Test Driven Infrastructure

Schlomo Schapiro
Systems Architect & Open Source Evangelist

Immobilien Scout 24
Andreasstr. 10
10243 Berlin
schlomo.schapiro@immobilienscout24.de

Abstract: DevOps is one of the biggest trends in modern IT environments. Interdepartmental walls are coming down, developers gain production access and learn to be responsible for their own code in production. Admins learn to code their infrastructure and to build delivery chains. Test Driven Development (TDD) is a common topic for both developers and admins. This paper explores the challenges that infrastructure development poses for TDD and how they were overcome at ImmobilienScout24.

1 DevOps

IT organizations "traditionally" have different departments for software development and operations. Opposing goals, different backgrounds, skill levels and ingrained prejudice have lead in some organizations to deep rifts between the departments. As a result, delivering changes from development into production is a slow, cumbersome and even unpleasant process.

In recent years, software development was reformed by Agile[1] methods like Scrum. Many agile methods have in common that they structure the communication between client and developer through backlogs and help to organize the development work itself. Agile teams are staffed cross-functional to include all required skills in the team.

In many organizations Agile was introduced to improve the software development process without changing the way how software is run in production or how the developers interact with the operations department. As a result admins suddenly faced a significant increase in developer output and developers felt more than ever before that operations is slowing them down.

[1]The Agile Manifesto for Software Development (http://agilemanifesto.org/)

DevOps is the extension of Agile methodology to cover all IT departments and to find a new base for cooperation between development and operations. It is based on four guiding principles: Culture, Automation, Measurement and Sharing.

At the core of DevOps culture are common human guiding principles: Respect for each other, understanding for the challenges and worries of others and the motivation to find a common solution by working together. Working together is also the base for learning important skills like operational awareness or development best practices from each other.

To reach understanding and respect, one must make his own work transparent and allow others to experience it themselves. The Agile way of creating transparency through backlogs, daily status meetings and visible task tracking works equally well for operations. Staff exchanges between development and operations are therefore an important tool to achieve the cultural change. Setting common goals and sharing responsibilities are the other important steps. Shared responsibility also entails granting all involved persons the same power to enact changes in the system they are responsible for. In practice this means giving developers production access and the same power as admins.

Sharing administrative privileges with a much larger group of persons brings its own challenge. Many organizations implemented “tight security” by allowing only a small number of admins to work on the production servers. A low degree of automation added another layer of perceived security since all important decisions and actions had to be done by humans. Human error was viewed as inevitable, checklists and four-eyes principle where common methods to reduce the risk of change.

All of this does not scale to the amount of people who eventually get administrative access with the shared responsibility model common with DevOps. On the other hand, not granting developers production rights usually leads to developers not caring about the troubles in production thereby nullifying the goal of shared responsibility.

2 Trust the Code

A solution to this dilemma can be changing the way how trust to do the right thing is handled within the organization (trust not to do evil is implied here). Instead of building up security by trusting only a few people it is also possible to build up security by trusting code and automation. This is very similar to how Open Source projects guarantee security: Everybody can look into the source code and development process. This openness makes it very hard to inject bad code (or at least to keep it there for a long time).

Applied to an IT organization that means to automate the tasks previously only admins could do. Then everybody can perform them – through trusted code. Typically the first

candidates for automation are systems provisioning, code deployment, configuration management and database management. Good automation that helps in most or all situations reduces the actual need for administrative privileges. Eventually developers and admins alike feel less need to actually login onto servers.

If the code for automation should serve as a trust base then it needs to be especially trustworthy. Test Driven Development (TDD) is a very common and well proven practice to ensure the long-term quality of software projects. The core of TDD consists of automating the testing in such a way that the automated testing can become part of the regular development process.

3 Test Driven Infrastructure Development

Infrastructure development is somewhat different from typical software development. The main difference lies in the fact that infrastructure development also covers the lower levels of systems administration like OS provisioning and systems setup, patch management, configuration management, email configuration, database migration, router and switch configuration and others. Some of these areas require elaborate setups to be able to run a single test: For example to test the installation procedure that installs an operating system into an empty server one must automate a system that does not yet run an operating system.

Consequently the two test types most useful in infrastructure development are the unit test and the system test. Our use of these terms is somewhat different from software development. The unit test will test the smallest possible component in an artificial environment. The system test will test the entire application in a real(istic) environment together with other applications.

An important aspect of any kind of tests is the right level of mocking. Mocking replaces external dependencies with fakes that provide the same result. Mocking is very important to focus the test on the actual code under scrutiny and not to test the new code together with other components or external systems. Those could fail on their own accord and falsify the test, misleading the developer to look for errors in his code that are not there.

Unit tests are typically run as part of the build process and can be run on the developer workstation or on a build server. Unit tests should not have any external dependencies and run shortly so that they can be run very often or even after each small code change. A maximum of mocking is common with unit test. Typical examples for infrastructure development are syntax checks and running Shell scripts with mock input.

System test are the opposite. They test a complete build of the software that could be also used in production. System tests require therefore a production-like setup that should be as close to production as possible. External dependencies are either

implemented in suitable test configurations or mocked at a systems-level. For example, to test the email server configuration one must setup a system that will not send any emails to the outside so that the test emails will not reach customers. To test OS provisioning a new VM is started, installed, checked and thrown away all in one go.

System tests consist of many steps: Build the software from source, install it on the test systems, setup the test environment, run the tests and propagate the software artifact into production or at least to the next stage. Automating this process creates a delivery chain.

4 Delivery Chains

Ideally people trust the test more than manual procedures, so that code changes will be deployed to production after they pass all tests. If the developers and admins (and product managers) do not trust the tests and require a manual release process then they should improve the tests instead of babysitting the test automation.

If each application – no matter if it is application software, an infrastructure service or data management, is covered by automated tests which are organized in automated delivery chains then the whole IT organization starts to profit. Everybody can focus on creating new values instead of dealing with repetitive manual steps (which are error prone) and systems administration.

5 Test Driven Infrastructure at ImmobilienScout24

The biggest learnings from development in operations was in the area of TDD. Over the last years all important infrastructure services where covered with extensive automated tests: ESX server and VM provisioning, Subversion repository hosting, database setup and migration, DKIM key rotation[2] and many more.

The operations teams also adopted a “test first” philosophy and create unit and system tests wherever possible. With new projects the test coverage is systematically built up during development, work on existing projects often starts from creating a test environment.

Trust in tests is well established and widespread and code changes are deployed automatically to production. The teams now focus much more on developing new features and less on system administration.

Further practical examples can be found in Linux Magazin 09/2014 “Testgetrieben”[3] and a EuroPython 2014 talk “DevOps Risk Mitigation – Test Driven Infrastructure”[4].

[2] https://www.heinlein-support.de/mk/2014/vortrag/automatische-rotation-von-dkim-schluesseln, Stefan Neben
[3] http://www.linux-magazin.de/Ausgaben/2014/09/Testgetrieben, Schlomo Schapiro
[4] http://www.slideshare.net/schlomo/europython-2014-devops-risk-mitigation, Schlomo Schapiro