AgileDev

v. 0.1.9-SNAPSHOT

Project Documentation

Table Of Content

1 Developers Setup

Developer setup guide

These instructions are intended to assist the would-be developer with setting up an environment in which to successfully build the source code.

1.1 Required tools

- Maven
- GPG

1.2 Details

1.2.1 Installation

1. Download and Install Maven - Download the latest version of Maven and install it
2. Add username/password github account to `settings.xml` maven. `<servers> <server> <id>github</id> <username>GitHubLogin</username> <password>GitHubPassw0rd</password> </server> </servers>`
3. Add username/password `ossrh` account to `settings.xml` maven. `<settings> <servers> <server> <id>ossrh</id> <username>your-jira-id</username> <password>your-jira-pwd</password> </server> </servers> </settings>`
4. Download GPG - Download GPG and follow the instructions and install it to your system. Verify your gpg installation by running gpg with the version flag `gpg --version`
5. Generate key pair - Before you do anything with GPG, you will need to generate a key pair for yourself. Once you have you own key pair, you can use your private key to sign artifacts, and distribute your public key to public key servers and end-users so that they can validate artifacts signed with your private key. Generate a key pair like this `gpg --gen-key` You'll be asked for the type, the size, and the time of validity for the key, just use the default value if you don't have any special requirements. You will be asked to input your name, email, and comment for the key. These identifiers are essential as they will be seen by anyone downloading a software artifact and validating a signatute. Finally, you can provide a passphase to protect your secret key, this is not mandatory, but I highly recommend you to do this. It is essential that you choose a secure passphrase and that you do not divulge it to any one. This passphrase and your private key are all that is needed to sign artifacts with your signature.
6. Add GPG Passphrase to `settings.xml`: `<settings> <profiles> <profile> <id>release</id> <activation> <property> <name>performRelease</name> <value>true</value> </property> </activation> <properties> <gpg.passphrase>the_pass_phrase</gpg.passphrase> </properties> </profile> </profiles> </settings>`
7. Distribute Your Public Key - Since other people need your public key to verify your files, you have to distribute your public key to a key server: `gpg --keyserver <key server url> --send-keys <key id>` You can get your `keyid` by listing the public keys (`gpg --list-keys`). The line starting with pub shows the length, the keyid, and the creation date of the public key.

1.2.2 Build

Once the above setup is complete:

1. Open a fresh terminal

2. Navigate to the `${project.name} directory, or whatever you called it.
3. Type the following on the command line: `mvn clean install`

1.2.3 Snapshot deployment

Snapshot deployment are performed when your version ends in -SNAPSHOT . When performing snapshot deployments simply run `mvn clean deploy` on your project.

SNAPSHOT versions are not synchronized to the Central Repository. If you wish your users to consume your SNAPSHOT versions, they would need to add the snapshot repository to their Nexus, `settings.xml`, or `pom.xml`. Successfully deployed SNAPSHOT versions will be found in `https://oss.sonatype.org/content/repositories/snapshots/`.

1.2.4 Release deployment

You can perform a release deployment to OSSRH with `mvn release:clean release:prepare` by answering the prompts for versions and tags, followed by `mvn release:perform`.

This execution will deploy to OSSRH and release to the Central Repository in one go, thanks to the usage of the Nexus Staging Maven Plugin with `autoReleaseAfterClose` set to `true`.

2 Site Documentation

Site Documentation

All documentation is within the code base and is written in MarkDown.

Markdown is a plain text formatting syntax designed so that it can be converted to HTML. There is no clearly defined Markdown standard but if you comply to the following syntax the generated pages should be formatted correctly.

Paragraphs and Breaks

```
Markdown accepts text on consecutive
as a hard-wrapped paragraph.

Put a blank Line in between to start a
```

Markdown accepts text on consecutive lines as a hard-wrapped paragraph.

Put a blank Line in between to start a new graph.

Headings

```
# Header 1
## Header 2
### Header 3
```

Header 1 Header 2 Header 3

Blockquote

```
> Blockquote.
```

Blockquote.

Lists

```
Start ordered list with number.

6. Ordered
7. List
    8. Items

Unordered list

* Unordered
* List
    * Item
```

1. Ordered
2. List
    1. Items

Unordered list

- Unordered
- List
    - Item

Links

```
An [inline link](http://www.google.co
An [inline link to header page](#Site
A [reference link][id].
[id] defined elsewhere
[id]: <http://www.google.com/> (optio
Linked URL: <http://www.google.com/>
Mail to: <address@example.com>
```

An inline link.

An inline link to header page references header on current page.

urrent page.

A reference link.

id defined elsewhere

Linked URL: http://www.google.com/

Mail to: address@example.com

Images

```
![Alternative title image](images/Uni
```

Code

```
Inline `code` example.
Preformatted code block:
    <strong>Just indent 4 spaces.</st
Double-backticks (`` ` ``) delimit li
```

Inline `code` example.

Preformatted code block:

```
<strong>Just indent 4 spaces.</strong>
```

Double-backticks (`) delimit literal backticks

3 Branching Models

Branching Models

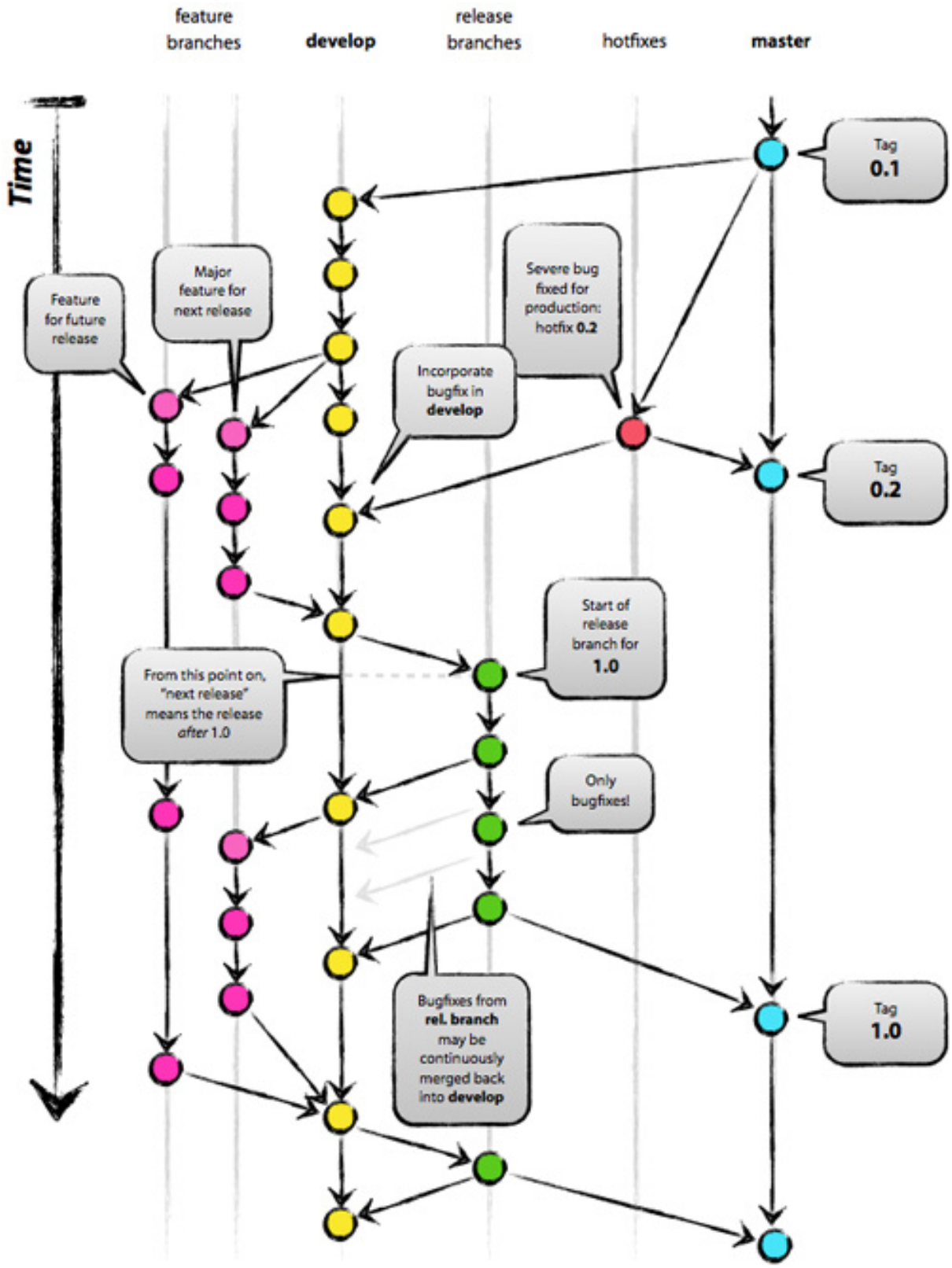
Within the following paragraphs will discuss some of the popular branching models which are around these days. After reading this you should be able to decide which one fits your project the most.

3.1 Git-Flow

This workflow has been published by Vincent Driesen as a successful branching model for git and covers most of the standard needs for a ‘classical’ development project.

3.1.1 How it works

Git-Flow has the following branching model:



There are 2 primary branches:

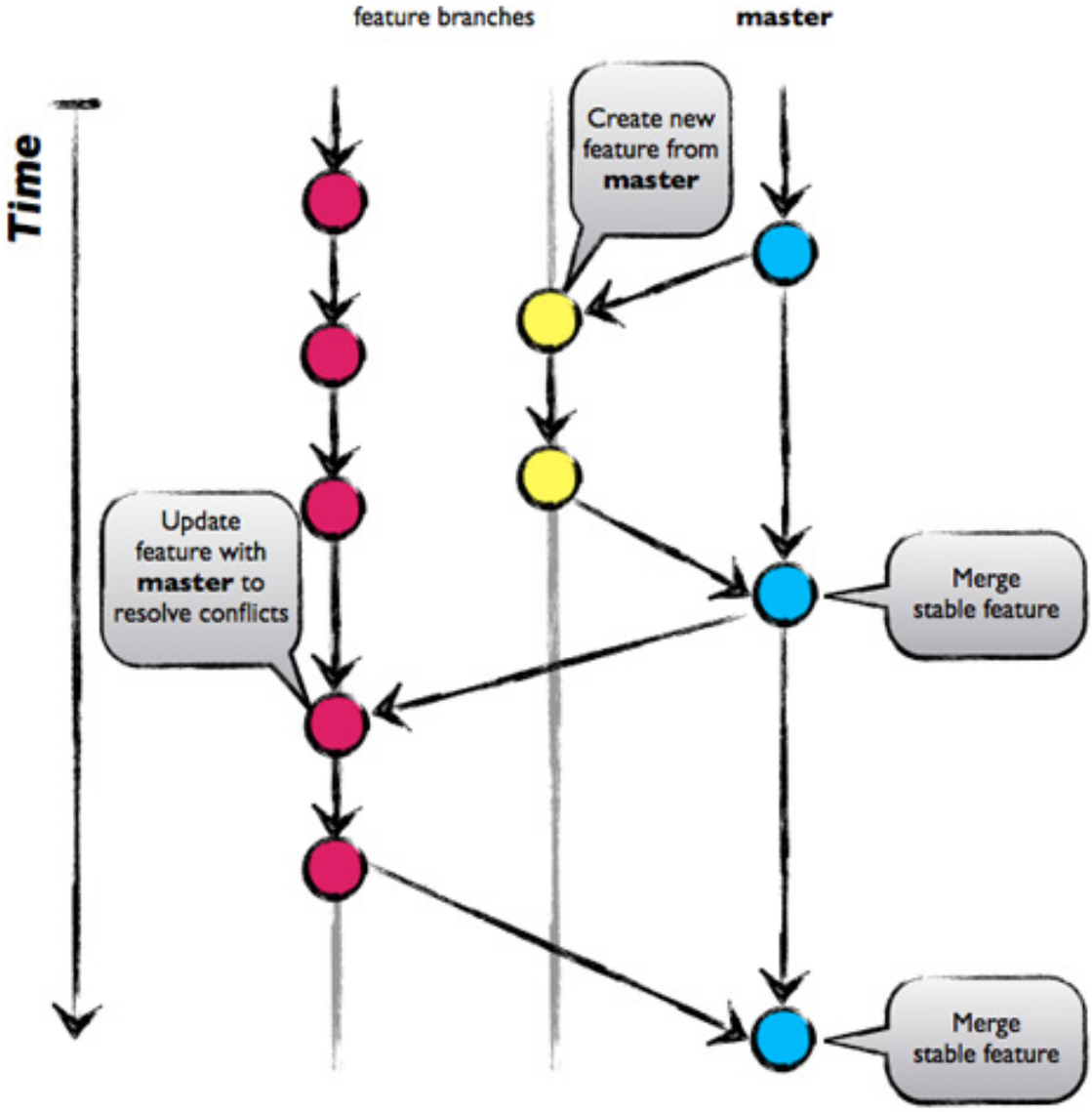
- `master` is the main branch where everything is stable. Each commit is a stable (fully tested) version of the project, (i.e. a release) which could be deployed to production and tagged accordingly.
- `develop` is the main branch where development is done. It will contain prepared changes for the next release in `master`.

And secondary branches which are flexible over time:

- `feature` starts from `develop` and merge into `develop`. When you are working on a specific feature, you create a `feature\xxx` branch and once it's done, you merge it back into `develop` to add the stable feature to the scope for the next release. The essence of a `feature` branch is that it exists as long as the feature is in development, but will eventually be merged back into `develop` or discarded.
- `release` starts from `develop` and merge into `master` and/or `develop`. When `develop` is reflecting the desired state of the feature release, (all features for the release haven been merged), you create a 'release/xxx' branch. By doing this, you can prepare the next release, correct eventual bugs and continue development in parallel. All features targeted at future releases may not be merged into the `develop` branch until the `release` branch is branched off. It is exactly at the start of a release branch that the upcoming release gets a version number assigned.
- `hotfix` starts from `master` and merge into `master` and/or `develop`/ `release`. When you want quickly to resolve critical bugs in production you create a `hotfix/xxx` branch. When the hotfix is developed, you merge it back into `master` with the appropriate version number, and into `develop` and/or `release` branch to update it with the modifications you made.

3.2 GitHub

GitHub has the following branching model:



As you can see in the picture above, there is only a single `master` branch and the following 6 basic rules should be followed:

1. **Everything that is in `master` could be deployed in production** - The `master` branch is the only meaningful branch of the project and it should stay stable in any circumstance so you can base your work upon and deploy it to production at anytime.
2. **Create descriptive feature branches from `master`** - When you want to develop a feature or a hotfix, you just create your branch from `master` with an explicit name that describe your work.
3. **Regularly push to `origin`** - In contrary of the Git-Flow where developers doesn't have to push its local `feature` branch to the main repo, you have to do that regularly within the GitHub model.
4. **Open a pull-request at anytime**
5. **Only merge after a pull request review** - This is more an advice than an absolute rule. It's a best practice that another developer should review the pull request and confirm that the branch is stable. From here you can, merge the branch back into `master` and delete the `merge` branch.

6. **Immediately deploy after you merge into `master`** - Once your branch is merged into `master` the whole thing is deployed to production. Doing so, you will stress on the necessite to keep `master` stable. Developers don't want to break everything because of its modifications were deployed, so they are more likely to pay attention to code stability before merge.

The GitHub model perfectly fits projects that don't have releases nor versions.

You do continuously integrate into `master` and you deploy the stable project to production regularly; sometimes several time a day.

Due to this it's very unlikely to add serries of big bugs. If problems appear, they are quickly fixed on the go. There is no difference between a big feature and a small hotfix in terms of process.

3.3 Final words

To choose the best model depends on the nature and needs of your project. It's up to you to consider which of these models fit more or less to your environment. If necessary adapt the framework to your needs.

4 CQRS clarified

Command Query Responsibility Segregation

CQRS stands for Command Query Responsibility Segregation and helps developers to develop scalable, extensible and maintainable applications. At its hearth the pattern makes a distinction between the model in which information is updated and the model from which information is read. Following the vocabulary of *Command Query Separation* the models are called Command and Query. The rationale behind this separation is that for many problems, particularly in complicated domains, having the same conceptual model for updating information and reading information leads to a complex model that does neither well.

The models may share the same database, in which case the database acts as the communication between the two models, but they may also use separate databases, making the query side database a real-time reporting database. In later case there needs to be some communication mechanism between the two models, or their databases. Commonly this communication is realized by events.

4.1 Querying

A query returns data and does not alter the state of the model. Once the data has been been retrieved by an actor, that same data may have been changed by another actor, in other words it is stale. If the data we are going to return to actors is stale anyway, is it really necessary to go to the master database and get it from there? Why transform the persisted data into entities if we just want data, not any rule preserving behavior? Why transform those entities to DTOs to transfer them across.

In short, it looks like we are doing a lot of unnecessary work. So why not creating a additional model, (whose data can be a bit out of sync), that matches the DTOs as they are expected by the requester. As data store you can use a regular database but that is not mandatory.

4.2 Commands

A command changes the state of an entity within the command model. A command may be accepted or rejected. An accepted command leads to zero or more events being emitted to incorporate new fact into the system. A rejected command leads to some kind of exception.

One should regard a command as a request to perform a unit of work which is not depending on anything else.

4.3 Domain event

In CQRS, domain events are the source all changes in the query model. When a command is executed, it will change state of one or more entities within the command model. As a result of these changes, one or more events are dispatched. The events are picked up by the event handlers of the query model and those update the query model.

4.4 When to use CQRS.

Like any pattern, CQRS is useful in some places and in others not. Many systems do fit a CRUD model, and should be done in that style. In particular CQRS should only be used on specific portions of a system (Bounded context in DDD lingo) and not to the system as whole; e.g. each bounded context needs its own decision on how it should be modeled.

So far there are the following benefits:

- Handling complexity - a complex domain may be easier to tackle by using CQRS.
- Handling high performance applications - CQRS allows you to separate the load from reads and writes allowing you to scale each independently.

5 Glossary

Glossary

- **Architectural Epic** - Architectural epics are large, typically cross cutting technology initiatives that are necessary to evolve solutions to support current and future business needs.
- **Business Epic** - Business epics are large, typically cross cutting, customer facing initiatives that encapsulate the new development to realize certain business benefits.
- **Epic** - There are 2 kind of epics, business epics and architectural epics.
- **Feature** - Features are services provided by the system that fulfill one or more stakeholder needs. Features are a central part of Agile and often come from the breakdown of Epics. Features live at a level above specific requirements and bridge the gap from the problem domain (understanding the needs of the user and stakeholders) to the solution domain (specific requirements and acceptance criteria). The attributes of a feature may include: *The feature* - A short 1-3 word title summarizing the feature *The benefit* - A short description which elaborates the feature and describes the value to the user. There may be multiple benefits per feature.
- **User story** - Traditionally expressing intended system behavior has been captured in software requirements specifications. In agile this traditional document is replaced by a series of stories which express the intent of the system. Stories are small, independent behaviors that can be implemented incrementally, each of which provides some value to the user. The recommended form of expressing a user story is the user voice form, and it appears as follows: As a *user role* I can *activity* so that *business value*

6 Abbreviations

Abbreviations

- CQRS - Command-Query Responsibility Segregation
- DTO - Data Transfer Object
- REST - REpresentational State Transfer