# Build a TVML App with Grails

Sergio del Amo

Version 1.0, 2017-01-19

# Table of Contents

# Getting Started

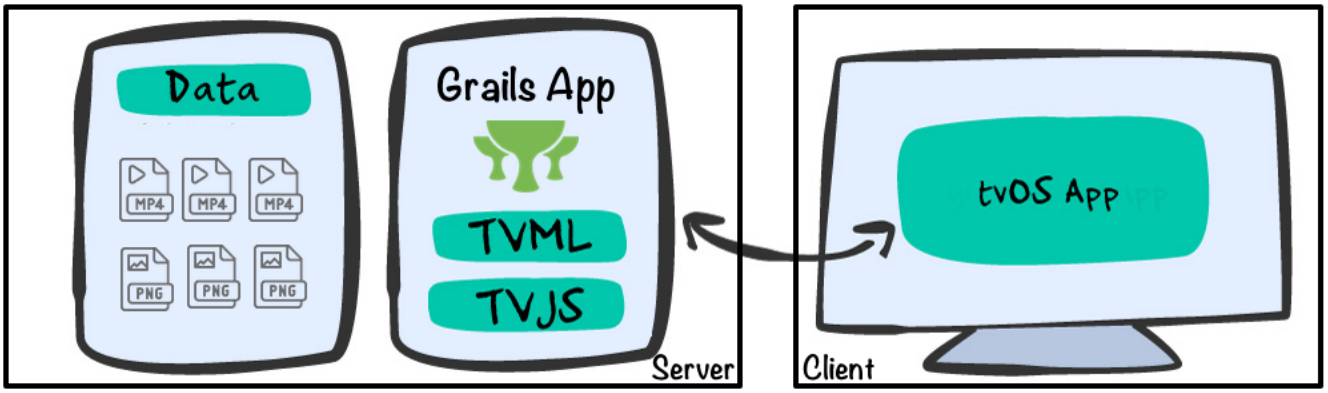
In this guide, you write a TVML application.

> TVML apps provide you with a way to create apps for Apple TV using TVMLKit JavaScript (TVMLKit JS), the Apple TV Markup Language (TVML), and the TVMLKit framework. Every template has a specific design that follows Apple style guidelines, allowing you to quickly create gorgeous looking apps.

The TVML App, built during this guide, displays an Apple TVML Stack Template of Grails Quickcasts. Grails Quickcasts are short videos of pure Grails coding. When the user selects one, an Apple TVML Product Template shows the Quickcast detail. The user can reproduce the video by selecting the play button.

# Architecture

This guide features a Client Server architecture. The Client, a tvOS application, connects to a Grails App, the servers, which responds with TVMLKit Js and TVML documents.




We have separated the Grails App from the Data ( mp4 files and Png files ); videos and thumbnails. We use MAMP to run a local apache server at port 8888 pointing to the folder data. During this guide you will see references to mp4 files such as:

```
releaseYear: 2016,
```

In a Production environment, you may want to host those files in a service such as Amazon Simple Store Service - AWS S3

# What you will need

To complete this guide, you will need the following:

- Some time on your hands
- A decent text editor or IDE
- JDK 1.7 or greater installed with `JAVA_HOME` configured appropriately
- Latest stable version of XCode. We wrote this guide with Xcode 8.2.1.

# How to complete the guide

To complete the Grails Application developed in this guide, you will need to checkout the source from Github and work through the steps presented by the guide.

To get started do the following:

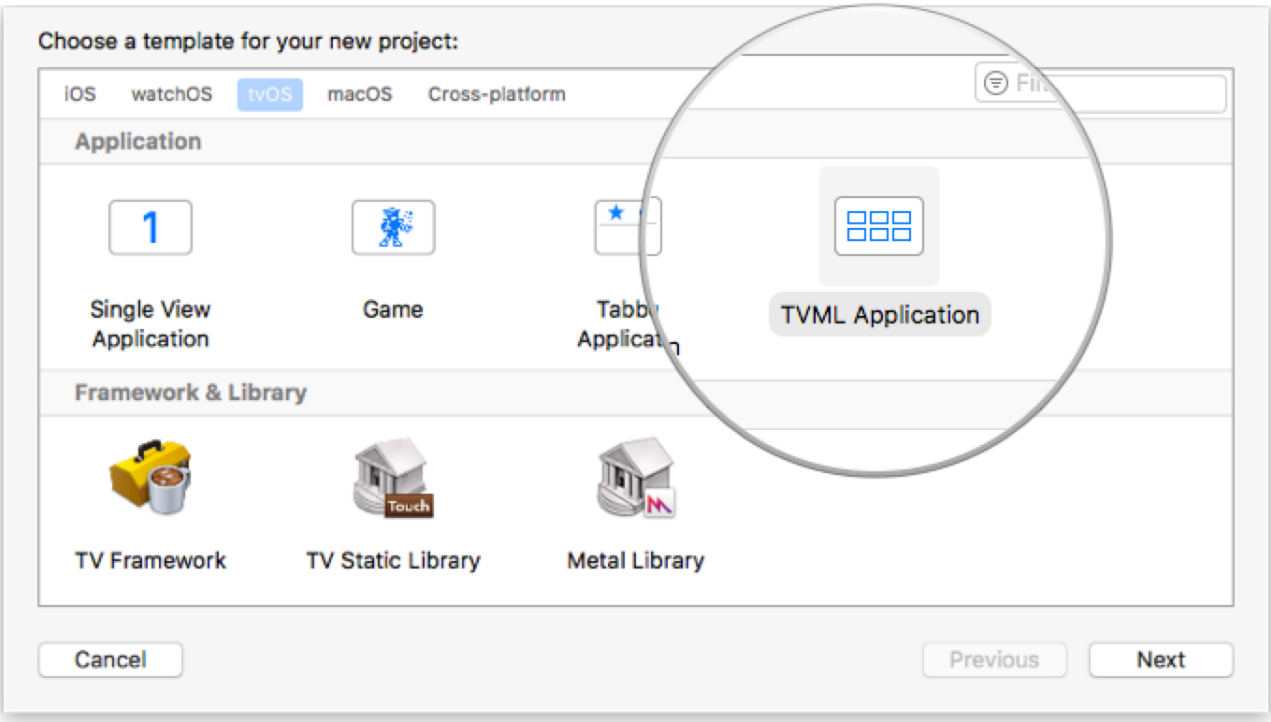
- Download and unzip the source or if you already have Git: `git clone https://github.com/grails-guides/grails-tvmlapp.git`
- `cd` into `grails-guides/grails-tvmlapp/initial`
- Head on over to the next section

| | |
|---|---|
| **TIP** | You can go right to the completed example if you `cd` into `grails-guides/grails-tvmlapp/complete` |

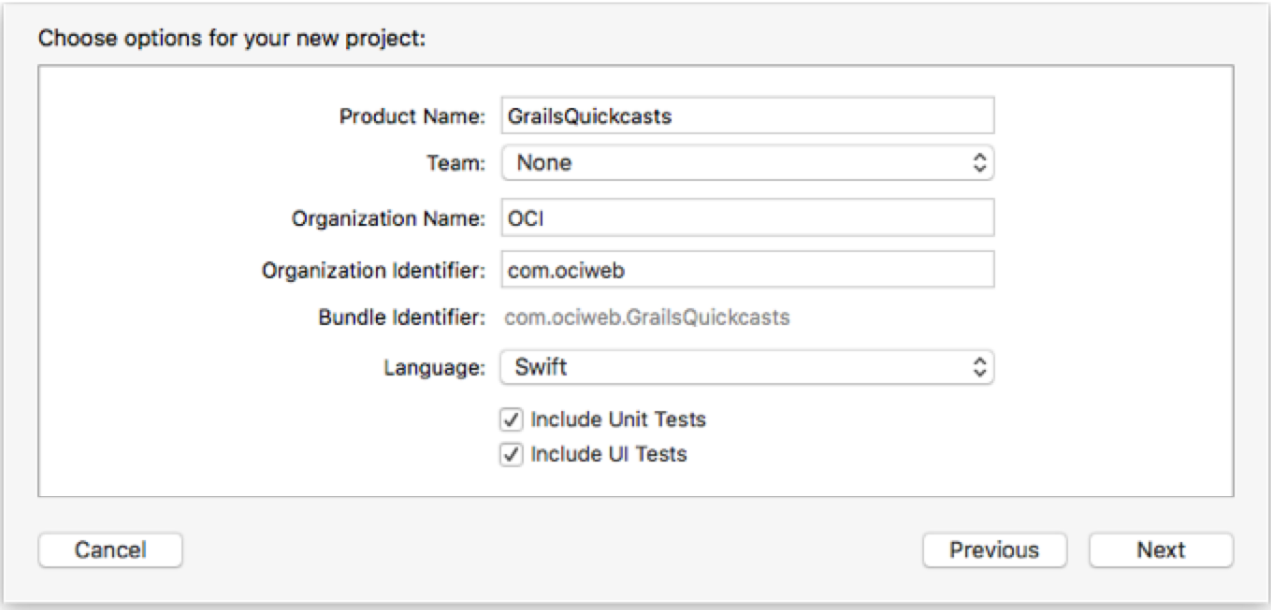
# Writing the TVML Client

To create a TVML application and connect it to a Grails Server we need to modify the variables *tvBaseURL* and *tvBootURL*.

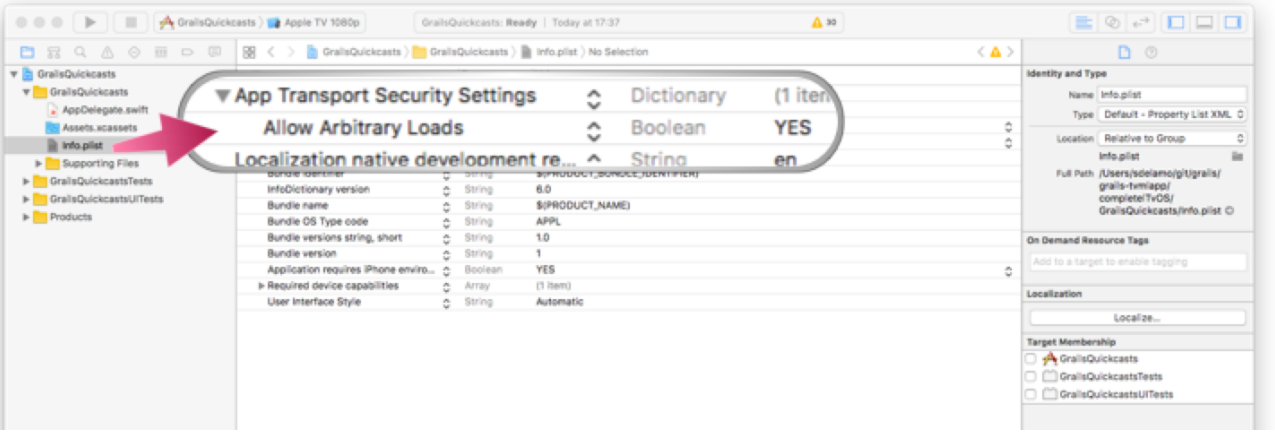
Create a TvOS project with the TVML application template.




Choose Swift as the language.

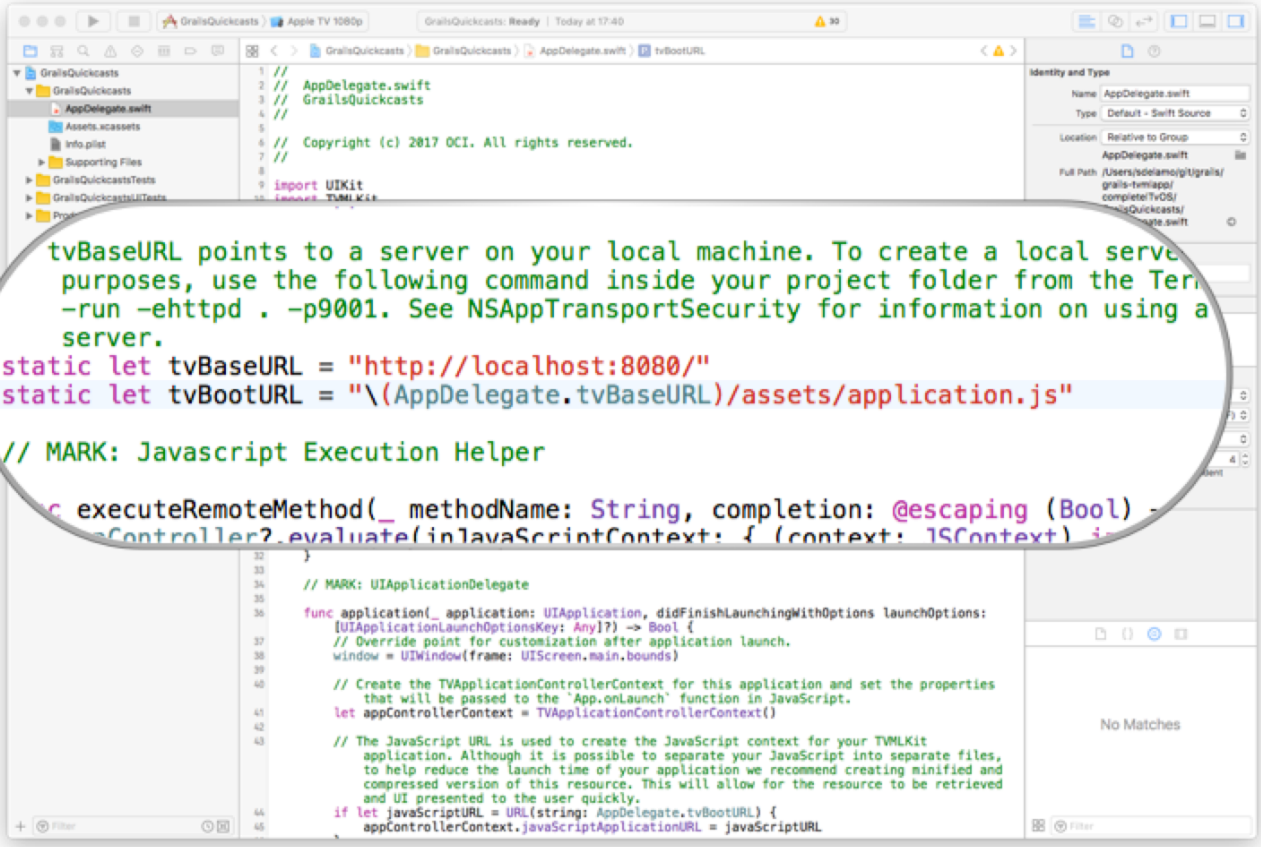



To connect it to a local Grails server configure *App Transport Security Settings* to *Allow Arbitrary Loads*




Modify *AppDelegate.swift* and set the static variables *tvBaseURL* and *tvBootURL*

```
static let tvBaseURL = "http://localhost:8080/"
static let tvBootURL = "\(AppDelegate.tvBaseURL)/assets/application.js"
```



# Writing the TVML Grails Application

The initial Grails application, which you can find in the *initial* folder, was created with the command:

```
grails create-app --profile rest-api -features=hibernate5,markup-views,asset-pipeline
```

**TIP**

Did you know you can download a complete Grails project without installing any additional tools? Go to start.grails.org and use the **Grails Application Forge** to generate your Grails project. You can choose your project type (Application or Plugin), pick a version of Grails, and choose a Profile - then click "Generate Project" to download a ZIP file. No Grails installation necessary!

You can even download your project from the command line using a HTTP tool like `curl` (see start.grails.org for API documentation):

```
curl -O start.grails.org/myapp.zip -d version=3.2.4 -d profile=angular
```

# TVMLKit JS Utils

TVMLKit JS

> TVMLKit JS is a JavaScript API framework specifically designed to work with Apple TV and TVML. TVMLKit JS incorporates the basic functionality found in JavaScript along with specialized APIs designed for Apple TV. Using TVMLKit JS, you are able to load and display TVML templates, play media streams, load customized content, and control the flow of your app. For more information, see TVJS Framework Reference.

Add a javascript file to our project. It defines a series of methods nearly identical as those described in the section Playing Media of TVML Programming Guide.

These methods allow to push pages into the navigation stack, fetch documents and play media.

*Listing 1. /grails-app/assets/javascripts/tvmlkit.js*

```
function loadingTemplate() {
    var template =
'<document><loadingTemplate><activityIndicator><text>Loading</text></activityIndicator
></loadingTemplate></document>';
    var templateParser = new DOMParser();
    var parsedTemplate = templateParser.parseFromString(template, "application/xml");
    navigationDocument.pushDocument(parsedTemplate);
}

function getDocument(extension) {
    var templateXHR = new XMLHttpRequest();
    var url = baseURL + extension;

    loadingTemplate();
    templateXHR.responseType = "document";
    templateXHR.addEventListener("load", function() {pushPage(templateXHR.responseXML
);}, false);
    templateXHR.open("GET", url, true);
    templateXHR.send();
}

function pushPage(document) {
    var currentDoc = getActiveDocument();
    if (currentDoc.getElementsByTagName("loadingTemplate").item(0) == null) {
        navigationDocument.pushDocument(document);
    } else {
        navigationDocument.replaceDocument(document, currentDoc);
    }
}

function playMedia(videourl, mediaType) {
    var singleVideo = new MediaItem(mediaType, videourl);
    var videoList = new Playlist();
    videoList.push(singleVideo);
    var myPlayer = new Player();
    myPlayer.playlist = videoList;
    myPlayer.play();
}
```

# TV Boot Url

This guide uses Asset Pipeline Plugin

> The Asset-Pipeline is a plugin used for managing and processing static assets in JVM applications primarily via Gradle (however not mandatory). Asset-Pipeline functions include processing and minification of both CSS and JavaScript files

Create a javascript file which will be the entry to our TVML Grails App:

*Listing 2. /grails-app/assets/javascripts/application.js*

```
// This is a manifest file that'll be compiled into application.js.
//
// Any JavaScript file within this directory can be referenced here using a relative
path.
//
// You're free to add application-wide JavaScript to this file, but it's generally
better
// to create separate JavaScript files as needed.
//
//= require tvmlkit.js
//= require_tree .
//= require_self
var baseURL;

App.onLaunch = function(options) {
    baseURL = options.BASEURL;
    var extension = "quickcast";
    getDocument(extension);
}
```

Please note, how the line displayed in the next Listing includes the TVMLKit JS file discussed in the previous section.

*Listing 3. /grails-app/assets/javascripts/application.js*

```
//= require tvmlkit.js
```

The initial TVML document is the XML rendered at *http://localhost:8080/quickcast*

The next line designates the initial document:

*Listing 4. /grails-app/assets/javascripts/application.js*

```
var extension = "quickcast";
```

# Domain class

Create a persistent entity to store Quickcasts. Most common way to handle persistence in Grails is the use of Grails Domain Classes:

> A domain class fulfills the M in the Model View Controller (MVC) pattern and represents a persistent entity that is mapped onto an underlying database table. In Grails a domain is a class that lives in the grails-app/domain directory.

```
./grailsw create-domain-class Quickcast
```

*Quickcast* domain class is our data model. We define different properties to store the Quickcast characteristics.

*Listing 5. /grails-app/domain/com/ociweb/quickcasts/Quickcast.groovy*

```
package com.ociweb.quickcasts

class Quickcast {
    String title
    String subtitle
    String description
    int durationMinutes
    int durationSeconds
    int releaseYear
    String heroImg
    String videoUrl
    static hasMany = [authors: String]

    static constraints = {
        description nullable: true
    }

    static mapping = {
        description type: 'text'
    }
}
```

With a Unit Test, we test the property body is option

*Listing 6. /src/test/groovy/com/ociweb/quickcasts/QuickcastSpec.groovy*

```
package com.ociweb.quickcasts

import grails.test.mixin.TestFor
import spock.lang.Specification

/**
 * See the API for {@link grails.test.mixin.domain.DomainClassUnitTestMixin} for usage
instructions
 */
@TestFor(Quickcast)
class QuickcastSpec extends Specification {

    void "test description is optional"() {
        expect:
            new Quickcast(description: null).validate(['description'])
    }
}
```

We load some test data in *BootStrap.groovy*

*Listing 7. /grails-app/init/com/ociweb/quickcasts/BootStrap.groovy*

```
package com.ociweb.quickcasts

class BootStrap {

    def init = { servletContext ->

        new Quickcast(title: 'Interceptors - Grails 3',
                subtitle: '#1 - Grails Quickcast from OCI',
                durationMinutes: 17,
                durationSeconds: 01,
                releaseYear: 2016,
                heroImg: 'http://localhost:8888/quickcast_interceptor.png',
                videoUrl: 'http://localhost:8888/grails_quickcast_1_interceptors.mp4',
                authors: ['Jeff Scott Brown'],
                description: 'This Quickcast assumes only basic familiarity with
Groovy (which is pretty darn expressive anyway) and the MVC concept (which you already
know). Also serves as an excellent introduction to the interceptor pattern in any
language, because Grails\' behind-the-scenes legwork lets you focus on the logic of
the pattern.'
        ).save()

        new Quickcast(title: 'JSON Views - Grails 3 ',
                subtitle: '#2 - Grails Quickcast from OCI',
                videoUrl: 'http://localhost:8888/grails_quickcast_2_jsonviews.mp4',
                heroImg: 'http://localhost:8888/quickcast_jsonviews.png',
                durationMinutes: 15,
```

```
                durationSeconds: 40,
                releaseYear: 2016,
                description: '''
n a delightful and informative 15 minutes, Brown probes JSON views. Beginning with a
Grails 3.1.1 application, created with a standard web profile, Brown added a few
custom domain classes. The artist class has albums associated with it, and is
annotated with grails.rest.Resource.

The ultimate goal is publishing a REST API under /artists for managing instances of
the artist class, and to support the JSON and XML formats.

Brown uses music examples, including Space Oddity by David Bowie (RIP), and Close to
the Edge by Yes. Sending a request to /artists gives a list of artists all of whom
have albums associated with them. While the app is running in development mode, the
JSON files can be altered and the effects of those changes can be seen real-time in
the application. For example, switching "ArtistName": "Riverside" to "theArtistName":
"Riverside".

This Quickcast assumes basic knowledge of Grails, JSON, and REST APIs. Check out Brown
's neat intro to JSON views!
''',
                authors: ['Jeff Scott Brown']).save()

        new Quickcast(title: 'Multi-Project Builds - Grails 3',
                subtitle: '#3 - Grails Quickcast from OCI',
                heroImg: 'http://localhost:8888/quickcast_multiprojectbuilds.png',
                videoUrl: 'http://localhost:8888/grails_quickcast_2_jsonviews.mp4',
                durationMinutes: 14,
                durationSeconds: 28,
                releaseYear: 2016,
                description: '''
In this Quickcast, Graeme Rocher, Head of Grails Development at OCI, walks you through
multi-project builds in Grails. Grails does a few handy things with multi-project
builds and plugins, not the least of which being that Grails compiles your plugins
first and puts the class and resources of those plugins directly in the classpath.
This lets you make changes to your plugins and instantly see those changes in your
build.
''',
                authors: ['Graeme Rocher']).save()

        new Quickcast(title: 'Angular Scaffolding - Grails 3',
                subtitle: '#4 - Grails Quickcast from OCI',
                heroImg: 'http://localhost:8888/quickcast_angularscaffolding.png',
                videoUrl: 'http://localhost:8888/grails_quickcast_2_jsonviews.mp4',
                durationMinutes: 9,
                durationSeconds: 27,
                releaseYear: 2016,
                description: '''
In this Quickcast, OCI Engineer James Kleeh walks you through using the Angular
Scaffolding for Grails to build a fully functional web app, using a simple blog format
for demonstration. The tutorial explains how to have Grails set up a REST endpoint and
```

```
all the Angular modules needed to get the web app running.
''',
                authors: ['James Kleeh']).save()

        new Quickcast(title: 'Retrieving Runtime Config Values - Grails 3',
                subtitle: '#5 - Grails Quickcast from OCI',
                heroImg:
'http://localhost:8888/quickcast_retrievingruntimeconfigvalues.png',
                videoUrl: 'http://localhost:8888/grails_quickcast_2_jsonviews.mp4',
                durationMinutes: 17,
                durationSeconds: 51,
                releaseYear: 2016,
                description: '''
In the fifth Grails QuickCast, Grails co-founder, Jeff Scott Brown, highlights some of
the great features of the Grails framework.  In less than 18 minutes, Jeff describes
several techniques for retrieving configuration values at runtime, and discusses the
pros and cons of these different techniques. For this Quickcast, you'll need no more
than a basic understanding of Grails. The Grails Quickcast series is brought to you
from OCI and DZone.

Grails leverages the "Convention Over Configuration" design paradigm, which functions
to decrease the number of decisions that a developer using the framework is required
to make without losing flexibility. This is one of the main reasons why Grails
significantly increases developer productivity!

While Grails applications often involve considerably less configuration than similar
applications built with other frameworks, some configuration may still be necessary.
In this short video, Jeff shares a number of mechanisms that make it easy for Grails
developers to define configuration values, and to gain access to those configuration
values at runtime.
''',
                authors: ['Jeff Scott Brown']).save()


        new Quickcast(title: 'Developing Grails Application with IntelliJ IDEA -
Grails 3',
                subtitle: '#6 - Grails Quickcast from OCI',
                heroImg:
'http://localhost:8888/quickcast_developinggrailsappswithintellij.png',
                videoUrl: 'http://localhost:8888/grails_quickcast_2_jsonviews.mp4',
                durationMinutes: 22,
                durationSeconds: 42,
                releaseYear: 2016,
                description: '''
In the sixth Grails QuickCast, Grails co-founder, Jeff Scott Brown, introduces several
tips and tricks related to building Grails 3 applications in IDEA. The Grails
Quickcast series is brought to you from OCI and DZone.

Grails 3 is a high productivity framework for building web applications for the JVM.
IntelliJ IDEA is a high productivity Integrated Development Environment (IDE) for
building a variety of types of applications. IDEA has always had really great support
```

```
for building Grails applications and, in particular, has the very best support of any
IDE for doing development with Grails 3.
'''
    ',
                authors: ['Jeff Scott Brown']).save()


    }
    def destroy = {
    }
}
```

# Stack Template

We show first an Apple TVML Stack Template with the Grails Quickcasts:


Create a Grails Controller to handle requests

> A controller fulfills the C in the Model View Controller (MVC) pattern and is responsible for handling web requests. In Grails a controller is a class with a name that ends in the convention "Controller" and lives in the grails-app/controllers directory.

```
./grailsw create-controller com.ociweb.quickcasts.Quickcast
```

A request to *http://localhost:8080/quickcast* executes the *index* action of *QuickcastController*

*Listing 8. /grails-app/controllers/com/ociweb/quickcasts/QuickcastController.groovy*

```
package com.ociweb.quickcasts

class QuickcastController {
    static responseFormats = ['xml']
    def index() {
        [quickcasts: Quickcast.findAll()]
    }

}
```

The *QuickcastController* index method renders every *Quickcast* using the next Markup View

> Markup views allow rendering of XML responses using Groovy's MarkupTemplateEngine

*Listing 9. /grails-app/views/quickcast/index.gml*

```
import com.ociweb.quickcasts.Quickcast

model {
    Iterable<Quickcast> quickcasts
}

document {
    stackTemplate {
        banner {
            title this.g.message(code: 'quickcasts.title')
        }
        collectionList {
            shelf {
                section {
                    quickcasts.each { quickcast ->
                        lockup(onselect: "getDocument('quickcast/${quickcast.id}')") {
                            img(src: quickcast.heroImg, width: 150, height: 226)
                            title quickcast.title
                        }
                    }
                }
            }
        }
    }
}
```

Markup Views are written in Groovy, end with the file extension *gml* and reside in the *grails-app/views* directory.

Write a functional test to test the XML generated is what we expect.

*Listing 10. /src/integration-test/groovy/com/ociweb/quickcasts/QuickcastControllerIndexSpec.groovy*

```
package com.ociweb.quickcasts

import grails.plugins.rest.client.RestBuilder
import grails.test.mixin.integration.Integration
import grails.transaction.Rollback
import groovy.xml.XmlUtil
import org.custommonkey.xmlunit.XMLUnit
import spock.lang.Specification
import org.springframework.beans.factory.annotation.Value
```

```
@Rollback
@Integration
class QuickcastControllerIndexSpec extends Specification {

    @Value('${local.server.port}')
    Integer serverPort

    def "test stack template is rendered correctly"() {

        given:
        def expected = '''<document>
    <stackTemplate>
        <banner>
            <title>Grails Quickcasts</title>
        </banner>
        <collectionList>
            <shelf>
                <section>
                    <lockup onselect="getDocument('quickcast/1')">
                        <img src="http://localhost:8888/quickcast_interceptor.png"
width="150" height="226"/>
                        <title>Interceptors - Grails 3</title>
                    </lockup>
                    <lockup onselect="getDocument('quickcast/2')">
                        <img src="http://localhost:8888/quickcast_jsonviews.png"
width="150" height="226"/>
                        <title>JSON Views - Grails 3</title>
                    </lockup>
                    <lockup onselect="getDocument('quickcast/3')">
                        <img
src="http://localhost:8888/quickcast_multiprojectbuilds.png" width="150"
height="226"/>
                        <title>Multi-Project Builds - Grails 3</title>
                    </lockup>
                    <lockup onselect="getDocument('quickcast/4')">
                        <img
src="http://localhost:8888/quickcast_angularscaffolding.png" width="150"
height="226"/>
                        <title>Angular Scaffolding - Grails 3</title>
                    </lockup>
                    <lockup onselect="getDocument('quickcast/5')">
                        <img
src="http://localhost:8888/quickcast_retrievingruntimeconfigvalues.png" width="150"
height="226"/>
                        <title>Retrieving Runtime Config Values - Grails 3</title>
                    </lockup>
                    <lockup onselect="getDocument('quickcast/6')">
                        <img
src="http://localhost:8888/quickcast_developinggrailsappswithintellij.png" width="150"
height="226"/>
                        <title>Developing Grails Application with IntelliJ IDEA -
```

```
Grails 3</title>
                    </lockup>
                </section>
            </shelf>
        </collectionList>
    </stackTemplate>
</document>
'''
        when:
        RestBuilder rest = new RestBuilder()
        def resp = rest.get("http://localhost:${serverPort}/quickcast")

        XMLUnit.setIgnoreWhitespace(true)
        XMLUnit.setNormalizeWhitespace(true)

        then:
        resp.status == 200
        XMLUnit.compareXML(XmlUtil.serialize(resp.xml), expected).identical()
    }
}
```

To facilitate XML comparison in the tests, we include *XMLUnit* as a dependency.

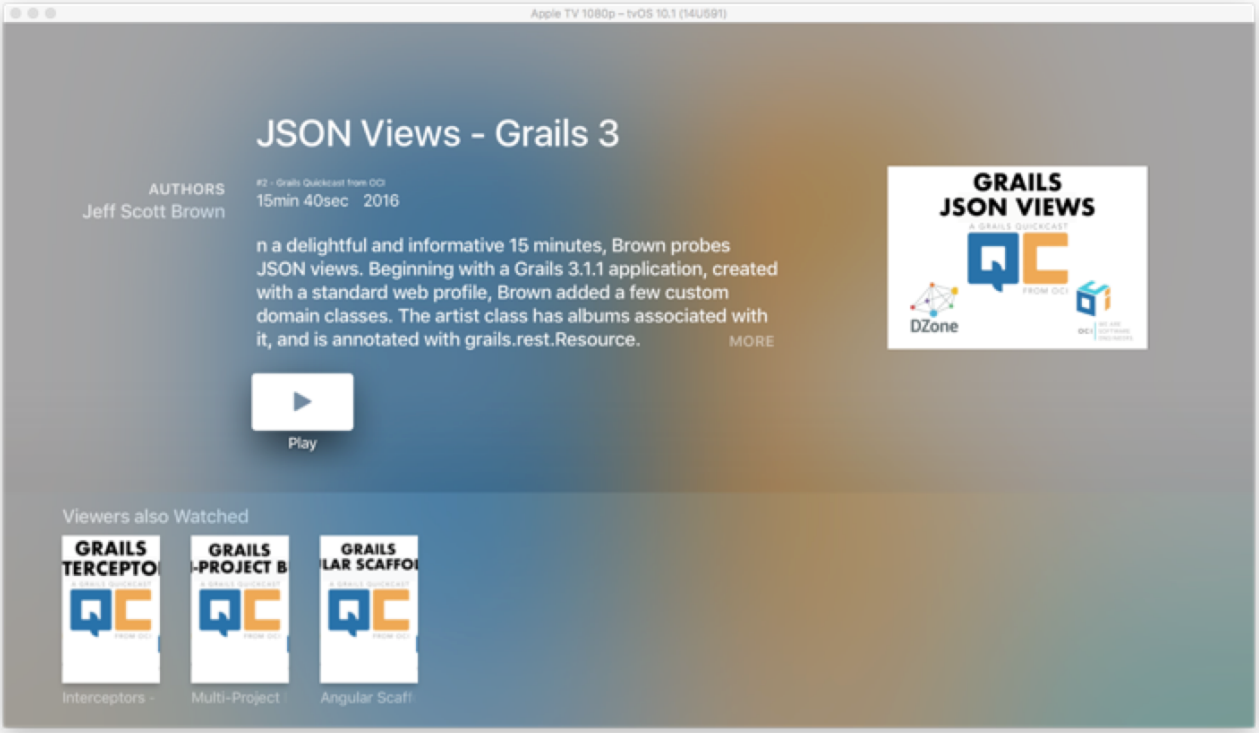
*Listing 11. /build.gradle*

```
compile "xmlunit:xmlunit:1.6"
```

# Product Template

When we select the first Quickcast, the document residing in *http://localhost:8080/quickast/1* is pushed into the navigation stack

To render a Quickcast we use the Apple TVML Product Template.




A request to *http://localhost:8080/quickcast/1* executes the *show* action of *QuickcastController*

*Listing 12. /grails-app/controllers/com/ociweb/quickcasts/QuickcastController.groovy*

```
package com.ociweb.quickcasts

class QuickcastController {
    static responseFormats = ['xml']

    def relatedQuickcastsService
    def show() {
        def quickcast = Quickcast.get(params.id)
        def relatedQuickcasts = relatedQuickcastsService.findAllRelatedQuickcasts
(quickcast)
        [quickcast: quickcast, relatedQuickcasts: relatedQuickcasts]
    }
```

The model passed to our view is the requested Quickcast and a series of related videos.

We configure in *application.yml* how many related quickcasts we want to show.

*Listing 13. /grails-app/conf/application.yml*

```
---
ociweb:
    quickcasts:
        numberOfRelatedQuickcasts: 3
---
```

Fetching of related quickcasts is encapsulated in a Service.

*Listing 14. /grails-app/services/com/ociweb/quickcasts/RelatedQuickcastsService.groovy*

```
package com.ociweb.quickcasts

import grails.config.Config
import grails.core.support.GrailsConfigurationAware
import grails.transaction.Transactional

@Transactional
class RelatedQuickcastsService implements GrailsConfigurationAware {

    int numberOfRelatedQuickcasts

    @Override
    void setConfiguration(Config co) {
        numberOfRelatedQuickcasts = co.getRequiredProperty(
'ociweb.quickcasts.numberOfRelatedQuickcasts', Integer)
    }

    List<Quickcast> findAllRelatedQuickcasts(Quickcast quickcast) {
        def criteria = Quickcast.createCriteria()
        criteria.list {
            ne('id', quickcast?.id)
            maxResults(numberOfRelatedQuickcasts)
        } as List<Quickcast>
    }
}
```

The *QuickcastController* index method renders a *Quickcast* using the next Markup View

*Listing 15. /grails-app/views/quickcast/show.gml*

```
import com.ociweb.quickcasts.Quickcast

model {
    Quickcast quickcast
    Iterable<Quickcast> relatedQuickcasts
}

document {
```

```
    productTemplate {
        banner {
            infoList {
                info {
                    header {
                        title this.g.message(code: 'quickcast.authors.header')
                    }
                    quickcast.authors.each { author ->
                        text "$author"
                    }
                }
            }
            stack {
                title quickcast.title
                subtitle quickcast.subtitle
                row {
                    text "${quickcast.durationMinutes}min ${quickcast.durationSeconds
}sec"
                    text quickcast.releaseYear
                }
                description(allowsZooming: "true", moreLabel: "more", "${quickcast
.description}")
                row {
                    buttonLockup(onselect: "playMedia('${quickcast.videoUrl}',
'video')") {
                        badge(src: "resource://button-play")
                        title "Play"
                    }
                }
            }
            heroImg(src: quickcast.heroImg)
        }

        shelf {
            header {
                title this.g.message(code: 'quickcast.cross.sell.header')
            }
            section {
                relatedQuickcasts.each { relatedQuickcast ->
                    lockup(onselect: "getDocument('quickcast/${quickcast.id}')") {
                        img(src: relatedQuickcast.heroImg, width: 150, height: 226)
                        title relatedQuickcast.title
                    }
                }
            }
        }
    }
}
```

With a functional test, we test the XML rendered is what we expected.

*Listing 16. /src/integration-test/groovy/com/ociweb/quickcasts/QuickcastControllerShowSpec.groovy*

```
package com.ociweb.quickcasts

import grails.plugins.rest.client.RestBuilder
import grails.test.mixin.integration.Integration
import grails.transaction.Rollback
import groovy.xml.XmlUtil
import org.custommonkey.xmlunit.XMLUnit
import spock.lang.Specification
import org.springframework.beans.factory.annotation.Value

@Rollback
@Integration
class QuickcastControllerShowSpec extends Specification {

    @Value('${local.server.port}')
    Integer serverPort

    def "test product template is rendered correctly"() {
        given:
        def expected = '''<document>
    <productTemplate>
        <banner>
            <infoList>
                <info>
                    <header>
                        <title>Authors</title>
                    </header>
                    <text>Jeff Scott Brown</text>
                </info>
            </infoList>
            <stack>
                <title>Interceptors - Grails 3</title>
                <subtitle>#1 - Grails Quickcast from OCI</subtitle>
                <row>
                    <text>17min 1sec</text>
                    <text>2016</text>
                </row>
                <description allowsZooming="true" moreLabel="more">This Quickcast
assumes only basic familiarity with Groovy (which is pretty darn expressive anyway)
and the MVC concept (which you already know). Also serves as an excellent introduction
to the interceptor pattern in any language, because Grails' behind-the-scenes legwork
lets you focus on the logic of the pattern.</description>
                <row>
                    <buttonLockup
onselect="playMedia('http://localhost:8888/grails_quickcast_1_interceptors.mp4',
'video')">
                        <badge src="resource://button-play"/>
                        <title>Play</title>
                    </buttonLockup>
```

```
                </row>
            </stack>
            <heroImg src="http://localhost:8888/quickcast_interceptor.png"/>
        </banner>
        <shelf>
            <header>
                <title>Viewers also Watched</title>
            </header>
            <section>
                <lockup onselect="getDocument('quickcast/1')">
                    <img src="http://localhost:8888/quickcast_jsonviews.png"
width="150" height="226"/>
                    <title>JSON Views - Grails 3</title>
                </lockup>
                <lockup onselect="getDocument('quickcast/1')">
                    <img src="http://localhost:8888/quickcast_multiprojectbuilds.png"
width="150" height="226"/>
                    <title>Multi-Project Builds - Grails 3</title>
                </lockup>
                <lockup onselect="getDocument('quickcast/1')">
                    <img src="http://localhost:8888/quickcast_angularscaffolding.png"
width="150" height="226"/>
                    <title>Angular Scaffolding - Grails 3</title>
                </lockup>
            </section>
        </shelf>
    </productTemplate>
</document>
'''
        when:
        RestBuilder rest = new RestBuilder()
        def resp = rest.get("http://localhost:${serverPort}/quickcast/1")

        XMLUnit.setIgnoreWhitespace(true)
        XMLUnit.setNormalizeWhitespace(true)

        then:
        resp.status == 200
        XMLUnit.compareXML(XmlUtil.serialize(resp.xml), expected).identical()
    }
}
```

# Internationalization

Internationalization is a first-class citizen in Grails.

> Grails supports Internationalization (i18n) out of the box by leveraging the underlying Spring MVC internationalization support. With Grails you are able to customize the text that appears in a view based on the user's Locale.

To render such a snippet:

```
<header>
    <title>Authors</title>
</header>
```

We use this code:

```
header {
    title this.g.message(code: 'quickcast.authors.header')
}
```

The message codes and default values are defined in *messages.properties*

*Listing 17. /grails-app/i18n/messages.properties*

```
resouce.button.play=Play
quickcast.authors.header=Authors
quickcast.cross.sell.header=Viewers also Watched
quickcasts.title=Grails Quickcasts
```

Including a Spanish localization is easy. Include the localized message codes in *messages_es.properties*

*Listing 18. /grails-app/i18n/messages.properties*

```
resouce.button.play=Reproducir
quickcast.authors.header=Autores
quickcast.cross.sell.header=Otros usuarios también vieron
quickcasts.title=Grails Quickcasts
```

# Running the Application

To run the Grails application use the *./gradlew bootRun* command which will start the application on port 8080.

Run the Xcode project and your Apple TvOS app will connect to your Grails Backend.