luz fuera controlado por una clase luz, con lo que habría tantas clases luz como puntos de iluminación. Lo mismo ocurre con la temperatura y la humedad. El objetivo sería que **cada objeto físico tuviera su** propio objeto virtual en forma de **clase en Java**. Esto permitiría una mayor facilidad tanto a la hora de programarlo, como a la hora de procesar toda la información.

Otra mejora, también comentada anteriormente, sería añadir la tecnología **WebSocker** con lo que evitaríamos la actualización periódica que tiene la interfaz para mantenerla actualizada consumiendo unos recursos innecesarios.

La alarma y el encendido automático de la luz de la entrada lo controla y gestiona Arduino. Considero que no es una buena opción ya que Arduino solo debería hacer de intermediario entre los sensores y actuadores, y el servidor Java. Y ser este, concretamente la capa de lógica de negocio, la que se encargue de evaluar cada situación y realizar las acciones pertinentes.

Por último, y más que una mejora, es un elemento necesario en una vivienda domótica real, es la instalación y configuración de una **Raspberry Pi** que haría la funcionalidad de servidor Java y servidor Web. Se configuraría también un dominio para tener salida a través de Internet con lo que también sería necesario crear un mecanismo de seguridad para poder acceder al control de la vivienda domótica.

8. Conclusiones

En este trabajo hemos tenido que trabajar con **varias tecnologías** diferentes y hacer que se comunicaran entre ellas, suponiendo un gran reto cuando comenzamos a realizar el proyecto.

La primera tecnología que usamos fue **Arduino**. Esto no implicaba solo la implementación de la placa, sino también estudiar cada dispositivo que íbamos a conectar, las librerías que hacían posible la comunicación y la realización de los circuitos eléctricos.

Posteriormente creamos un **servidor** en **Java**, el cual se iba a encargar de la comunicación con Arduino, almacenar el estado de la vivienda y realizar cambios sobre esta. También se añadió un servicio **REST** el cual nos permitirá interactuar desde la interfaz web.

Por último, creamos un **servidor web** para ver el estado de la vivienda y apagar o encender los puntos de iluminación.

En este proyecto he podido por una parte aplicar los conocimientos adquiridos a lo largo de la carrera, y por otra parte, y desde mi punto de vista muy importante, ha sido el poder hacer frente a nuevas tecnologías que no hemos visto, pero que con los conocimientos y experiencia adquirida en estos años es necesario que sepamos enfrentarnos a nuevas tecnologías.

El hacer un **prototipo** del **sistema** diseñado ha sido muy importante porque nos proporciona un **escenario casi real**, el cual es muy diferente a la mayoría de prácticas que hemos realizado ya que siempre tiene un escenario más definido y guiado. Por esto ha sido un gran reto hacer frente a esto y conseguir solucionar todos los problemas que iban surgiendo.

En definitiva, ha sido una gran experiencia el poder hacer este proyecto en el que he adquirido nuevos conocimientos y he podido enfrentarme al reto de hacer un sistema completo, aunque sin gran funcionalidad, desde cero.

9. Bibliografía

ARDUINO. <http://arduino.cc/> [Consulta: 20 de mayo de 2015]

ELECTRONILAB. *Sensor de movimiento PIR HC-SR501*. <http://electronilab.co/tienda/sensor-de-movimiento-pir-hc-sr501/>. [Consulta: 3 de mayo de 2015]

Introducing JSON. <http://json.org/>. [Consulta: 1 de junio de 2015]

ISSUU. *DHT11 Humidity & Temperature Sensor*. <http://issuu.com/rduinostar/docs/dht11_datasheet>. [Consulta: 3 de mayo de 2015]

LAJARA VIZCAINO, J. y PELEGRÍ SEBASTIÁ, J. (2014). Sistemas integrados con Arduino. Marcombo.

RICHARDSON, L. y RUBY. S. (2007). *RESTful Web Services*. O'Reilly.

MCEWEN, A. y CASSIMALLY, H. (2014). *Internet de las cosas: la tecnología revolucionaria que todo lo conecta*. Anaya.

MODMYPI. *PIR Infrared Motion Sensor (HC-SR501)* <http://www.modmypi.com/raspberry-pi/hacking-and-prototyping/sensors/pir-infrared-motion-sensor-hc-sr501->. [Consulta: 3 de mayo de 2015]

ORACLE. *Java Platform, Standard Edition 7 API Specification*. <http://docs.oracle.com/javase/7/docs/api/>. [Consulta: 25 de junio de 2015]

RESTLET Framework. <http://restlet.com/>. [Consulta: 25 de junio de 2015]

TORRENTE ARTERO, O. (2013). *Arduino: curso práctico de formación*. RC Libros.

UNIVERSIDAD POLITÉCNICA DE VALENCIA. *Rest vs Web Services*. <http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf>. [Consulta: 25 de junio de 2015]

W3SCHOOLS. <http://www.w3schools.com/>. [Consulta: 15 de junio de 2015]

WIKIPEDIA. <http://wikipedia.es>. [Consulta: 15 de junio de 2015]

Índice de ilustraciones

Índice de tablas

Índice de abreviaturas

Por orden de aparición.

HTML	HyperText Markup Language
CSS	Cascading Style Sheets
WoT	Web of Things
IoT	Internet of Things
TPC	Transmission Control Protocol
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
FTP	File Transfer Protocol
DOM	Document Object Model
AJAX	Asynchronous JavaScript And XML
MIT	Massachusetts Institute of Technology
GPL	General Public License
PHP	Hypertext Preprocessor
IDE	Integrated development environment
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
RPC	Remote Procedure Call
MIME	Multipurpose Internet Mail Extensions
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
XML	eXtensible Markup Language
JSON	JavaScript Object Notation
LDR	Light-Dependent Resistor
LCD	Liquid Crystal Display
JSP	JavaServer Pages
JSF	JavaServer Faces

API	Application Programming Interface
UML	Unified Modeling Language
CORS	Cross-Origin Resource Sharing