Пишем плагин для Visual Studio Code. Теория и стандартный плагин

Опубликовано Jun 20, 2016 в «VS Code»

Эта статья продолжает небольшую серию «Создаём ваш первый плагин для…», в которую уже вошли статьи про написания плагина для Grunt и Gulp.

Дисклеймер

Я люблю JavaScript. Мне довольно приятно наблюдать за тем, что этот прекрасный язык программирования вышел за пределы браузера и собирает всё больше и больше областей применения. Так, например, благодаря Electron от GitHub у меня появилось сразу несколько приложений, которые я использую в повседневной жизни. К таким приложениям относится Hain, 1Clipboard, Wagon, Gitify и, конечно же, Visual Studio Code.

Теперь поговорим о приятном для некоторых людей и противном для других. У меня нет симпатий к TypeScript. На то есть свои причины, в основном, связанные с типизацией – не люблю я её, что тут поделать. При этом я не буду отрицать, что когда-нибудь начну использовать TypeScript или подобный язык, компилируемый в JavaScript – всё бывает, мнение может меняться со временем. Однако, в связи с этим, в статье все примеры будут написаны на JavaScript,

хотя сам редактор и вся документация к нему написана с применением TypeScript в примерах.

Кстати, я недавно узнал, что создателем TypeScript был Андерс Хейлсберг, который, оказывается, приложил руку к Turbo Pascal, Delphi и C#.

Что-то вместо введения

Герой нашего дня (VS Code) построен на Electron, который подробно рассматривался в статье «Построение Electron приложения. Введение». На момент написания статьи (июнь 2016) в основе редактора лежит Electron версии *0.37.6*, что подразумевает под собой Chromium 49-ой ветки и Node.js версии *5.10.0*. В репозитории на GitHub уже думают над переходом на новую версию Electron, где версия Chromium поднимется минимум до 51-ой ветки, а Node.js до версии *6.1.0* или выше. Всё это означает, что вы можете писать плагины, используя синтаксис ES2015 без Babel и его альтернатив, а также применяя любой API Node.js.

> **Предупреждение**
>
> Не стоит читать эту статью дальше введения, если вы не понимаете фишки, введённые в ES2015. Если говорить конкретно, то от вас требуется понимание деструктуризации, обещаний, стрелочных функций, const и let, а также понимание основ Node.js.

Итак, пожалуй, начнём с того, что плагины в VS Code изолированы от самого редактора и запускаются в отдельном хост-процессе (**extension host process**), который представляет собой процесс Node.js с возможностью использования VS Code API. Такой подход не позволяет плагинам влиять на производительность редактора при его запуске или в процессе его работы. Для пользователя это означает, что редактор не зависнет на время выполнения задач каким-либо плагином или, если плагин выдаст фатальную ошибку.

Для экономии расхода памяти разработчики также добавили ленивую загрузку плагинов. Это означает, что плагины активируются лишь в тот момент, когда они нужны. Например, если пользователь открывает Markdown-файл, то плагины, работающие с Markdown, будут загружены только в момент открытия файла. Разумеется, мы должны сообщить редактору о том, когда именно он должен

активировать какой-либо плагин. О настройке ленивой загрузки мы поговорим позднее в разделе про файл манифеста.

Помимо всего прочего, плагины делятся на три вида в зависимости от функционала:

К первому виду относятся стандартные плагины, которые запускаются в хост-процессе, активируются в нужный момент и не требуют серьёзных вычислений.

Ко второму виду относятся так называемые **«языковые серверы»**. Клиентская часть плагина запускается в хост-процессе, а серверная часть создаёт дополнительный процесс, в котором производятся все сложные вычисления. К такому виду плагинов относятся линтеры.

К третьему виду относят **«службы отладки»**, которые пишутся в виде отдельной программы и взаимодействуют с VS Code по специальному протоколу CDP (VS Code Debug Protocol).

В этой статье будет рассматриваться лишь первый вид плагинов на примере vscode-lebab. Во второй статье разбирается процесс построения второго вида плагинов на примере vscode-puglint.

Манифест плагина

Написание плагина начинается не с кода, а с файла манифеста, которым в мире Node.js является файл `package.json`. VS Code дополняет стандартный файл манифеста своими полями. Ниже будут рассмотрены самые основные из них.

publisher [string]

Имя пользователя, под которым вы зарегистрировались в vsce.

icon [string]

Путь до иконки, которая будет отображаться в магазине расширений. Размер иконки 128x128 пикселей. Также поддерживается SVG формат.

displayName [string]

Название плагина, которое будет отображаться в магазине расширений.

categories [array]

Массив, содержащий имена категорий, к которым относится плагин. Доступны следующие категории: `[Languages, Snippets, Linters, Themes, Debuggers, Other]`. Пожалуйста, указывайте категорию или категории обдуманно. Например, если ваше расширение включает в себя подсветку синтаксиса языка и сниппеты, то указывайте только эти две категории.

galleryBanner [object]

Настройки оформления страницы расширения в магазине. Используется для того, чтобы иконка расширения и фон подложки были контрастны. Свойство `color` отвечает за цвет фона, свойство `theme` за цвет шрифта: `dark` – белый, `light` – чёрный.

```
"galleryBanner": {
    "color": "#0000FF",
    "theme": "dark"
}
```

preview [boolean]

Флаг, позволяющий пометить плагин сообщением о том, что он доступен в режиме предварительного просмотра.

activationEvents [array]

Массив событий, в случае наступления которых плагин будет активирован редактором. Поддерживаются следующие события, активирующие плагин в том случае, если будет:

- `onLanguage` – открыт файл указанного языка (не расширения).
- `onCommand` – вызвана указанная команда.
- `onDebug` – запущен сеанс отладки указанного типа.
- `workspaceContains` – найден указанный файл в корневой папке проекта.

Отдельно стоит отметить событие `*`, которое активирует плагин при загрузке редактора. Однако, использовать это событие нужно крайне редко и в том случае, если комбинации других событий не могут решить сложившуюся проблему.

contributes [object]

С помощью этого поля в `package.json` разработчик описывает своё расширение. Доступно довольно большое количество всевозможных полей, описывающих:

- `configuration` – поля, доступные пользователю в настройках редактора.
- `commands` – команды, доступные пользователю в палитре команд `F1`.
- `keybindings` – сочетания клавиш для вызова команд.
- `languages` – языки.
- `debuggers` – отладочный адаптер.
- `grammars` – TextMate-грамматику, необходимую для подсветки синтаксиса.
- `themes` – темы.
- `snippets` – сниппеты.
- `jsonValidation` – схемы проверки определённых JSON-файлов.

Несложно догадаться, что поля необходимо указывать в зависимости от типа вашего расширения. Если это плагин, сортирующий строки, то, вероятнее всего, вы будете использовать поля: `configuration`, `commands` и `keybindings`.

extensionDependencies [array]

Массив идентификаторов расширений, которые требуются для работы плагина. Например, если ваше расширение требует поддержки синтаксиса C#, то необходимо будет добавить в массив строку `vscode.csharp`, где `vscode` – ник опубликовавшего расширение, а `csharp` – имя расширения.

Немного про VS Code API

Как и полагается любому крупному проекту, VS Code имеет довольно обширный API, доступный разработчикам плагинов. Рассмотрим лишь так называемые пространства имён:

- **commands** – это пространство имён для работы с командами. С помощью доступных методов разработчик может регистрировать, получать и выполнять команды.
- **env** – пространство имён, содержащее описание переменных окружения редактора при запуске. С помощью соответствующих методов можно получить имя окна редактора, его язык, идентификатор редактора в системе и идентификатора сессии редактора, которые устанавливается при запуске.
- **extensions** – пространство имён для работы с установленными расширениями. С помощью этого API можно получить все или конкретные расширения, известные редактору.
- **languages** – пространство имён, позволяющее получить доступ к языковым возможностям редактора, например, к IntelliSense, подсказкам, а также функциям диагностики кода для линтеров.

- **window** – пространство имён для работы с текущим окном редактора. Доступно API для работы с видимыми и активными окнами редактора, а также элементами пользовательского интерфейса. Последнее подразумевает под собой возможность отображения различных сообщений, ввода текста или выбора каких-либо вариантов.
- **workspace** – пространство имён для работы с текущей рабочей областью, включая открытую директорию и файлы. С помощью этого API осуществляется вся работа с содержимым открытого файла.

Пишем стандартный плагин

Постановка задачи

В этой части статьи я буду описывать процесс написания плагина для VS Code на основе lebab, который автоматически конвертирует JavaScript-код, написанный по стандарту ES5 в ES2015. Это проект является альтернативой проекту Babel, но в обратную сторону.

Манифест

И снова скажу, что написание плагина начинается не с кода на JavaScript, а с манифеста. Первым делом создаём файл `package.json` и пишем туда пару десятков строк, описывающих плагин. Полный листинг манифеста вы сможете найти в репозитории плагина vscode-lebab. Остановимся именно на тех моментах, которые касаются работы с VS Code.

Во-первых, укажем информацию, которая будет отображаться в маркете:

```
{
  "displayName": "lebab",
  "publisher": "mrmlnc",
  "icon": "icon.png",
  "homepage": "https://github.com/mrmlnc/vscode-lebab/blob/master/
  "categories": [
    "Other"
  ]
}
```

Во-вторых, укажем массив событий, на которые наш плагин должен откликаться. Про команду `lebab.convert` я расскажу немного позднее.

```
{
  "activationEvents": [
    "onCommand:lebab.convert"
  ]
}
```

В-третьих, опишем плагин. Предполагается, что пользователю будет доступна лишь одна команда, по вызову которой он получит сконвертированный ES5-код в синтаксис ES2015. Также предполагается, что в настройках редактора пользователь сможет указать, что именно он хочет конвертировать с помощью lebab. Для этого я определил опцию `lebab.transforms`, содержащую объект ключей, с которыми будет работать конвертер.

```
{
  "contributes": {

    "commands": [{
      "command": "lebab.convert",
      "title": "Lebab: convert JavaScript code from ES5 to ES2015"
    }],

    "configuration": {
      "type": "object",
      "title": "Lebab configuration",
      "properties": {
        "lebab.transforms": {
          "type": "object",
          "default": {},
          "description": "Convert your old-fashioned code with a s
        }
      }
    }

  }
}
```

Убираем из маркета лишнее

Сколько раз не говори, но я всё равно встречаю модули в npm, у которых вместе с кодом я получаю файлы тестов, изображений и прочей лабуды. К счастью, теперь я могу ссылаться на эту статью в отношении npm. В случае VS Code, необходимо создать файл `.vscodeignore`, который действует так же, как и файл `.gitignore`, но в отношении маркета расширений.

У меня файл `.vscodeignore` имеет следующее содержимое:

```
.vscode/**
typings/**
test/**
.editorconfig
.gitignore
.travis.yml
jsconfig.json
```

Я очень прошу, убирайте всё лишнее из плагина, иначе я вас вычислю по IP и накажу.

Базовый код

В мире Node.js принято писать код модуля в файле `index.js`, а приложения – `app.js`. Мир VS Code тоже имеет традиции, и код плагина пишется в файле `extension.js`.

> **Внимание**
>
> Если очень хочется писать код в файле с именем, отличным от `extension.js`, то, как и в мире Node.js, вам придётся указать имя файла в поле `main` в манифесте.

Для начала определим две функции. Первая функция будет иметь имя `activate` и вызываться в том случае, если плагин был активирован событием, указанным в манифесте. Вторая функция имеет имя `deactivate` и вызывается в том случае, если плагин был деактивирован. Под деактивацией следует понимать последействие команды, а не удаление плагина. Её предназначение в большинстве плагинов излишне, поэтому она не обязательна. Далее в статье я не буду упоминать функцию деактивации плагина.

```
const vscode = require('vscode');

function activate(context) {
  // Code...
}

exports.activate = activate;

function deactivate() {
  // Code...
}

exports.deactivate = deactivate;
```

Напомню, что в файле манифеста была указана команда `lebab.convert` – самое время её зарегистрировать. Для регистрации команд существует два метода:

- `registerTextEditorCommand` – регистрирует команду в контексте текстового редактора или файла.
- `registerCommand` – регистрирует команду в глобальном контексте, то есть вне зависимости от наличия открытого редатора с текстом.

Второй метод используется крайне редко, вследствие того, что, в основном, плагины нацелены на работу с содержимом текстового редактора.

В конце все объявленные команды должны быть добавлены в массив `subscriptions`.

```
'use strict';

const vscode = require('vscode');

function activate(context) {
  const convert = vscode.commands.registerTextEditorCommand('lebab
    // ...
  });

  context.subscriptions.push(convert);
}

exports.activate = activate;
```

При регистрации команды необходимо передать идентификатор команды, указанный в файле манифеста и функцию обратного вызова, в которую передаётся объект `textEditor`, содержащий всю информацию о текущем редакторе, такую как:

- Файл на диске или новый файл (untitled)
- Путь до файла и его имя
- Идентификатор языка
- Наличие EOL
- Если было выделение текста, то параметры этого выделения
- Статистика текста (количество символов в строке и прочее)
- Версия файла (проще говоря, номер сохранения в истории файла)
- Строки файла
- и т.д.

Чисто практически, вы можете обращаться к свойствам объекта и работать с полученными из него данными. Но, разумеется, лучше всего обращаться к этому объекту с помощью определённых методов. На данном этапе нам нужно получить текст файла, чтобы в дальнейшем обработать его, и настройки, определённые в редакторе для нашего плагина.

Получить текст открытого документа можно, используя метод `getText`, а настройки, используя метод `getConfiguration` у `workspace` API:

```
const convert = vscode.commands.registerTextEditorCommand('lebab.c
  // Обычный объект, где имена свойств совпадают с теми, что были
  const options = vscode.workspace.getConfiguration('lebab');

  // Текст открытого файла.
  const text = textEditor.document.getText();
});
```

> **Внимание**
>
> Настройки редактора нужно получать в момент вызова команды, иначе, если получить их в момент активации плагина, то при обновлении настроек редактора, объект в переменной не обновится.

Далее я не буду рассматривать процесс вызова Lebab, потому что это элементарное действие, не относящееся к VS Code. Покажу лишь тот участок,

что отвечает за вставку обработанного текста обратно в окно редактора. Для этого мы обратимся к объекту `textEditor` и вызовем метод `edit` с коллбэком, представленным ниже. Конечно же, код должен располагаться после получения текста документа и настроек редактора в функции обратного вызова регистрации команды.

```
textEditor.edit((editBuilder) => {
  // Получаем текущий документ.
  const document = textEditor.document;

  // Получаем последнюю строку документа.
  const lastLine = document.lineAt(document.lineCount - 1);

  // Создаём нулевую позицию, то есть начало документа, где первый
  // строки, а второй ноль — номер символа в строке.
  const start = new vscode.Position(0, 0);

  // Создаём завершающую позицию, где первое число — последняя стр
  // а второе — номер последнего символа в строке.
  const end = new vscode.Position(document.lineCount - 1, lastLine

  // Создаём диапазон, используя специальное API.
  const range = new vscode.Range(start, end);

  // Заменяем текст в обозначенном диапазоне на что-либо.
  editBuilder.replace(range, text);
});
```

Собственно, это всё, что требуется сделать в обычном плагине для VS Code: получить текст, обработать его и вернуть обратно. Полный листинг кода содержит 52 строки и размещён на GitHub в репозитории vscode-lebab.

Как можно заметить, ничего сложно — простейший JavaScript-код, который разбавлен вызовами необходимых API редактора. В этой статье был рассмотрен простейший случай, когда у вас есть готовое решение и его нужно подружить с редактором. В случае, если готового решения нет, то лучше оформить его в виде npm-пакета по всем канонам (тесты, документация), а уже потом подружить его с редактором. В качестве примеров, если решите писать плагин на JavaScript, вы можете посмотреть код следующих плагинов:

- vscode-lebab
- vscode-attrs-sorter

- vscode-csscomb
- vscode-postcss-sorting
- vscode-stylefmt

Что-то вместо вывода

В этой статье я помог вам начать писать плагины для VS Code. Многое осталось за кулисами и не рассматривалось, однако вам в любом случае придётся обращаться к документации. Считайте, что эта статья преследует цель показать, что писать плагин для VS Code довольно просто, причём необязательно делать это на TypeScript. Хотя, при этом не стоит забывать, что TypeScript – это всё тот же JavaScript.

Также, советую посмотреть на код, представленный в репозитории VSCode-Sample. Здесь автор собрал примеры взаимодействия с UI редактора, которые, возможно, помогут вам освоиться.

Если вам интересна тема разработки плагинов для VS Code, то добро пожаловать во вторую часть этой статьи, в которой рассматривается процесс написания плагина, использующего языковой сервер.

Что почитать?

- Конечно, документацию:
  - Обзор экосистемы расширений
  - Обзор API
  - Про публикацию расширений
  - Про утилиты, упрощающие создание расширений
  - Примеры
- Visual Studio Code Extensions: Editing the Document
- Getting Input and Displaying Output in a Visual Studio Code Extension

---

Сделано с ❤ @mrmlnc