Code Generation as a Service

Robert Crocombe and Dimitris Kolovos

Department of Computer Science, University of York, United Kingdom
{rjwc501,dimitris.kolovos}@york.ac.uk

Abstract. Code generation is an important tool in model-driven engineering. It eases the transition between designing models and implementing systems in code. This project aims to make code generation a web service, adding a layer of abstraction between the user's models and the generator used to produce their code.

1 Introduction

The goal of this project is to develop a web service that allows users to publish code generators and execute them online. Generators are publicly available so anyone can use them on their own models. A user uploads their models to be processed and receives the output files produced by their chosen generator.

The motivation for this project is having the ability to generate code from models without the need for locally-available software. This removes the effort of learning and maintaining the software from the user. It benefits the situation where the individual writing the code generator is different to the one designing the models, like a team or client. The model designer does not need to know how the generator works but can benefit from the output it creates using their own models. The same abstraction can be placed on the service itself. While the internal system may change - to be optimised or made faster for example - the API stays the same.

Code generation as a service also benefits situations where text is generated from very large models depicting a complex system. Computing in the cloud can have increased processing power and resources over a local machine, so code generation could take a shorter amount of time to complete, aiding the productivity of the user.

2 Project Background

An investigation of previous work in this area has been carried out and will be presented in this section. This process helps determine whether the project is novel enough to pursue, and the potential benefits of the finished system.

Recently, there has been an interest for model-driven engineering solutions in the cloud. Other fields of technology have, in recent years, adopted web-based services successfully. For example, cloud storage of files and document editing, integrated development environments (IDEs) accessed through the web, and cloud computation.

As people that make use of MDE are also software engineers, it is interesting to take a look at what side of software engineering the current services focus on. There are currently many services that provide standard IDE facilities like a code editor with syntax highlighting and error detection (such as Koding [8] and Cloud 9 [3]), some of which include code compilation (such as Visual Studio Online [10] and IDE One [11]). It is this aspect of these services that most closely resembles the aim of this project, as it involves uploading the source to the service, processing it and returning an output.

While there has been lots of progress made with online code compilation, code generation has been less prevalent. APIMATIC [2] is a service that will generate source code in multiple languages from a REST API model. Virtual Developer [4] is a company offering a marketplace for code generators and support in setting up their own software for code generation. Current solutions take a similar business model as some of the online IDE services by asking for a monthly fee and the tools are accessed via a web browser. Virtual Developer's distinction from this project is its goal to be more of a hosting platform for code generator technologies (like their own modelling tool or MetaEdit+ [5]), including having the client setup or pay to setup the tools themselves.

One of the most fully-featured services available online is GenMyModel [6]. It features a browser-based UML diagram editor and code generation from UML. Unlike APIMATIC it does not limit users to a REST API - anything can be modelled. It also appears to be easier to use than Virtual Developer as there's no setup to perform or self-hosted tools required. GenMyModel still limits the user to their own UML tools and the limited set of languages that can be output.

This project aims to have some novel differences in its implementation. Because it will primarily use Epsilon [9] to perform model-to-text transformations, the generated output can be made for any language. The service will be implemented with an API, so any client program can utilise it. An experienced developer could write a client for their own uses for example. Finally, generators can be reused or made public for others, improving the options available for users of the service.

3 Service Implementation

This section will describe how the service is used by a client as a step-by-step process, and detail how the service works internally. The service was written in Java, chosen because of its portability and maturity of support.

Representational State Transfer (REST) was chosen as the web service architecture because of the ease in which a client can be implemented to interface with a RESTful service. It was desirable to allow anyone to write a client for the service. Requests made to the service also work well with the RESTful style. For example, getting a list of generators with a GET request, or publishing a new one with a POST request.

3.1 Publishing a Generator

One of the main aims of the project is allowing users to execute generators that others have made, and reuse generators without having to upload them to the service again. To this end, generators are 'published' to the service and available for anyone to use on their own models. A system consists of an input, a process, and an output. Generators act as the processing stage of the system, with a model as the input and text as output. Because generators should be reusable, they have to be independent from the models they can parse.

Apache Ant [1], a tool for automating the software build process, is used by the service to execute generators. This design decision was made because of the advantages of having Ant handle all code execution instead of performing it inside the service. Epsilon already has Ant tasks for every language it supports. The Epsilon Generation Language (EGL) is a domain-specific language (DSL) used by Epsilon to generate text from models. Epsilon handles all execution of the EGL generator, the service only needs to execute the Ant build. This greatly simplifies the work the service has to do, as well as offer greater flexibility to the generator maker. They can include other code such as the Epsilon Validation Language which validates models. An Ant project allows for custom properties to be defined, removing the need for a separate file defining metadata about the generator.

```
<project name="pacs-emf-gen" default="main">
        <property name="model" value="" description="The input EMF model"/>
        <target name="main">
                <epsilon.emf.register file="test.ecore"/>
                <epsilon.emf.loadModel name="M" modelfile="${model}"
                        metamodeluri="http://cs.york.ac.uk/"
                        read="true" store="true"/>
                <epsilon.egl src="test.egl" outputroot="${outputRoot}">
                        <model ref="M"/>
                </epsilon.egl>
        </target>
</project>
```

Fig. 1. An example Ant build.xml file

Figure 1 is an example of how an Ant build file might be constructed for a generator. XML nodes starting with "epsilon" are tasks for generating the output text. The Ant build lists "test.ecore" as the metamodel in which all models need to conform to. "Test.egl" is also listed as the EGL file that will be executed against the model. Both these files need to be included in upload sent to the service.

While planning the service, a decision needed to be made about how generators would be stored/accessed on the service. The straightforward approach would be to directly upload the files to the service. Instead, it was decided that generators would be retrieved from GitHub [7] and cloned to service's file system. This adds several advantages over direct upload. It allows a generator to be updated automatically if one is modified using a Git pull request. There are indirect benefits to this approach as well. The generator can be uniquely identified by its GitHub name ('owner/repository') as administration is all handled by GitHub. The names are also guaranteed to be URL friendly. A generator can be worked on in collaboration and Git encourages source control and backup.

```
POST /service/generators?owner=robcrocombe&repo=emf-gen
```

Fig. 2. An example URL to publish a generator

Figure 2 gives an example of how the service can be invoked to publish a generator. Because GitHub is used, the publisher only needs to provide their username and the name of the repository as query parameters, they do not need to upload the files themselves. When a generator is published, the Git repository is cloned to a local directory on the server. To discover generators made by others, a client can make a GET request on */generators* to list all generators in the system, or get details on a specific one, such as its description and properties.

3.2 Generating Code

In this project, a job refers to the single process of creating a request for text to be generated from a model, executing that request, and returning the output to the client. A job request should select which generator to use and contain the necessary metadata required for the generator to run.

An important design decision made was how the client would wait for a job to be completed. The amount of time it takes for code generation to run depends on the size of the model and complexity of the generator. Jobs normally take longer than the usual HTTP request so there needed to be a suitable system for dealing with this. The solution was that a job POST request would initialise a separate thread for the job and return a unique ID. The client then uses the ID to poll the service for the status of the job, ideally at regular intervals. When the job is complete a status request will instead return a zip file containing the generated output.

When a client wishes to create a job, a POST request is sent to the */job* endpoint in the service. The request body has a 'multipart/form-data' content type. This allows files to be attached as fields in the form. Every request must contain a JSON file containing the metadata needed by the service.

Figure 3 is an example of the JSON file. Here, the client specifies which generator to use with the "githubOwner" and "repoName" fields. Any string properties can be given as an array here. The properties and files are both inserted into the build.xml file before Apache Ant executes it.

```
{
        githubOwner: "robcrocombe",
        repoName: "emf-gen",
        properties:
        [ {
                name: "debug",
                value: "true"
        } ],
        files:
        [ {
                fieldName: "file1",
                propertyName: "model",
                filePath: "test.model"
        } ]
}
```

Fig. 3. An example manifest.json file

For example, the file with “model” as the “propertyName” will be put into the “model” property in figure 1.

Each item in the “files” array contains three fields. “fieldName” links the item where the file has been attached in the form. These two fields must have the same name. “propertyName” is the name of property in the Ant build this file is for. “filePath” is the path of the file including the extension. This field must be a relative path. This means, for example, if the field contains “directory/sub-directory/test.model” the file will be put into that folder structure relative to the job folder on the server.

This upload method also supports folders. To do this, each file must be included as an item in the “files” array, with their file path including the folder they belong to. The property name is not included, as the file on its own is not a property value. Instead, the folder is added to the “properties” array, the name field being the name of the property in the Ant build.xml, and the value being the name of the folder. This works because files are still Ant properties, they just have to be declared separately so they can be uploaded to the service.

Figure 4 is a sequence diagram summarising how a job is created, how its status is polled, and the output received by the client. The advantage of this solution is that it scales well. If the processing time is short, the first poll would return the output. If it is long, there is little overhead to requesting the status regularly. An alternative solution was to open a small server on the client program and when the job is complete it will send the output to the client like fired event. This was not as preferable because it was desired that the client would be lightweight and simple to implement. The service should also stay RESTful so the client should be the only entity making requests.

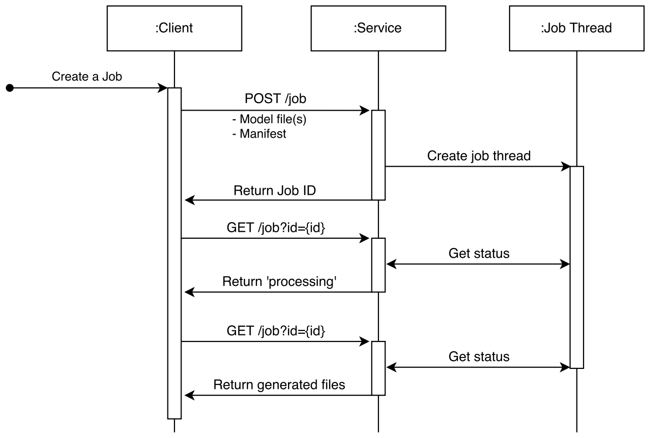

Fig. 4. Sequence diagram of a processing job

4 Conclusions

This paper has covered reasons for pursuing this project, its development progress, and how the service can be consumed. If this project was to be made available for public use some features would have to be developed further. For example, currently only public GitHub repositories can be published to the service, as private repositories would need authentication. Private repositories would be important for users if they wish to work on closed-source projects.

Fleshing out generator publishing would also be desirable. For example adding user accounts to keep track of generators and allow them to be updated by their owners.

References

1. Apache Software Foundation: Apache ant (Jul 2015), http://ant.apache.org
2. APIMATIC: Apimatic (Jul 2015), http://apimatic.io
3. Cloud9 IDE, Inc: Cloud 9 development environment (Jul 2015), http://c9.io
4. Generative Software GmbH: Virtual developer (Jul 2015), http://www.virtual-developer.com
5. Generative Software GmbH: Virtual developer modelling (Jul 2015), http://www.virtual-developer.com/en/modeling
6. GenMyModel: Genmymodel (Jul 2015), http://www.genmymodel.com
7. GitHub, Inc: Github (Jul 2015), http://github.com
8. Koding, Inc: Koding cloud development environment (Jul 2015), http://koding.com
9. Kolovos, D. et al.: Epsilon (Jul 2015), http://www.eclipse.org/epsilon
10. Microsoft: Visual studio online (Jul 2015), http://www.visualstudio.com/products/what-is-visual-studio-online-vs
11. Sphere Research Labs: IDE One (Jul 2015), http://ideone.com