# Design Decisions

## Frontend

### Angular

For our frontend framework, we chose Angular 4. Angular 4 is extensible, and it can be used in web, mobile, and desktop applications. Angular 4 has some powerful testing frameworks such as jasmine or mocha. As a final reason as to why we chose Angular 4, because it is actively maintained.

### Jasmine

We chose jasmine to test our framework because it supports testing front-end code, and works very well with Angular.

## Backend

### Docker

We chose Docker because it helps tremendously with security. Docker essentially creates lightweight VMs (containers) where we can add whatever dependencies, software, etc. to that container. The student's code is restricted to that container, and cannot access files outside of it, which eliminates a security concern for us. We can also safely destroy Docker containers should the process take too long, and errors that occur within a Docker container do not affect the rest of the system.

Docker is also flexible. Any number of dependencies can be downloaded and used to create an image. We utilized this flexibility to easily create images to run any coding language. If a professor creates a python assignment, we can create a container that runs and compiles the python language. We can also do this with java, and this flexibility allows for additional language support to be added with relative ease.

### Spring

We chose Spring because it is a Java-based framework commonly used for developing web applications. Spring allows the development team to move as quickly as possible because of its capabilities for configuration and dependency injection. A Java-based framework also works well for the team as all developers are well versed in Java development.
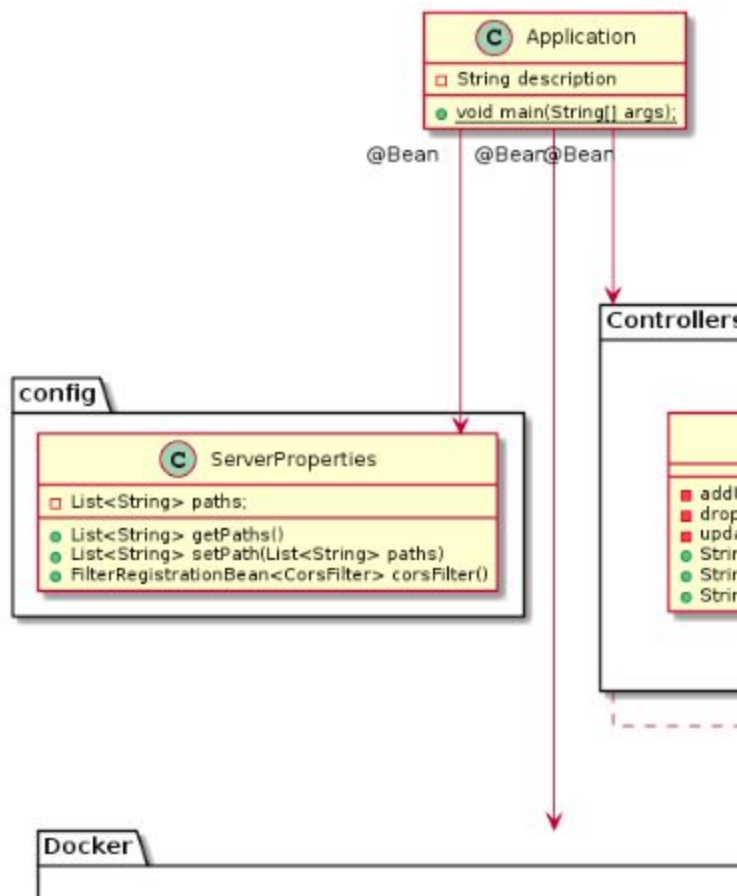
# Database

## MongoDb

MongoDB is a object-oriented NoSQL database. MongoDB was the right choice for Compile.io because of how flexible and extensible it is. In addition to being a powerful database, if the clients wish to extend the application they can do so with ease.

# Design Pattern Decisions

## Dependency Injection



One Example of Dependency Injection is how our application class runs our different packages in the backend. Spring has a annotation named @bean that connects one piece of code to another through dependency injection. For example Spring runs our Docker package with just 3
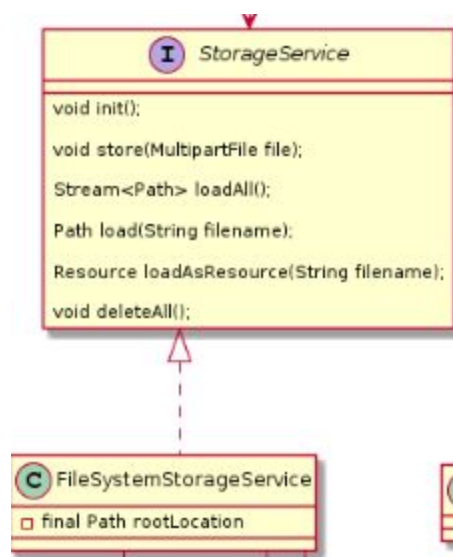
lines of code annotated with @bean, the same goes for the controllers. All of the arrows in the above picture are dependency injection arrows through the @bean property part of Spring.
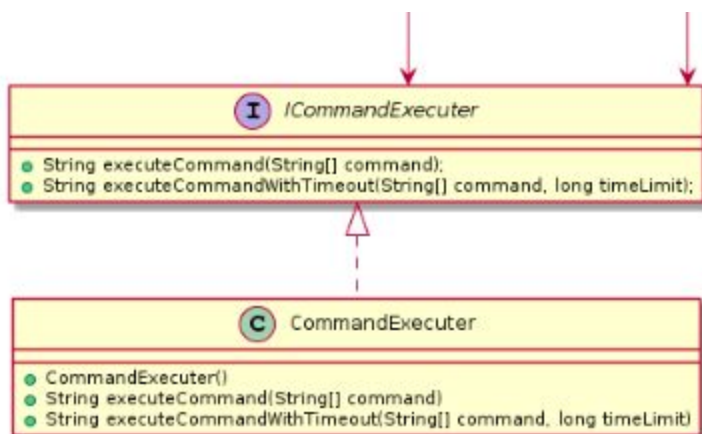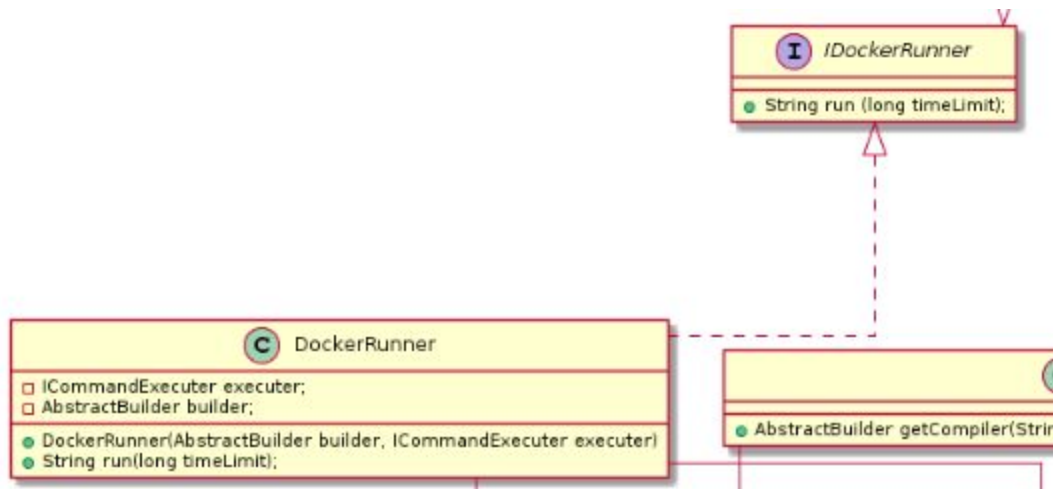
## Single Responsibility Principle
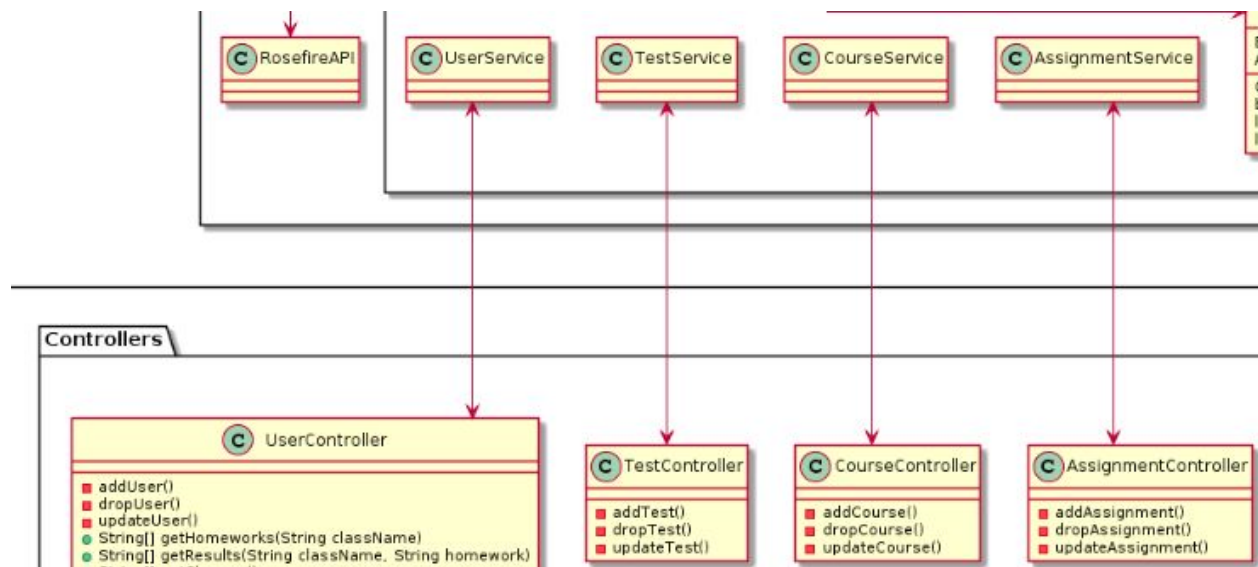


This Design decision for having one frontend service talk to one backend controller falls most closely with the single responsibility principle. This principle is prevalent in our software, as we have partitioned each class to do only one thing, but is easiest to see in the above picture. Users have their own service and controller, as do Tests and so on and so forth.

## Composition over inheritance

## IDockerRunner
*(I)* *IDockerRunner*

● String run (long timeLimit);

## DockerRunner
*(C)* DockerRunner

□ ICommandExecuter executer;
□ AbstractBuilder builder;

● DockerRunner(AbstractBuilder builder, ICommandExecuter executer)
● String run(long timeLimit);

● AbstractBuilder getCompiler(Strir

## ICommandExecuter
*(I)* *ICommandExecuter*

● String executeCommand(String[] command);
● String executeCommandWithTimeout(String[] command, long timeLimit);

## CommandExecuter
*(C)* CommandExecuter

● CommandExecuter()
● String executeCommand(String[] command)
● String executeCommandWithTimeout(String[] command, long timeLimit)

The above three snippets of our UML are examples of Programming to an interface or following the principle of composition over inheritance. This helps with code reuse which maximizes the benefit of using an object-oriented programming language such as java.

Builder Pattern
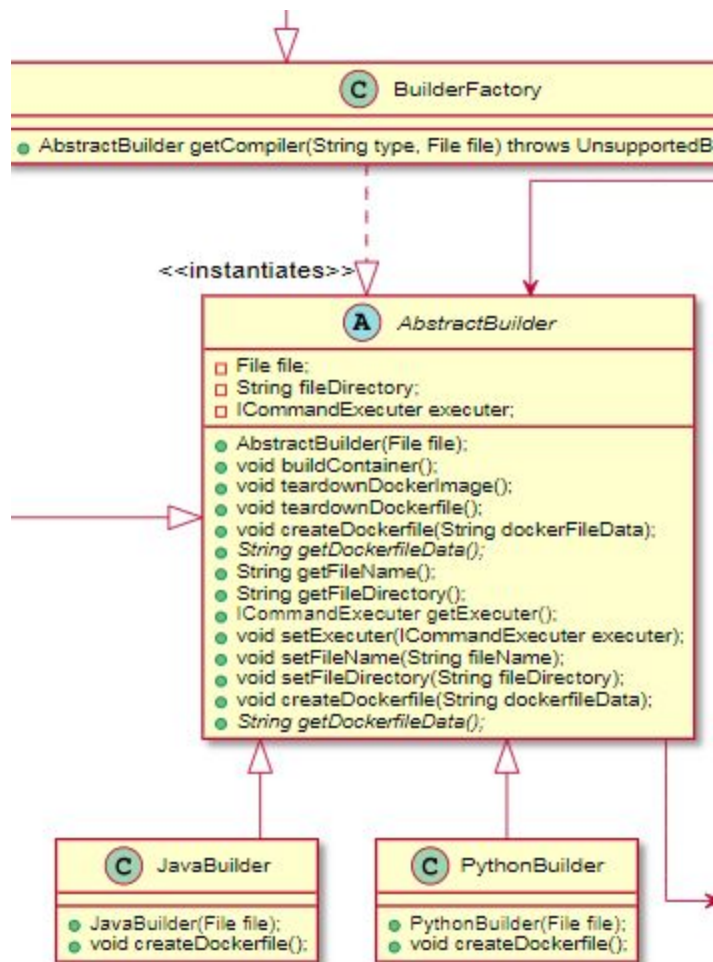


This Factory pattern was used because it allows us to flexibly add more language support for Compile.io on the backend. We started off by using a Simple Factory Pattern, which contained a nested `if` statement. So, if a programmer wanted to add another return statement in the Factory, they'd have to go in an make modifications to legacy code. This violates the open/close principle. In order to move away from this, we will be moving to an Abstract Factory Pattern in the future, coming in the next release (release 3.0). This is more flexible than a simple factory, as it allows developers to add return values to the factory, without violating the open/close principle. This is accomplished via a Map, which programmers can add keys and values to, thus expanding the factory. In this scenario, add a new Map entry would be like extending the nested `if` statement from the Simple Factory Pattern.

# Security

Globally apply CORS to application
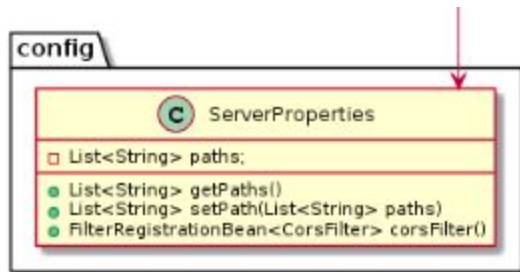


Anyone within the rose-hulman network could have been able to access our server, and our code if it wasn't for our globally configured Cors (Cross-origin resource sharing) filter. This Filter is run through dependency injection into our application class in the ServerProperties class, which also initializes the server to accept localhost when we are running in Development, or to our VM's if we are running in production. Another feature that this class has to offer is that it is directly tied to the properties folders and builds a "configuration file" every time we run the app.