### 

[Prerequisites](#_cmcu0n8chbkc)

[Use PCF Dev (Use this only in case you have some problems with Pivotal web services)](#_uyrjdmy8l3rh)

[I. Codebase](#_jmzty9puvb03)

[One codebase tracked in revision control, many deploys](#_du0lao3d7u6f)

[II. Dependencies](#_i4q263i9lavm)

[Explicitly declare and isolate dependencies](#_szm5ux9d9jmr)

[III. Config](#_jpglhx9z99zd)

[Store config in the environment](#_rjsd4mao3fvr)

[IV. Backing Services](#_4mun2z2mcna1)

[Treat backing services as attached resources](#_j2l6tdpp4m1f)

[V. Build, release, run](#_b1i92vpmj1z)

[Strictly separate build and run stages](#_ga6v8j37iwz2)

[VI. Processes](#_e79uh29pqy2e)

[Execute the app as one or more stateless processes](#_msj7may92h5t)

[Introduce a stateful process, that violates the rule](#_8mtrf9ordvyt)

[Fix stateful process](#_hvhg49n6hvvm)

[VIII. Concurrency](#_3vy2h2bnu11n)

[Scale out via the process model](#_6kuf21mb1cp7)

[IX. Disposability](#_je33un48i82g)

[Maximize robustness with fast startup and graceful shutdown](#_gdhsijtsw646)

[X. Dev/prod parity](#_plu9ksxpnkuk)

[Keep development, staging, and production as similar as possible](#_91cufpcjx7y3)

[XI. Logs](#_pex3d2l24f74)

[Treat logs as event streams](#_i630i7jhubtn)

[XII. Admin processes](#_yowy2np8l6lo)

[Run admin/management tasks as one-off processes](#_d0vx4jmyp939)

### 

### 

### **Prerequisites**

1. Download and install the Cloud Foundry CLI: https://goo.gl/okTayH
2. Check required dependencies

|  |
| --- |
| $ mvn --version  $ git --version  $ cf --version |

1. Register at https://run.pivotal.io/
2. Login using cf cli.

|  |
| --- |
| $ cf login -a api.run.pivotal.io  $ cf target -o YOUR\_ORG -s YOUR\_SPACE |

##### 

##### **Install PCF Dev**

We are not going to use this installation right now, but we need it later. PCF Dev download can take a very long time, so you need to start this process as early as possible.

1. Install VirtualBox https://www.virtualbox.org/wiki/Download
2. Sign in or sign up to https://network.pivotal.io/
3. Download and install the PCFDev CLI plugin: https://network.pivotal.io/products/pcfdev
4. Start up the PCFDev environment
5. Connect to your PCFDev environment (use cf login command)

### 

### **I. Codebase**

##### **One codebase tracked in revision control, many deploys**

A twelve-factor app is always tracked in a version control system, such as Git, Mercurial, or Subversion. A copy of the revision tracking database is known as a code repository, often shortened to code repo or just repo.

*There is always a one-to-one correlation between the codebase and the app:*

* If there are multiple codebases, it’s not an app – it’s a distributed system. Each component in a distributed system is an app, and each can individually comply with twelve-factor.
* Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the dependency manager.

1. Create a repository on github https://github.com/YOUR\_ACCOUNT\_NAME/cloud-native-workshop
2. Clone the repository

|  |
| --- |
| $ git clone https://github.com/YOUR\_ACCOUNT\_NAME/cloud-native-workshop |

### 

### 

### **II. Dependencies**

##### **Explicitly declare and isolate dependencies**

A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a dependency declaration manifest. Furthermore, it uses a dependency isolation tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development.

In Java we use Maven and Gradle.

1. Generate project (replace YOUR\_USERNAME with your github username)

|  |
| --- |
| $ mvn archetype:generate -DgroupId=com.github.YOUR\_USERNAME.app -DartifactId=12f\_app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false |

1. The folder structure should be the following

|  |
| --- |
| ├── cloud-native-workshop/  │ ├── 12f\_app/  │ │ ├── pom.xml  │ │ ├── src  │ │ │ ├── main  │ │ │ │ ├── java  │ │ │ │ │ ├──{project-packaging}  │ │ │ │ │ │ ├──App.java  │ │ │ ├── test  │ │ │ │ ├──java  │ │ │ │ │ ├──{project-packaging}  │ │ │ │ │ │ ├──AppTest.java |

1. Add spring boot parent element

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.8.RELEASE</version>  </parent> |

1. Add spring boot dependency

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> |

1. Add spring boot build plugin

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> |

1. Delete App.java, whole Create StockSpringBootStarter class

|  |
| --- |
| @SpringBootApplication  public class StockSpringBootStarter {  public static void main(String[] args) {  SpringApplication.run(StockSpringBootStarter.class, args);  }  } |

1. Create StockResource class

|  |
| --- |
| @RestController  @RequestMapping(value = "/stock", produces = MediaType.APPLICATION\_JSON\_VALUE)  public class StockResource {  @RequestMapping(value = "/ping", method = RequestMethod.GET)  public String ping() {  return "Ping";  }  } |

1. Build the project

|  |
| --- |
| $ mvn clean install |

1. Launch spring boot application

|  |
| --- |
| $ java -jar target/12f\_app-1.0-SNAPSHOT.jar |

1. Check that app is working http://localhost:8080/stock/ping
2. Push app to cloudfoundry

|  |
| --- |
| $ cf push APP\_NAME -p target/12f\_app-1.0-SNAPSHOT.jar |

1. Check that it works http://APP\_NAME.cfapps.io/ping

### 

### 

### **III. Config**

#### **Store config in the environment**

An app’s config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

\*. Resource handles to the database, Memcached, and other backing services

\*. Credentials to external services such as Amazon S3 or Twitter

\*. Per-deploy values such as the canonical hostname for the deploy

\*. Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

The twelve-factor app stores config in environment variables (often shortened to env vars or env). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

1. Change StockResource class with the next code:

|  |
| --- |
| @Autowired  private Environment environment;  @RequestMapping(value = "/get-env", method = RequestMethod.GET)  public String getEnv(@RequestParam String env) {  return "Env" + " " + environment.getProperty(env);  } |

1. Execute (replace APP\_NAME with some unique name)

|  |
| --- |
| $ mvn clean install  $ cf push APP\_NAME -p target/12f\_app-1.0-SNAPSHOT.jar |

1. Check http://APP\_NAME.cfapps.io/stock/get-env?env=CF\_INSTANCE\_IP
2. Set some custom variable

|  |
| --- |
| $ cf set-env APP\_NAME my-env some\_value  $ cf restart APP\_NAME |

1. Check http://APP\_NAME.cfapps.io/get-env?env=my-env

### 

### 

### **IV. Backing Services**

#### **Treat backing services as attached resources**

A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app’s code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.

1. Add spring-boot-starter-data-jpa to pom.xml

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency> |

1. Add mysql connector

|  |
| --- |
| <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

1. Add flywaydb

|  |
| --- |
| <dependency>  <groupId>org.flywaydb</groupId>  <artifactId>flyway-core</artifactId>  </dependency> |

1. Create db migration resources/db/migration/V1.0.0\_\_create\_stock\_table.sql .

|  |
| --- |
| CREATE TABLE `stock\_item` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `type` varchar(45) NOT NULL,  `title` varchar(45) NOT NULL,  `description` varchar(1000) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8; |

1. In your main package create subpackage domain.model and put ItemType enum and StockItem entity there

|  |
| --- |
| public enum ItemType {  UNKNOWN(1L),  ELECTRONICS(2L),  CD(3L);  Long id;  ItemType(Long id) {  this.id = id;  }  public Long getId() {  return id;  }  } |

|  |
| --- |
| @Entity  @Table(name = "stock\_item")  public class StockItem {  @Id  @GeneratedValue  @Column(name = "id", nullable = false, unique = true)  private Long id;  @Enumerated(EnumType.STRING)  private ItemType type;  @Column(name = "title", nullable = false, length = 45)  private String title;  @Column(name = "description", length = 1000)  private String description;  public StockItem() {  }  public StockItem(Long id, ItemType type, String title, String description) {  this.id = id;  this.type = type;  this.title = title;  this.description = description;  }  public Long getId() {  return id;  }  public ItemType getType() {  return type;  }  public String getTitle() {  return title;  }  public String getDescription() {  return description;  }  } |

1. Create corresponding repository StockItemRepository.

|  |
| --- |
| public interface StockItemRepository extends CrudRepository<StockItem, Long> {  List<StockItem> findByType(ItemType type);  } |

1. Update your StockResource to provide standard CRUD operations

|  |
| --- |
| @Autowired  private Environment environment;  @Autowired  private StockItemRepository stockItemRepository;  @RequestMapping(method = RequestMethod.GET, value = "/{id}")  public StockItem stockItem(@PathVariable("id") Long id, HttpServletResponse response) {  StockItem stockItem = stockItemRepository.findOne(id);  if (stockItem == null) {  response.setStatus(HttpStatus.NOT\_FOUND.value());  return null;  }  return stockItem;  }  @RequestMapping(value = "", method = RequestMethod.POST)  public StockItem storeInStock(@RequestBody StockItem stockItem, HttpServletResponse response) {  StockItem storedItem = stockItemRepository.save(stockItem);  response.setStatus(HttpStatus.CREATED.value());  return storedItem;  }  @RequestMapping(value = "", method = RequestMethod.PUT)  public StockItem update(@RequestBody StockItem stockItem, HttpServletResponse response) {  if (!stockItemRepository.exists(stockItem.getId())) {  response.setStatus(HttpStatus.NOT\_FOUND.value());  return stockItem;  }  return stockItemRepository.save(stockItem);  }  @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)  public void removeFromStock(@PathVariable("id") Long id, HttpServletResponse response) {  StockItem stockItemToRemove = stockItemRepository.findOne(id);  if (stockItemToRemove == null) {  response.setStatus(HttpStatus.NOT\_FOUND.value());  return;  }  stockItemRepository.delete(stockItemToRemove);  }  @RequestMapping(value = "", method = RequestMethod.GET)  public Iterable<StockItem> items() {  return stockItemRepository.findAll();  } |

1. Create CF configuration file manifest.yml. Place it in the root of your project.

|  |
| --- |
| applications:  - name: APP\_NAME  path: target/12f\_app-1.0-SNAPSHOT.jar  memory: 1G  services:  - my-db |

1. Create MySQL service in our CF (If you are using not PWS, service name “cleardb” and service plan name “spark” might be different. Use ‘cf marketplace’ command to find out those values)

|  |
| --- |
| $ cf create-service cleardb spark my-db |

1. Rebuild and push application

|  |
| --- |
| $ mvn clean install  $ cf push |

1. Insert some resource (If you don’t have curl - use any other tool cappable of sending http requests)

|  |
| --- |
| $ curl -H "Content-Type: application/json" -d '{"type":"CD","title":"Some awesome movie","description":"Hello"}' http://APP\_NAME.cfapps.io/stock |

1. Check that resource is inserted correctly (open the url in your browser)

|  |
| --- |
| http://APP\_NAME.cfapps.io/stock |

### 

### 

### **V. Build, release, run**

#### **Strictly separate build and run stages**

A codebase is transformed into a (non-development) deploy through three stages:

The build stage is a transform which converts a code repo into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches and vendors dependencies and compiles binaries and assets.

The release stage takes the build produced by the build stage and combines it with the deploy’s current config. The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.

The run stage (also known as “runtime”) runs the app in the execution environment, by launching some set of the app’s processes against a selected release.

The twelve-factor app uses strict separation between the build, release, and run stages. For example, it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.

1. Register on travis-ci.org
2. Activate your repository
3. Install travis gem (optional)

|  |
| --- |
| $ gem install travis |

1. Add .travis.yml with the following content

|  |
| --- |
| language: java  before\_script: cd 12f\_app  script: mvn clean install  deploy:  edge: true  provider: cloudfoundry  username: YOUR\_USERNAME  api: https://api.run.pivotal.io  organization: YOUR\_ORG  space: YOUR\_SPACE |

1. Encrypt your password. (You can also specify password in clear text)

|  |
| --- |
| $ travis encrypt --add deploy.password |

1. Push the app to git and see how it is deployed

|  |
| --- |
| $ git add .  $ git commit -m "initial commit"  $ git push --set-upstream origin master |

### 

### 

### **VI. Processes**

#### **Execute the app as one or more stateless processes**

In the simplest case, the code is a stand-alone script, the execution environment is a developer’s local laptop with an installed language runtime, and the process is launched via the command line (for example, python my\_script.py). On the other end of the spectrum, a production deploy of a sophisticated app may use many process types, instantiated into zero or more running processes.

Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.

The memory space or filesystem of the process can be used as a brief, single-transaction cache. For example, downloading a large file, operating on it, and storing the results of the operation in the database. The twelve-factor app never assumes that anything cached in memory or on disk will be available on a future request or job – with many processes of each type running, chances are high that a future request will be served by a different process. Even when running only one process, a restart (triggered by code deploy, config change, or the execution environment relocating the process to a different physical location) will usually wipe out all local (e.g., memory and filesystem) state.

#### **Introduce a stateful process, that violates the rule**

1. Add security dependency

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency> |

1. Add SecurityConfig class

|  |
| --- |
| @Configuration  @EnableWebSecurity  public class SecurityConfig extends WebSecurityConfigurerAdapter {  @Override  protected void configure(AuthenticationManagerBuilder auth) throws Exception {  auth.inMemoryAuthentication()  .withUser("admin").password("admin\_pass").roles("ADMIN")  .and()  .withUser("user").password("user\_pass").roles("USER");  }  @Override  protected void configure(HttpSecurity http) throws Exception {  http  .authorizeRequests()  .antMatchers("/secured/admin").hasRole("ADMIN")  .antMatchers("/secured/user").authenticated()  .and()  .httpBasic()  .and()  .requestCache()  .requestCache(new NullRequestCache());  }  } |

1. Create two resources to test Spring Session: SecuredResource and LogoutResource

|  |
| --- |
| @RestController  @RequestMapping(value = "/secured")  public class SecuredResource {  @RequestMapping(value = "/user", method = RequestMethod.GET)  public String userHello(Principal principal) {  return "Hello " + principal.getName() + "!";  }  @RequestMapping(value = "/admin", method = RequestMethod.GET)  public String adminHello(Principal principal) {  return "Hello " + principal.getName() + "!";  }  } |

|  |
| --- |
| @RestController  public class LogoutResource {  @RequestMapping(value = "/logout", method = RequestMethod.GET)  public String logout(HttpSession session) {  session.invalidate();  return "Logout done!";  }  } |

1. Build and push the application to the CF
2. Login to the application

|  |
| --- |
| https://user@APP\_NAME.cfapps.io/secured/user |

(@user) in the beginning of the url prevents browser from caching basic authentication credentials.

1. Trigger the same request a few times and ensure that your credentials are still active
2. Restart application and see that you need to login one more time

#### **Fix stateful process**

1. Add redis dependency

|  |
| --- |
| <dependency>  <groupId>org.springframework.session</groupId>  <artifactId>spring-session-data-redis</artifactId>  </dependency> |

1. Mark StockSpringBootStarter with annotation @EnableRedisHttpSession to let Spring Boot know that we use Redis to keep sessions
2. Create Redis service in CF (Sample below applies to PWS)

|  |
| --- |
| $ cf create-service rediscloud 30mb my-redis |

1. Include my-redis service instance in manifest.yml file

|  |
| --- |
| applications:  - name: APP\_NAME  path: target/12f\_app-1.0-SNAPSHOT.jar  memory: 1G  services:  - my-db  - my-redis |

1. Build and push the app and check that authentication stays between restart

### 

### 

### **VIII. Concurrency**

#### **Scale out via the process model**

Any computer program, once run, is represented by one or more processes. Web apps have taken a variety of process-execution forms. For example, PHP processes run as child processes of Apache, started on demand as needed by request volume. Java processes take the opposite approach, with the JVM providing one massive uberprocess that reserves a large block of system resources (CPU and memory) on startup, with concurrency managed internally via threads. In both cases, the running process(es) are only minimally visible to the developers of the app.

In the twelve-factor app, processes are a first class citizen. Processes in the twelve-factor app take strong cues from the *unix process model for running service daemons*. Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.

1. Scale your application

|  |
| --- |
| $ cf scale APP\_NAME -i 2 |

1. Execute the following request several times, and observe that that it is handled by different application instances (if you see response from the same instance all the time)

|  |
| --- |
| http://APP\_NAME.cfapps.io/stock/get-env?env=CF\_INSTANCE\_PORT |

### 

### 

### **IX. Disposability**

#### **Maximize robustness with fast startup and graceful shutdown**

The twelve-factor app’s processes are disposable, meaning they can be started or stopped at a moment’s notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

Processes should strive to minimize startup time. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs. Short startup time provides more agility for the release process and scaling up; and it aids robustness, because the process manager can more easily move processes to new physical machines when warranted.

Processes shut down gracefully when they receive a SIGTERM signal from the process manager. For a web process, graceful shutdown is achieved by ceasing to listen on the service port (thereby refusing any new requests), allowing any current requests to finish, and then exiting. Implicit in this model is that HTTP requests are short (no more than a few seconds), or in the case of long polling, the client should seamlessly attempt to reconnect when the connection is lost.

1. Delete one application instance by scaling down the application

|  |
| --- |
| $ cf scale APP\_NAME -i 1 |

1. Ensure that app still works fine

### 

### 

### **X. Dev/prod parity**

#### **Keep development, staging, and production as similar as possible**

Historically, there have been substantial gaps between development (a developer making live edits to a local deploy of the app) and production (a running deploy of the app accessed by end users). These gaps manifest in three areas:

* The time gap: A developer may work on code that takes days, weeks, or even months to go into production.
* The personnel gap: Developers write code, ops engineers deploy it.
* The tools gap: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux.

The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small. Looking at the three gaps described above:

* Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.
* Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.
* Make the tools gap small: keep development and production as similar as possible.

1. Create new space

|  |
| --- |
| $ cf create-space staging |

1. Create new git branch

|  |
| --- |
| $ git checkout -b staging |

1. Modify .travis.yml and update the space here
2. Create necessary services in the new space
3. Push the application to git

### 

### 

### **XI. Logs**

#### **Treat logs as event streams**

*Logs* provide visibility into the behavior of a running app. In server-based environments they are commonly written to a file on disk (a “logfile”); but this is only an output format.

Logs are the stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services. Logs in their raw form are typically a text format with one event per line (though backtraces from exceptions may span multiple lines). Logs have no fixed beginning or end, but flow continuously as long as the app is operating.

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior.

1. Add logging dependencies

|  |
| --- |
| <dependency>  <groupId>org.slf4j</groupId>  <artifactId>jcl-over-slf4j</artifactId>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  </dependency> |

1. In the resources folder create Logback configuration file logback.xml file with the next content. Dont forget to replace YOUR.PACKAGE with the package name.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  <layout class="ch.qos.logback.classic.PatternLayout">  <Pattern>  %d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n  </Pattern>  </layout>  </appender>  <logger name="YOUR.PACKAGE" level="info" additivity="false">  <appender-ref ref="STDOUT"/>  </logger>  <root level="error">  <appender-ref ref="STDOUT"/>  </root>  </configuration> |

1. Add logging to the get by id endpoint

|  |
| --- |
| private static final Logger logger = LoggerFactory.getLogger(StockResource.class);  @RequestMapping(method = RequestMethod.GET, value = "/{id}")  public StockItem stockItem(@PathVariable("id") Long id, HttpServletResponse response) {  logger.info("Starting search of stock item(id={}) search", id);  StockItem stockItem = stockItemRepository.findOne(id);  if (stockItem == null) {  logger.info("Stock item(id={}) has not been found", id);  response.setStatus(HttpStatus.NOT\_FOUND.value());  return null;  }  logger.info("Finishing search of stock item(id={}) search", id);  return stockItem;  } |

1. Redeploy and execute request to view any stock item info.
2. Check the logs, ensure that the lines that we have added are in the log.

|  |
| --- |
| $ cf logs APP\_NAME |

### 

### 

### **XII. Admin processes**

#### **Run admin/management tasks as one-off processes**

The *process formation* is the array of processes that are used to do the app’s regular business (such as handling web requests) as it runs. Separately, developers will often wish to do one-off administrative or maintenance tasks for the app, such as:

* Running database migrations (e.g. manage.py migrate in Django, rake db:migrate in Rails).
* Running a console (also known as a *REPL* shell) to run arbitrary code or inspect the app’s models against the live database. Most languages provide a REPL by running the interpreter without any arguments (e.g. python or perl) or in some cases have a separate command (e.g. irb for Ruby, rails console for Rails).
* Running one-time scripts committed into the app’s repo (e.g. php scripts/fix\_bad\_records.php).
* One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.

The same dependency isolation techniques should be used on all process types. For example, if the Ruby web process uses the command bundle exec thin start, then a database migration should use bundle exec rake db:migrate. Likewise, a Python program using Virtualenv should use the vendored bin/python for running both the Tornado web server and any manage.py admin processes.