Search

Introduction to OpenShift Applications

Tell us what you think!

Congratulations on completing at least 25% of the course. We would appreciate feedback on your learning experience. Click below to fill out a brief survey.

TAKE ME TO THE SURVEY (HTTPS://SURVEY.OLE.REDHAT.COM/SURVEY?COURSE=DO101&MODALITY=COURSE&OFFERING=101)

NOT RIGHT NOW

Don't ask again

TRANSLATIONS

➜ NEXT (/ROL/APP/COURSES/DO101-4.2/PAGES/CH01S02)

Chapter 1. Configuring a Cloud Application Developer Environment

Abstract

Goal	Configure a developer environment with a modern integrated developer environment and version control.

Objectives	• Edit application source code with Visual Studio Code (VS Code). • Create a Git repository. • Use version control to collaborate and manage application source code.
Sections	• Developing Applications with Visual Studio Code (and Guided Exercise) • Initializing a Git Repository (and Guided Exercise) • Managing Application Source Code with Git (and Guided Exercise)

Developing Applications with Visual Studio Code

Objectives

After completing this section, you should be able to edit application source code with Virtual Studio Code (VS Code).

Integrated Development Environments

Software developers execute many different types of tasks during the software development life cycle of an application:

- compile and build the source code
- correct syntax errors
- debug runtime errors
- maintain version control changes to the source code
- refactor source code
- execute source code tests

In the past, developers used a collection of separate tools, such as editors, compilers, interpreters, and debuggers, to develop software applications. Inefficiencies arose from using separate tools, leading to the creation of **Integrated Development Environments**, or IDEs. IDEs improve developer productivity by integrating common software development tools into a single

application. IDEs often integrate:

- language-specific editors
- code completion capabilities
- syntax highlighting
- programming language documentation
- code debugging
- source control management tools, such as Git, SVN, or Mercurial

Many modern IDEs support multiple programming languages. Using these IDEs, developers create applications in a variety of different languages, without needing to learn language-specific tooling, such as compilers and interpreters.

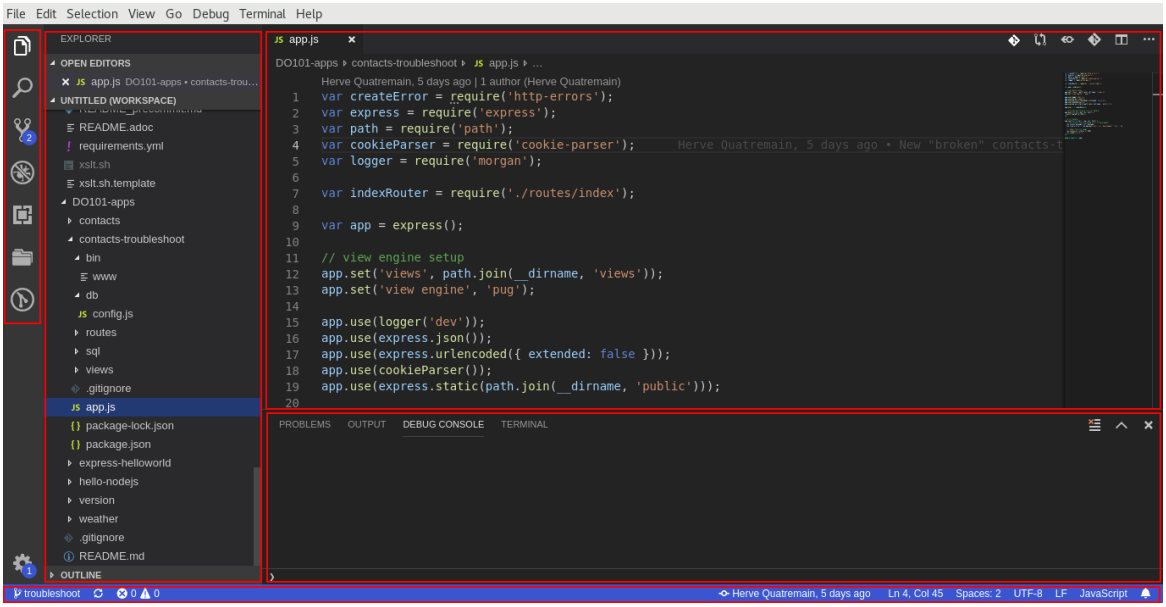
Developing Software with VS Code

VS Code is a popular open source IDE that supports multiple languages, including JavaScript, Python, Go, C#, TypeScript, and more. It provides syntax highlighting, code completion, debugging, and code snippet capabilities, along with a plug-in framework that allows you to install additional functionality from a plug-in marketplace.

In this course, you use VS Code to create, edit, and manage source code projects.

Overview of the VS Code Interface

The VS Code interface contains five primary components:



1. **The Activity Bar**. Located at the far-left, it contains shortcuts to change the view of the Side Bar. By default, the Activity Bar contains shortcuts for the Explorer, Search, Source Control, Debug, and Extensions views. You can also access these activity views from the `View` menu.

2. **Side Bar**. Located immediately to the right of the **Activity Bar**, this area displays the selected Activity view, such as the Explorer view.

3. **Editor Groups**. The top-right region of VS Code contains one or more groups of editors.

   By default, there is only one editor group. Any active editor takes up the entire region of the editor group.

   Click the `Split Editor Right` icon in the upper-right to create a second editor group. With two editor groups, you can edit two different files side by side.

4. **Panels**. Located below the editors, individual panels provide different output or debugging information for activities in VS Code. By default, four panels are provided:
   - Problems
   - Output
   - Debug Console
   - Terminal
5. **Status Bar**. Located at the bottom, this area provides information about the open project and files as you edit.

VS Code Workspaces

VS Code organizes a collection of related software projects and configuration settings in a **workspace**. Configuration data for each workspace is stored in a file with a `.code-workspace` extension.

From the `File` menu, you can close, save, and open a workspace.

For example, consider a web application, which is named `myapp`, with the following components:

- JavaScript code for the front end user interface of the application.
- Python code for the back-end application logic.
- Configuration and scripts for the application database.
- AsciiDoc files for the application documentation.

If you maintain each of these components in separate code repositories or folders, then you can add each folder to a `myapp` VS Code workspace that you dedicate for the application:

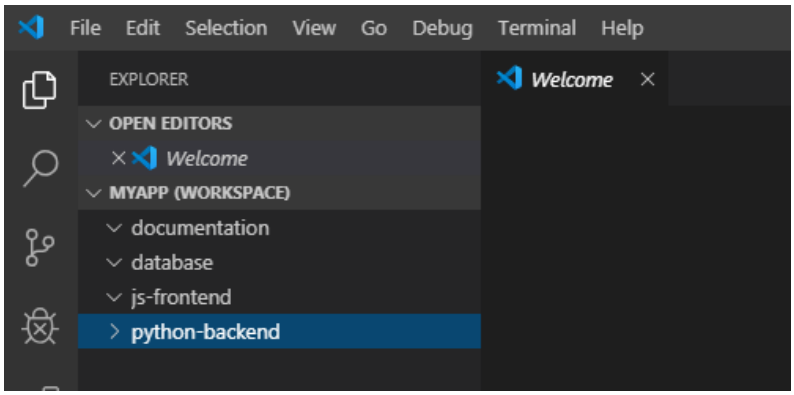


Figure 1.1: A VS Code workspace with multiple source code folders

To add a source code folder to a workspace, click **File** → **Add Folder to Workspace...**.

To remove a source code folder from a workspace, access the Explorer view (**View** → **Explorer**). Right-click a selected top level workspace folder, and then select `Remove Folder from Workspace` to remove the folder from your workspace.

VS Code Integrated Terminal

Although VS Code integrates many development tools, you might need access to external development tools or applications. VS Code integrates the terminal from your operating system, enabling you to execute arbitrary terminal commands in VS Code. To view the integrated terminal, click **View** → **Terminal**.

You can open multiple terminals in the VS Code terminal panel. The terminal panel contains a list of open terminals. To display a terminal, select it from the list. To close a terminal, click **Kill Terminal**.

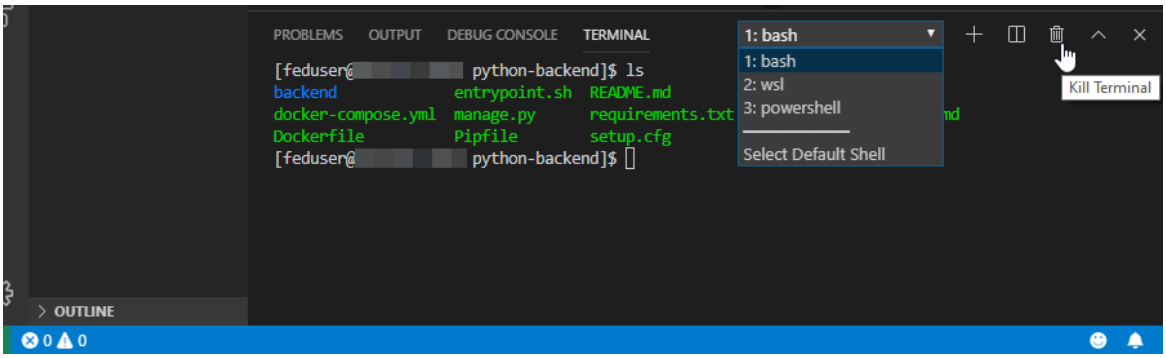


Figure 1.2: The VS Code integrated terminal

VS Code Extensions

Although the VS Code integrated terminal aids arbitrary command execution, you must install and learn to use any needed external commands. Additionally, terminal commands do not take advantage of common usage patterns in VS Code.

VS Code provides an extension framework to encourage the integration of software development capabilities into VS Code. Any user or organization can contribute extensions to VS Code. After an extension is developed, it is advertised and published on the VS Code marketplace.

You can search, download, and install extensions from the VS Code marketplace in VS Code. Click **View** → **Extensions** to access the Extensions view in the Side Bar.

> **NOTE**
>
> In this course, you do not install any additional extensions for VS Code.

Developing a Node.js Application Using VS Code

Many of the example web applications in the course exercises are Node.js applications. If an exercise in this course requires you to edit code, then use VS Code to make the changes.

Node.js is an open source runtime engine that executes JavaScript outside of a browser. It is designed to efficiently handle many concurrent connections for network applications. Additionally, Node.js enables you to write both front end and back end code in one language, JavaScript. For these reasons, Node.js is a popular runtime engine for web application development.

Installing Node.js

To install Node.js, navigate to https://nodejs.org/en/download (https://nodejs.org/en/download) in a browser. Click the appropriate link for your operating system, and then follow the instructions.

After you install Node.js, you use the `node` command to execute Node.js applications.

Node.js Modules

All modern programming languages support code reuse through shared libraries, packages, and modules.

Modules are the smallest unit of reusable code in Node.js. Node.js provides several built-in modules. You can also download and use third party Node.js modules.

When you create a complex Node.js application, you write custom Node.js modules to group related logical code. Your custom Node.js modules are defined in JavaScript text files.

Use the `require` keyword to load a Node.js module. Consider the following example:

```
var http = require('http');
var mymodule = require('./mymodule');
```

The `http` variable contains a reference to the built-in `http` module, while the `mymodule` variable contains a reference to the module defined in the `mymodule.js` file.

Node.js Packages

Like other programming languages, Node.js provides a way to define and manage application dependencies. A Node.js application dependency is called a **package**. A package is collection of one or more Node.js modules, or application code, that you download from a Node.js package repository.

Application dependencies are defined in the `packages.json` file, located at the root of the Node.js project folder. An example `packages.json` file follows:

```
{
    "name": "hello",
    "version": "0.1.0",
    "description": "An example package.json",
    "author": "Student <student@example.com (mailto:student@example.com)>",
    scripts: {
        "start": "node app.js"
    },
    "dependencies": {
        "cookie-parser": "1.4.*",
        "http-errors": ">=1.6.3",
    },
    "license": "MIT"
}
```

In this example, the `hello` application requires specific versions of the `cookie-parser` and `http-errors` packages.

The `packages.json` file also defines other metadata for a Node.js application, such as the name, version, and author of the application.

The Node Package Manager

The Node Package Manager (NPM) is a command line tool used to create, install, and publish Node.js packages. For most operating systems, the `npm` command (short for Node Package Manager) is installed as part of the Node.js installation process.

To install the dependencies for a Node.js application, use the `npm install` command.

To initialize an empty directory as a Node.js project, use the `npm init` command to create a `packages.json` file for a new Node.js application.

To manage the application life cycle, use the `npm` command to start, stop or restart the application.

The Express Web Application Framework

Express is a common Node.js framework that aims to simplify the creation of web services. Because Express is a Node.js package, use the `npm install` command to install Express:

```
$> npm install express
```

After installing the `express` Node.js package, the `express` command is available on your system. Use the `express` command to create a set of initial project files for a new Express application.

Consider the example that follows:

```
$> express /path/to/project/folder/myapp
```

The command creates a `myapp` folder that contains a `packages.json` file:

```
{
  "name": "myapp",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "express": "~4.16.1",
    "http-errors": "~1.6.3",
    "jade": "~1.11.0",
    "morgan": "~1.9.1"
  }
}
```

The `express` package is a dependency for the `myapp` Node.js application. Start the `myapp` application with the `node ./bin/www` command.

The source code in the `./bin/www` file loads the`app` Node.js module:

```
...output omitted...
/**
 * Module dependencies.
 */

var app = require('../app');
...output omitted...
```

The `app` module source code is contained in the `app.js` file, located in the root of the `myapp` project directory. The `app.js` file contains the primary application logic.

The `app.js` file for a simple "Hello, World!" Express application contains the following:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello, World!\n');
});

app.listen(8080, function () {
  console.log('Example app listening on port 8080!');
});

module.exports = app;
```

The `app` variable references an instance of an Express application. The application is configured to listen for requests on port 8080. When you access the application endpoint, the applications sends a response of `Hello, World!`.

REFERENCES

Integrated Development Environment - Wikipedia (https://en.wikipedia.org/wiki/Integrated_development_environment)

For more information about Visual Studio Code workspaces and project folders, refer to the Visual Studio Code documentation at https://code.visualstudio.com/docs/editor/multi-root-workspaces (https://code.visualstudio.com/docs/editor/multi-root-workspaces)

For more information about Visual Studio Code integrated terminal, refer to the Visual Studio Code documentation at https://code.visualstudio.com/docs/editor/integrated-terminal (https://code.visualstudio.com/docs/editor/integrated-terminal)

For more information about Node.js modules, refer to the Node.js documentation at https://nodejs.org/api/modules.html#modules_modules (https://nodejs.org/api/modules.html#modules_modules)

➜ NEXT (/ROL/APP/COURSES/DO101-4.2/PAGES/CH01S02)