Cloudfier: Automatic Architecture for Information Management Applications

Rafael Chaves

Abstratt Technologies
Florianópolis – SC – Brazil

rafael@abstratt.com

http://youtu.be/Ntya06InAbU

Abstract. *Information management applications have always taken up a significant portion of the software market. The primary value in this kind of software is in how well it encodes the knowledge of the business domain and helps streamline and automate business activities. However, much of the effort in developing such applications is wasted dealing with technological aspects, which in the grand scheme of things, are of little relevance.*

Cloudfier is a model-driven platform for development and deployment of information management applications that allows developers to focus on understanding the business domain and building a conceptual solution, and later apply an architecture automatically to produce a running application. The benefit is that applications become easier to develop and maintain, and inherently future-proof.

1. Motivation

Information management systems such as line-of-business and departmental applications make up a significant portion of what is produced by the software industry. The primary value in those kinds of applications is in how well they encode the knowledge of the business domain they serve and the business-centric features they provide that streamline, automate and control key business activities and processes. They tend to have fairly standard requirements on deployment technology stack though.

Yet, much of the effort in building and maintaining such applications is wasted on mapping a conceptual solution (that usually only exists in the developers' minds) to a concrete solution using some general purpose implementation language – which, by the way, often provide poor capabilities for encoding domain knowledge.

A technology stack, while an enabler, is also a major liability – an application needs to be regularly updated to keep up with evolution of the components of the chosen stack, even if there are no changes in the business requirements; and if the platform is eventually discontinued, businesses are faced with the hard reality of keeping using an unsupported platform, or a costly rewrite of the application from scratch on some alternative platform.

Cloudfier is intended as a solution for this quagmire: applications are domain-centric, described using a high-level language, and free from technological concerns.

The architecture is defined separately from the application, and applied automatically, at deploy time, and evolves independently. Since the code is much simpler (as it ignores technological concerns) and the language better suitable for describing business domains, it is much easier to learn the domain requirements and rules from the application code.

2. Cloudfier features

Cloudfier is a platform for development and deployment of line of business applications based on model-driven development with executable models [Mellor, 2002]. In Cloudfier, applications only address the business requirements explicitly – the technical requirements are fulfilled automatically by the platform.

2.1. Modeling Language

Cloudfier applications are built using UML [Object Management Group, 2011], but that may not be obvious at first glance. UML, as a high-level language, provides a set of features for conceptual modeling of solutions that is unparalleled in breadth by any general purpose programming language currently available. However, its features for executable modeling (informally known as *action semantics*) are not exposed at all by the graphical notation. Hence, Cloudfier adopts TextUML [TextUML, 2014], a textual notation for UML that exposes the structural and behavioral language elements required for building UML models that are precise and complete enough to be executable.

2.2. Development Environment

Development in Cloudfier happens all within the web browser. In order to accomplish that, Cloudfier builds on Orion, an open-source platform for web-based development. Orion is a generic foundation for building cloud-based development environments, just as Eclipse was for desktop-based IDEs[1]. Cloudfier builds on three main areas of Orion:

Editor-based features. Orion provides a highly functional text editor. Cloudfier extends the Orion text editor with a TextUML specific syntax highlighter, a validator, an automatic code formatter, and an outline analyzer.

Shell-based features. Orion sports a *Shell page*, a page where users can issue commands as if they were working at a (very simple) character terminal. Many of Cloudfier features are triggered using shell commands.

Collaboration and Versioning. The textual nature of Cloudfier applications makes for a great fit for source control. Orion includes a rich support for collaboration via Git repositories, which can be used for developing Cloudfier applications.

2.3. Deployment

Cloudfier contributes a shell command that performs a full deploy of the application. This command recompiles the application, and after ensuring it is valid, publishes the model into the runtime and recreates the database to back it. After a successful deploy (using the *full-deploy* command), the user is presented with links for exploring the application using automatically generated user interfaces and a REST API.

---

1 The similarities go further: Orion is also an Eclipse.org project, and was also started by IBM

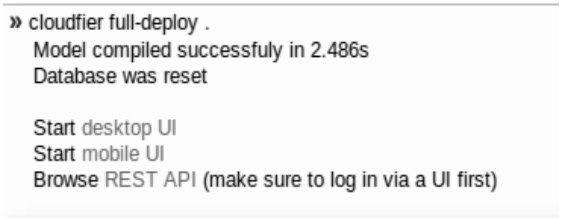

Figure 1. Deploying the application

2.4. Automated testing

Executable models are programs, and as such they can have bugs, so Cloudfier supports automated testing.

Tests are defined in Cloudfier at the model level, following the xUnit style. A stereotype allows developers to mark a class as a test class. Test cases are any public parameter-less operations found in a test class. Test cases are expected to complete successfully, but it is also possible to mark a test case as expecting a failure, which can include the name of the constraint expected to fail.

Once the application is recompiled, the user can run all tests in the project using the *run-tests* command.

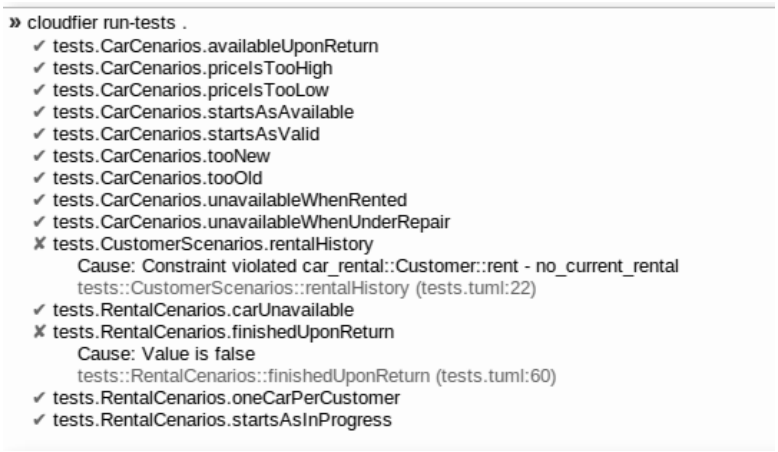

Figure 2. Running the application automated tests

2.5. Automatic REST API Generation

No application is an island. Cloudfier applications integrate with external systems via a (potentially bidirectional) REST API.

By default, every application has a dynamically generated inbound REST API that exposes all features of the application in a RESTful way, including both (structural) metadata and data. In fact, the REST API is the only way data can get out or into a Cloudfier application, user interfaces (generated or hand-crafted) have to go through the same REST API.

For the cases when the application should reach out to an external system, Cloudfier can provide an outbound API as well. Basically, whenever the application includes an *external service* class (where service and external are UML stereotypes), that means such service will not be part of the application itself, and any operation calls or signals sent to it will be delegated to its corresponding external REST endpoint.

2.6. Automatic User Interface Generation

Cloudfier generates a fully functional web-based user-interface for accessing the application (a variant for mobile devices is under development).

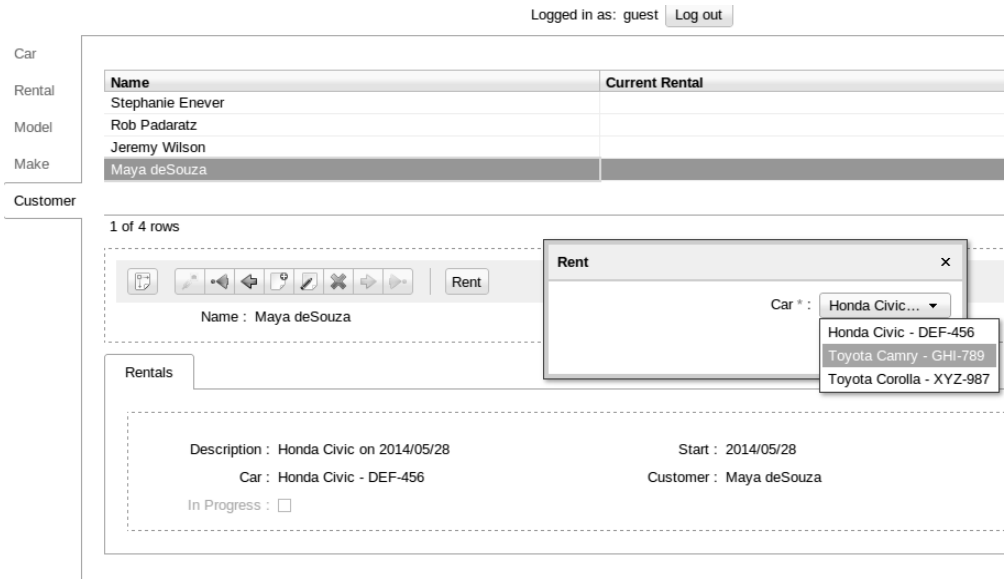

Figure 3. Cloudfier can generate a fully functional user interface automatically

The generated user interface allows users to perform basic CRUD manipulation of instances of the entities in the system, ensuring constraints defined by the model are honored (lower/upper boundaries, data type conformance, invariants). The UI allows the user to perform business-specific actions on the instances (based on operations defined in the model). It can also enable actions conditionally, based on their preconditions being satisfied (e.g., a customer can only rent a car if it has no pending payments). And if an operation receives a reference to some entity instance as a parameter (e.g., the car to rent), it will only offer those instances that would satisfy the corresponding operation's preconditions (e.g., only cars that are available). That is possible due to the application being a precise and detailed model, and the platform is aware of all that meta-information.

3. Developing an application

Consider you were building a simple expense reporting application. The requirements are quite trivial:

REQ1 Employees report expenses, including description, amount, and date.
REQ2 An expense starts as a draft, which still needs to be explicitly submitted.
REQ3 Once submitted, an expense can be approved, rejected, which are final.
REQ4 Rejection must include a rejection rationale to the submitter.
REQ5 Expenses below a threshold of $50.00 can be automatically approved.
REQ6 A submitter should be able to see all of her expenses classified by status.
REQ7 A submitter cannot approve her own expenses.

From REQ1, two entities jump to one's eyes: Employee and Expense. At a minimum, we would need a name to identify the employee:

```
class Employee
    attribute name : String;
end;
```

Expense is going to be a bit more elaborated:

```
class Expense
    attribute description : String;
    attribute amount : Double;
    attribute dateReported : Date := { Date#today() };
end;
```

There is also a relationship – an employee can have multiple expenses:

```
association EmployeeExpenses
    navigable role submitter : Employee;
    navigable role expenses : Expense[*];
end;
```

Based on REQ2 and REQ3, an expense starts in *Draft* status, and can then be submitted. Once submitted, it can be approved or rejected. That calls for a state machine:

```
class Expense …
    attribute status : Status;
    statemachine Status
        initial state Draft end;
        state Submitted end;
        state Approved end;
        state Rejected end;
    end;
end;
```

This is a good for a start but, as is, expenses will be stuck forever in the *Draft* status – the state machine is lacking transitions between states (and what triggers them):

```
statemachine Status
    initial state Draft
        transition on call(submit) to Submitted;
    end;
    state Submitted
        transition on call(approve) to Approved;
        transition on call(reject) to Rejected;
    end; …
end;
```

Which in other words, means: an expense starts as *Draft*; once the operation *submit* is invoked (generating an operation *call* event), it becomes *Submitted*; from there, it can be approved or rejected. But we have yet to define those operations:

```
class Expense …
    attribute comment : Memo[0,1];
    operation submit();
    operation approve();
    operation reject(reasonForRejection : Memo);
    begin
        self.comment := reasonForRejection;
    end;
end;
```

Now not only we have operations to trigger the transitions, but we also defined behavior for the only one that needed any: we require and record a comment (a new attribute) for rejection (REQ4).

Still from the point of view of object life cycle, the last thing missing is REQ5 – we need a direct transition from *Draft* to *Approved,* guarded on whether the expense admits automatic approval:

```
class Expense …
    private derived attribute automaticApproval : Boolean { self.amount < 50.0 };
    statemachine Status
        initial state Draft
            transition on call(submit) to Approved when { self.automaticApproval };
            transition on call(submit) to Submitted;
        end;
    … end;
end;
```

There are now only two requirements left. REQ6 can be satisfied with derived associations in Employee. Let's start with the listing of expenses in Draft (adding derived associations for filtering based on other statuses would be analogous):

```
class Employee …
    derived attribute draftExpenses : Expense[*] := {
        Expense#byStatus(self<-EmployeeExpenses->expenses, Expense::Status#Draft)
    };
end;
class Expense …
    private static query byStatus(expenses : Expense[*],
        toMatch : Status) : Expense[*];
    begin
        return expenses.select((e : Expense) : Boolean { e.status == toMatch });
    end;
end;
```

The expression *role1<-association->role2* traverses the association from role1 to role2. The helper query *byStatus* uses a built-in *select* operation to filter a set based on a criteria (a closure that takes an object and returns a boolean)[2].

Which leaves us with REQ7, our last requirement. In order to prevent an employee from approving her own expenses, we need a precondition based on the user currently logged in:

```
class Expense …
    operation approve()
        precondition must_be_another_user { not (System#user() == self.submitter) };
… end;
```

System is a built-in class that among other things gives access to the logged-in user.

And that does it. We have now a fully functional application that satisfies all requirements. Or do we? In order to ensure the application satisfies the requirements, we should encode the requirements as automated test cases. For example:

```
package tests;

[Test] class ExpenseRequirements
    operation expenseUnder50IsEligibleForAutomaticApproval();
    begin
        Assert#isTrue(Tests#declareExpense(49.9).automaticApproval);
        Assert#isTrue(not Tests#declareExpense(50.0).automaticApproval);
    end;
end;

end.
```

End users and other non-technical stakeholders can do further validation by exploring the application functionality via the generated user interface.

2 Closures and collection operations are features added to UML via a profile

4. Architecture

Cloudfier is a double-sided system: it is a development environment, and a deployment platform, each with their own audiences. As such, Cloudfier's architecture is easier to describe by looking at it from each perspective at a time.

4.1. Development Use Cases

During development, developers use Cloudfier to edit and validate their work, deploy a new version of the application and run automated tests. As the developer edits the content in the editor, or issues commands in the Shell page, REST requests are sent to server-side services for model validation, application deployment, database deployment, test running etc. These services read from and/or write to the model repository or the application database (directly or, in the case of the test runner, also via the model interpreter).

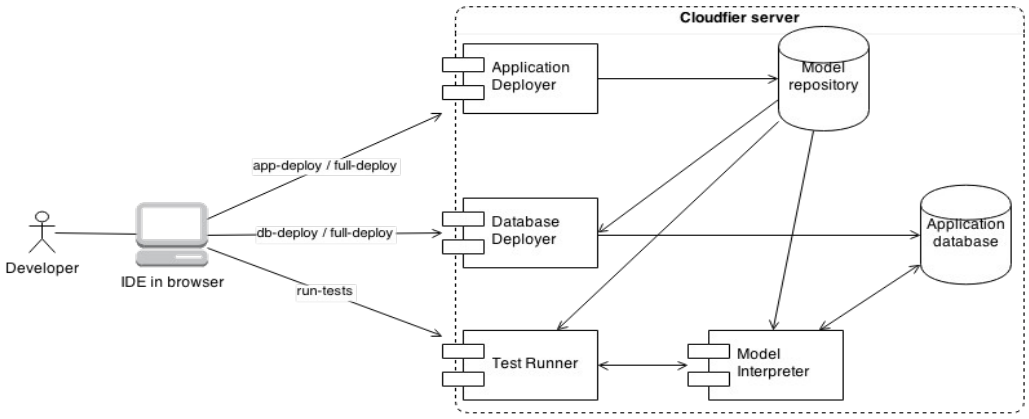

Figure 4. The architecture for development use cases

4.2. Production Use Cases

Once deployed, the application functionality becomes available via REST endpoints. The model repository is no longer modified but drives API rendering and model execution. The model interpreter also relies on the model repository to update and query data in the database (as the database schema matches the model metadata). The Data API performs all data updates and queries, and operation invocations via the model interpreter.

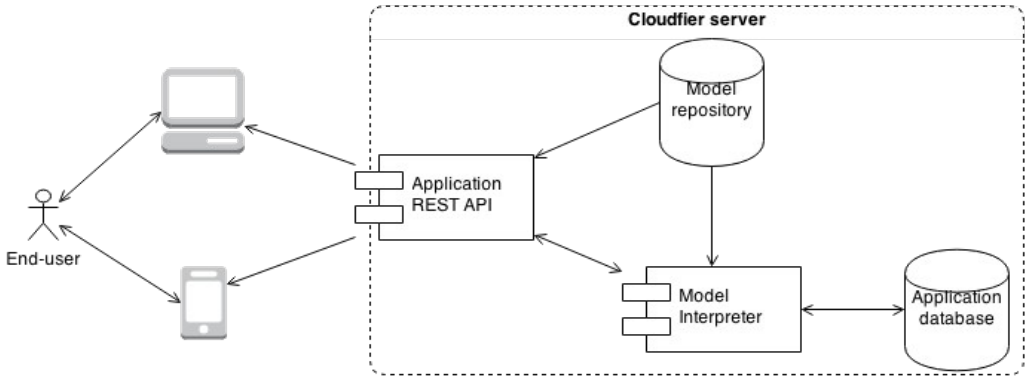

Figure 5. The architecture for application production use cases

5. Related Tools

Tools for generating full-fledged applications from executable models, such as Bridgepoint, Kennedy-Carter and Pathfinder, have existed for a long time, some even before UML existed. Those tools tend to be marketed as suitable for mission critical software development to large clients in the automotive, aviation, defense and telecommunication industries. Cloudfier is an attempt at bringing the same benefits to mainstream software development.

Also, in recent years, there has been no scarcity of frameworks that automate API creation, database schema creation, object-relational mapping etc for a domain model. However, we believe frameworks fall short of what the model-driven development approach that tools like Cloudfier can offer: 1) implementation languages lack important constructs that are needed for domain modeling, 2) they limit what platforms one can deploy to, and 3) and they require a full rewrite when migrating to a completely new platform.

References

Mellor, S. and Balcer, M. (2002). Executable UML: A Foundation for Model-Driven Architecture, Addison-Wesley.

Object Management Group, 2011. Unified Modeling Language Superstructure version 2.4.1. Available at <http://www.omg.org/spec/UML/2.4.1/>. Visited in June 2014.

TextUML, 2014. TextUML Toolkit web site. Available at <http://textuml.org>. Visited in June 2014.