Examples

JsonConfigFile.ino

⚠ CAUTION: SLIPPERY FLOOR ⚠

The Arduino Library Manager **installs the ArduinoJson version 6 by default.**
However, using version 5 is highly recommended because version 6 is still in beta stage.

👉 Open the Arduino Library Manager and make sure that ArduinoJson version 5 is installed.

Description

This example shows how to store your project configuration in a file. It uses the SD library but can be easily modified for any other file-system, like SPIFFS.

The file contains a JSON document with the following content:

```
{
  "hostname": "examples.com",
  "port": 2731
}
```

Source code

```
// ArduinoJson - arduinojson.org
// Copyright Benoit Blanchon 2014-2018
// MIT License

#include <ArduinoJson.h>
#include <SD.h>
#include <SPI.h>

// Configuration that we'll store on disk
struct Config {
  char hostname[64];
  int port;
};

const char *filename = "/config.txt";  // <- SD library uses 8.3 filenames
Config config;                         // <- global configuration object

// Loads the configuration from a file
void loadConfiguration(const char *filename, Config &config) {
  // Open file for reading
  File file = SD.open(filename);

  // Allocate the memory pool on the stack.
  // Don't forget to change the capacity to match your JSON document.
  // Use arduinojson.org/assistant to compute the capacity.
  StaticJsonBuffer<512> jsonBuffer;

  // Parse the root object
  JsonObject &root = jsonBuffer.parseObject(file);

  if (!root.success())
    Serial.println(F("Failed to read file, using default configuration"));

  // Copy values from the JsonObject to the Config
  config.port = root["port"] | 2731;
  strlcpy(config.hostname,                   // <- destination
          root["hostname"] | "example.com",  // <- source
          sizeof(config.hostname));          // <- destination's capacity

  // Close the file (File's destructor doesn't close the file)
  file.close();
}

// Saves the configuration to a file
void saveConfiguration(const char *filename, const Config &config) {
  // Delete existing file, otherwise the configuration is appended to the file
  SD.remove(filename);

  // Open file for writing
  File file = SD.open(filename, FILE_WRITE);
  if (!file) {
    Serial.println(F("Failed to create file"));
    return;
  }

  // Allocate the memory pool on the stack
  // Don't forget to change the capacity to match your JSON document.
  // Use https://arduinojson.org/assistant/ to compute the capacity.
  StaticJsonBuffer<256> jsonBuffer;

  // Parse the root object
  JsonObject &root = jsonBuffer.createObject();

  // Set the values
  root["hostname"] = config.hostname;
  root["port"] = config.port;

  // Serialize JSON to file
  if (root.printTo(file) == 0) {
    Serial.println(F("Failed to write to file"));
  }
```

```
  // Close the file (File's destructor doesn't close the file)
  file.close();
}

// Prints the content of a file to the Serial
void printFile(const char *filename) {
  // Open file for reading
  File file = SD.open(filename);
  if (!file) {
    Serial.println(F("Failed to read file"));
    return;
  }

  // Extract each characters by one by one
  while (file.available()) {
    Serial.print((char)file.read());
  }
  Serial.println();

  // Close the file (File's destructor doesn't close the file)
  file.close();
}

void setup() {
  // Initialize serial port
  Serial.begin(9600);
  while (!Serial) continue;

  // Initialize SD library
  while (!SD.begin()) {
    Serial.println(F("Failed to initialize SD library"));
    delay(1000);
  }

  // Should load default config if run for the first time
  Serial.println(F("Loading configuration..."));
  loadConfiguration(filename, config);

  // Create configuration file
  Serial.println(F("Saving configuration..."));
  saveConfiguration(filename, config);

  // Dump config file
  Serial.println(F("Print config file..."));
  printFile(filename);
}

void loop() {
  // not used in this example
}
```

Classes used in this example

- `JsonBuffer`
- `JsonObject`

Functions used in this example

- `JsonBuffer::createObject()`
- `JsonBuffer::parseObject()`
- `JsonObject::operator[]`
- `JsonObject::printTo()`
- `JsonVariant::operator|`

See also

- `Mastering ArduinoJson` contains a more complex example with nested structures and that uses SPIFFS; see the Case Studies chapter.