Ganttis documentation

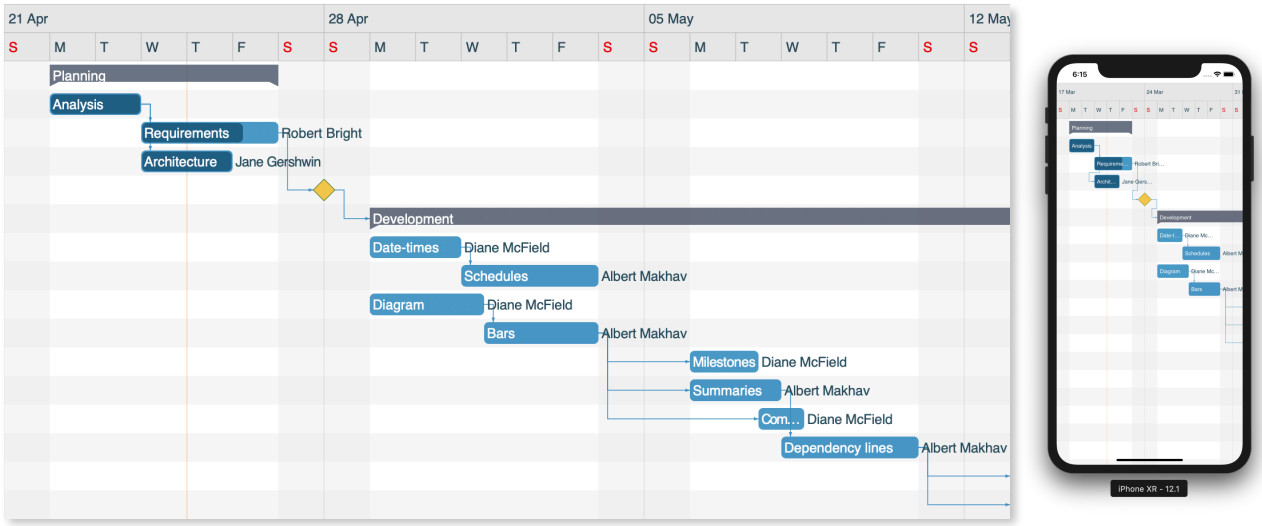
for *Ganttis* (macOS) and *GanttisTouch* (iOS) 2.0.3 — DlhSoft, 2020

Overview

Ganttis framework 2 — designed and developed by DlhSoft using Swift 5.1 — allows you to easily add outstanding interactive Gantt charts to your macOS 10.13+ and iOS 11.3+ apps built with Xcode 11, displaying few to virtually trillions of scrollable items with customizable timeline headers, schedule definitions, appearance, and behavior, optionally synchronized with associated outline views (macOS).

While Gantt charts were originally used mostly to present project plans and associated resource assignments, they can actually be used in multiple other time-enabled contexts, since any data type that defines one or more date-time fields can actually be mapped and shown as bars (or as other kind of shapes) with optional labels, styling, and dependencies between them, on a classic horizontal timeline.

Although it may be fully reconfigured by defining a custom schedule (specifying working and nonworking times), by updating behavioral options, and by setting up overall or item-level appearance styles (to name just a few of its features), a typical Gantt chart component initialized using Ganttis framework and a project-like data hierarchy would look like this:



Ganttis framework is free to download and try for unlimited time, without any feature limitations (a runtime warning screen would periodically let you know, however, that purchasing and applying a product license is required for production purposes.)

Demo source code as well as simplified sample apps (for both macOS and iOS) are also publicly available on GitHub.

Moreover, free and unlimited technical support (provided directly by developers!) is available regardless of whether you have yet ordered a product license or not.

Setup

Ganttis framework includes both macOS and iOS components. While most of their features are similar and can be configured the same way on each of the supported platforms, initial setup steps will partially depend on the actual target platform of your app.

Getting Ganttis package

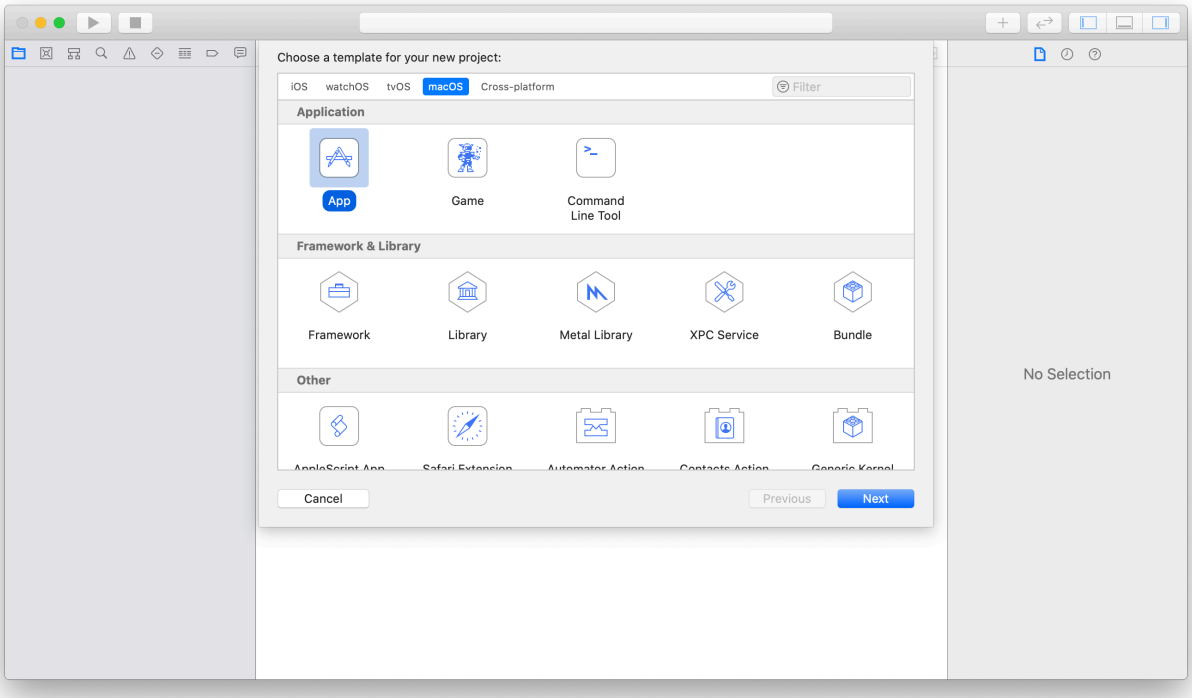
To prepare for using the Gantt chart components in your apps, you need to download Ganttis package from DlhSoft Web site and extract it to a convenable location on your Mac.

Extracted content includes all the necessary binary files, packaged as an *XCFramework* bundle to be referenced from your projects at development time. The correct binaries will be linked by Xcode at build time, depending on the actual platforms that your targets specify.

The *license agreement* associated with Ganttis software is also included in the downloaded archive. You should only use the product if you agree with all its terms and conditions.

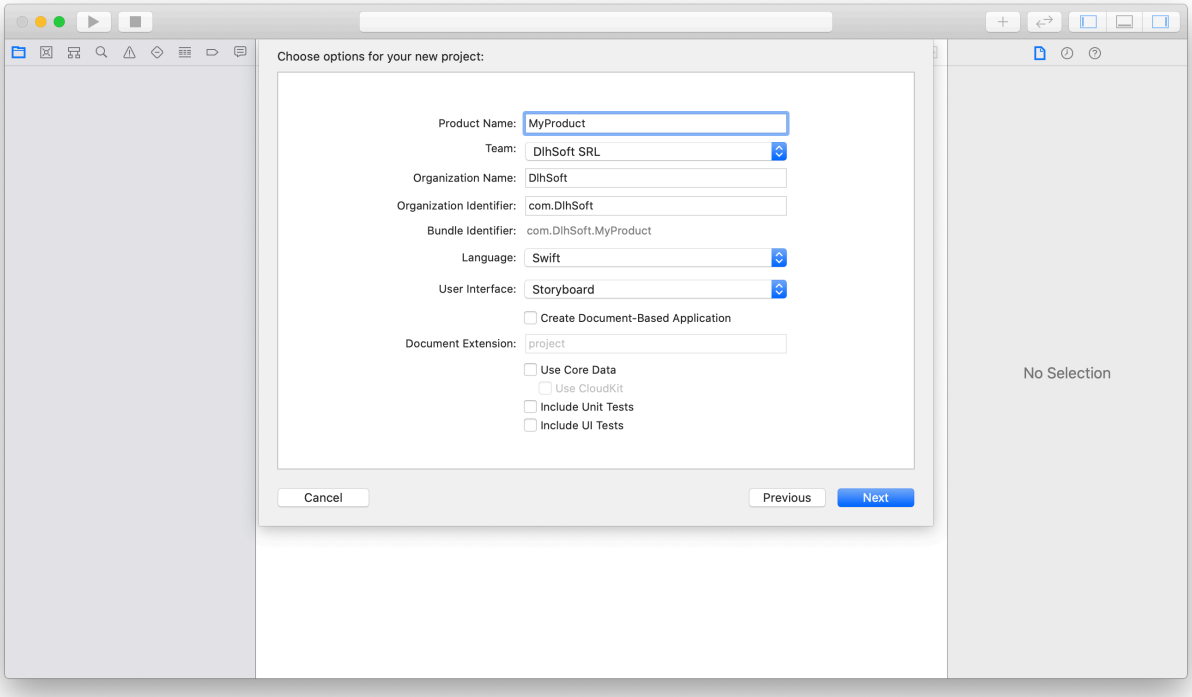
Initializing Xcode project

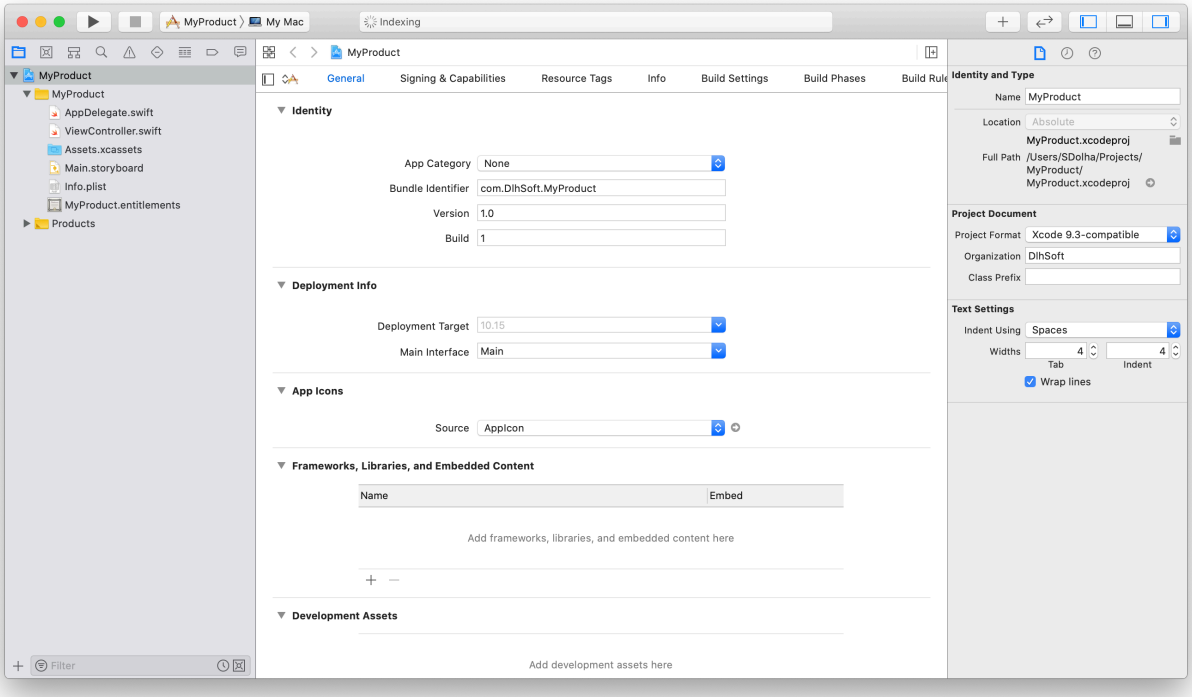
If you already have a macOS or iOS app project, open it in Xcode, using *Open* command from the *File* menu. Otherwise, you can create one by using *New Project* command from Xcode's *File* menu. For macOS, select *Cocoa App* template from *Application* section; for iOS, select *Single View App* template (or another template that may better fit the requirements):



To complete the setup, define the project options that apply and select *Swift* programming language in the following dialogs.

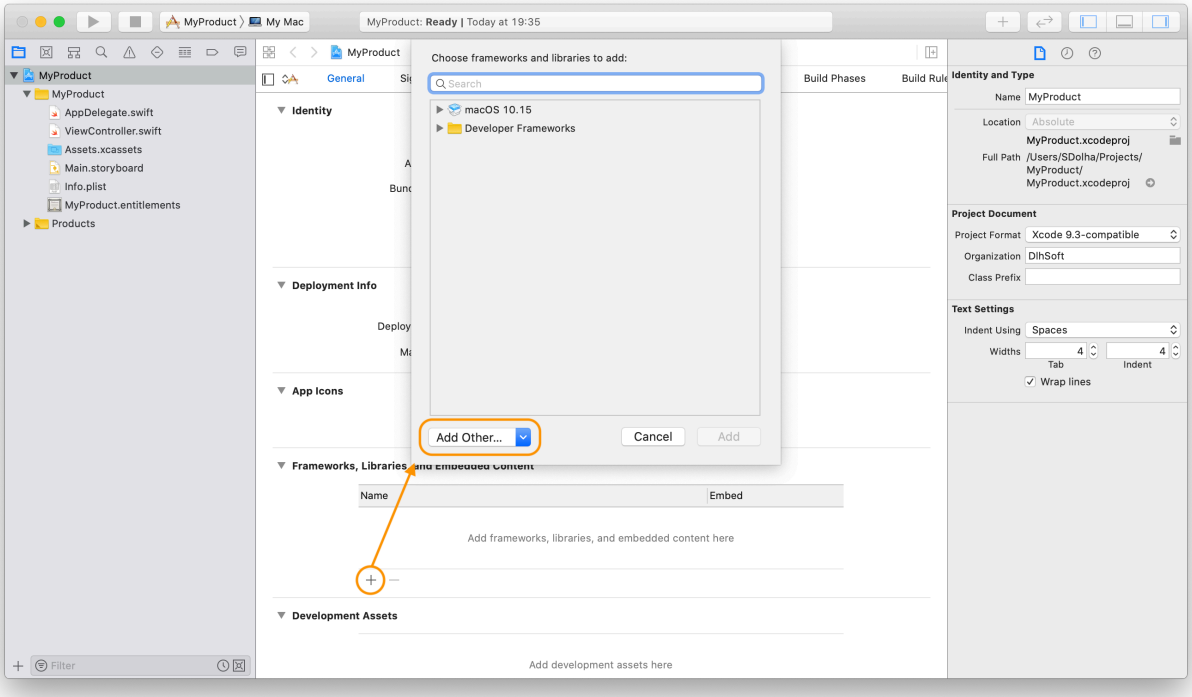
You can also select *Objective C* instead, but you will then need to refer to the last section of this chapter for follow up information on how to perform calls to the Ganttis APIs from your app (Swift integration code is required.)

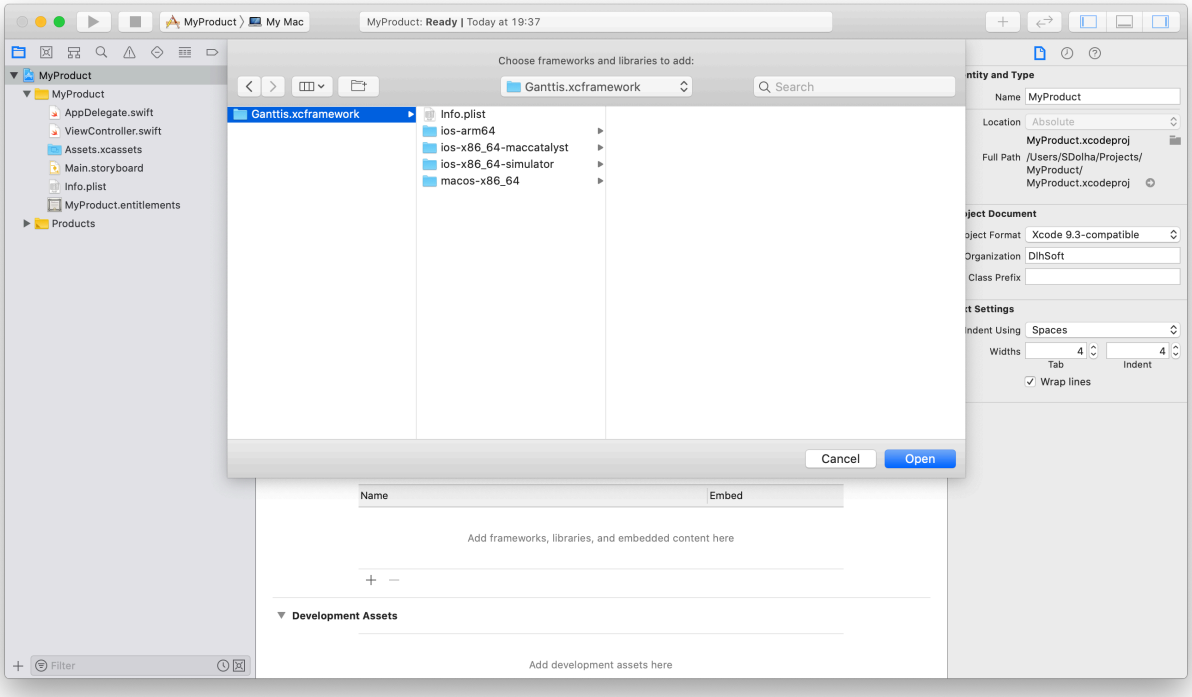


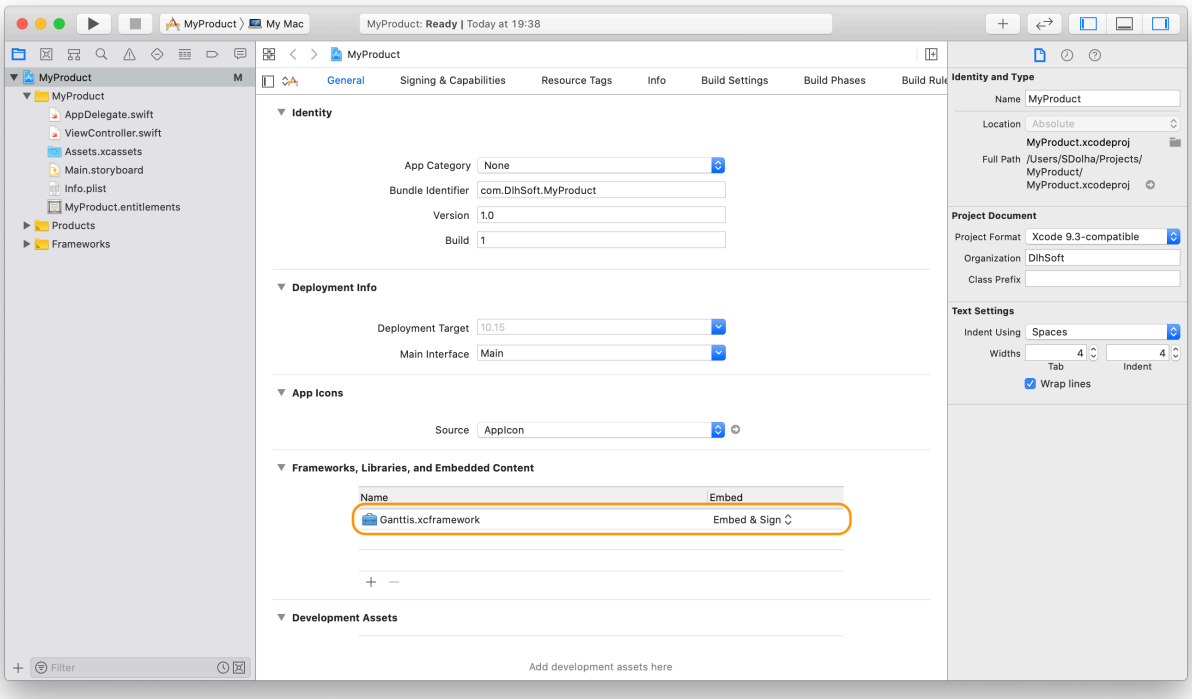


Adding Ganttis framework

Add *Ganttis* framework (that you have extracted from the archive downloaded from Dlh-Soft Web site) to the project and ensure it is set to be *embedded and signed* within each of the targets that you need it to be accessible from:

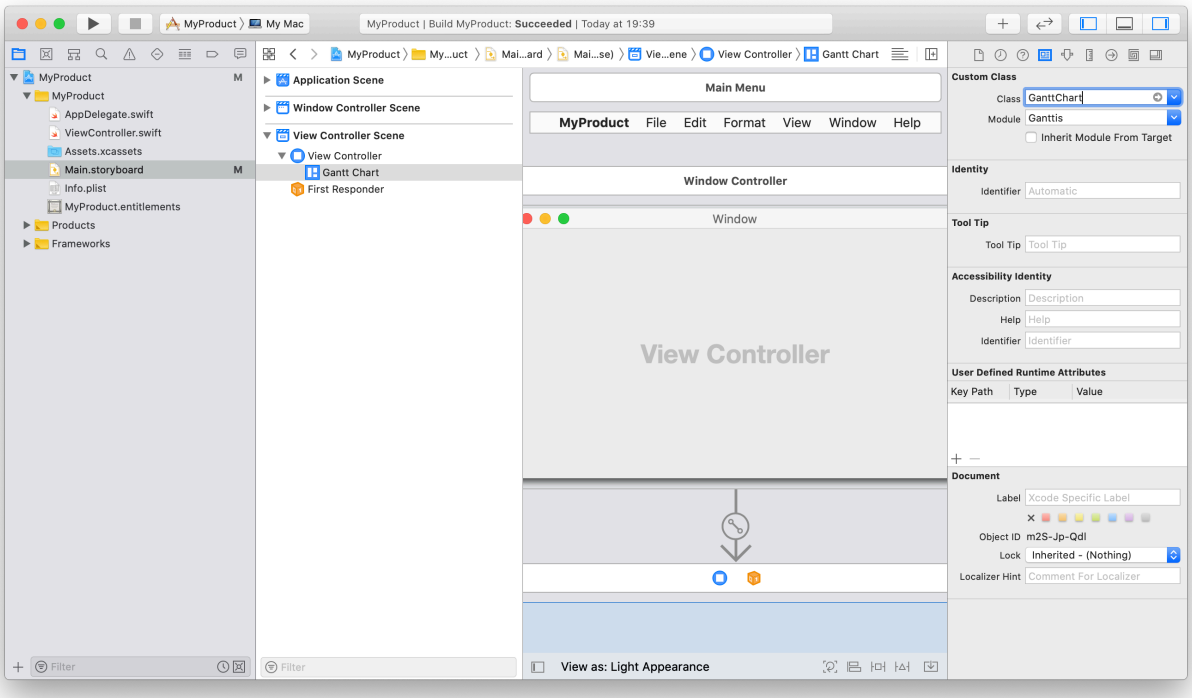






Using Ganttis components in storyboards

You can use *GanttChart* component, for example, as a *View* within a storyboard. To set it up, enter *GanttChart* as *Custom Class* under *Identity Inspector* when the view is selected, and *Ganttis* (or *GanttisTouch*, for iOS) as *Module*:



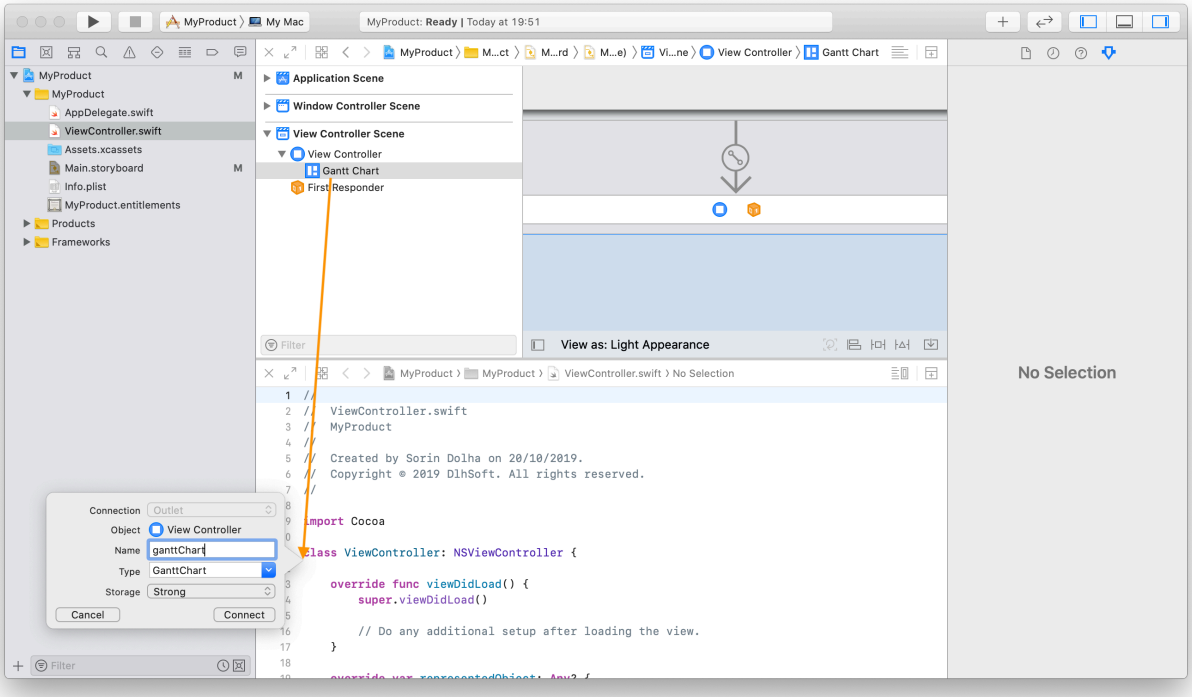
Another helpful type is *OutlineGanttChart*, representing a chart with outline view (macOS).

Importing and using Ganttis framework in code

To to be able use Ganttis components in code, such as in your *ViewController*, simply import the appropriate module (*Ganttis* or *GanttisTouch*):

macOS	iOS
`import Ganttis`	`import GanttisTouch`

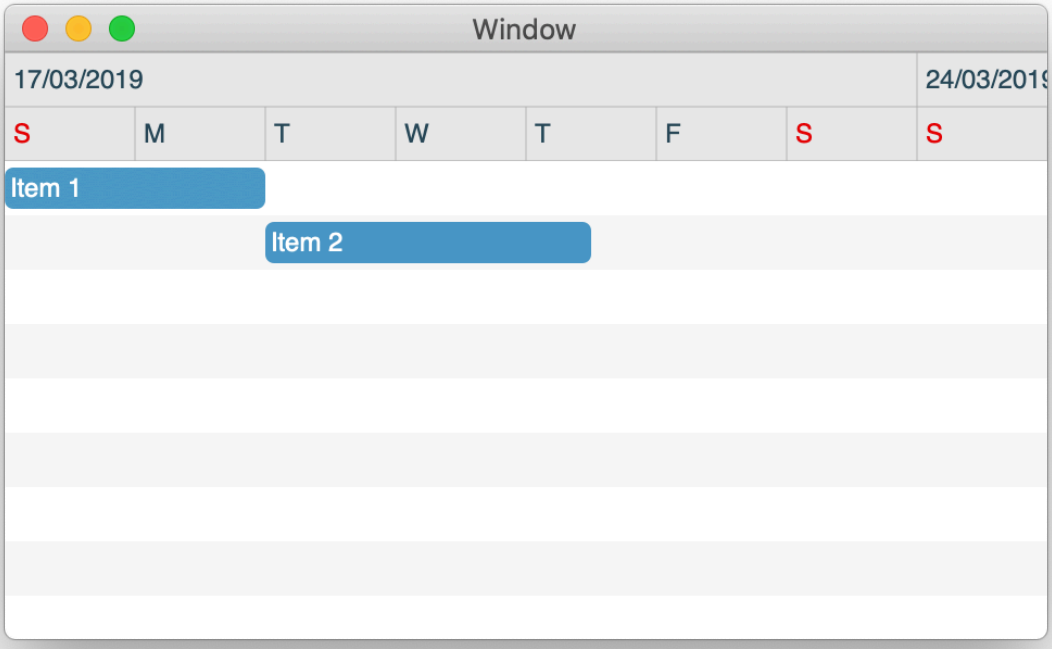
You can then create an Interface Builder outlet from the storyboard's *GanttChart* view as usual — *Option*-click the code file while the storyboard is selected to open it in the *Assistant Editor*, *Control*-drag the view to the location in code where the new outlet should be created, enter a name for the associated variable (e.g. *ganttChart*) and click *Connect*:



To set up the *GanttChart* component, write code in the appropriate *NSViewController* (or *UIViewController*) class to define a *GanttChartController* instance, which would in turn need header and content controllers. The underlying controllers layer provides all services that the user interface components need. The content controller can be initialized using a simple item array as data source (or using a highly customizable *GanttChartItemManager* instead):

```
class ViewController: NSViewController {
    @IBOutlet var ganttChart: GanttChart!
    override func viewDidLoad() {
        super.viewDidLoad()
        let item1 = GanttChartItem(
            label: "Item 1", row: 0,
            start: Time(year: 2019, month: 03, day: 17, hour: 0),
            finish: Time(year: 2019, month: 03, day: 18, hour: 24))
        …
        let items = [item1, …]
```

```
        let headerController = GanttChartHeaderController()
        let contentController = GanttChartContentController(items: items)
        contentController.desiredScrollableRowCount = 10
        contentController.scrollableTimeline = TimeRange(
            from: item1.start, to: item2.finish.adding(weeks: 2))
        let controller = GanttChartController(
            headerController: headerController,
            contentController: contentController)
        ganttChart.controller = controller
    }
}
```



Objective C considerations

Ganttis framework has been developed in pure Swift, and therefore to access its API from classic Objective C code you will need to define a Cocoa bridging class in Swift. This class may be a *ViewController* with an associated *.xib* definition that reuses *GanttChart* inside:

```
class GanttChartViewController: NSViewController {
    @IBOutlet var ganttChart: GanttChart!
    override func viewDidLoad() {
        super.viewDidLoad()
        ganttChart.controller = …
    }
}
```

SwiftUI integration

You can create wrappers for Ganttis components and use them in your SwiftUI apps, as presented in this tutorial from Apple – for specific details see our macOS and iOS sample apps.

Items and dependencies

GanttChartContentController class that governs the main chart area of a *GanttChart* component uses a *GanttChartItemManager* instance to handle data items represented as bars in the view and item dependencies shown using segmented arrow lines between the item bars.

The controller provides all user interface services needed by the component exposed to the end user and allows the developer to configure its settings and behavior as needed.

Note that for a main *GanttChart* component the *GanttChartContentController* would be part of a *GanttChartController* container object that links it to a *GanttChartHeaderController*:

```
let headerController = GanttChartHeaderController()
let contentController = GanttChartContentController(itemManager: itemManager)
let controller = GanttChartController(headerController: headerController,
                                      contentController: contentController)
ganttChart.controller = controller
```

Item management

You can create a *GanttChartContentController* instance in a convenient way by passing a collection of items (and optionally one of dependencies as well) instead of a manager instance. In this case, a *GanttChartItemSource* object that inherits from *GanttChartItemManager* is created under the hood, providing all the necessary management features:

```
let items = […]
let dependencies = […]
let contentController = GanttChartContentController(items: items,
                                                    dependencies: dependencies)
```

You may use a custom manager object if you want to achieve better performance or if you need specialized item management services, though. Such a custom manager object can be defined in two ways: as an instance of custom class that inherits from *GanttChartItemManager* and overrides its functions, or (recommended) as a direct instance initialized with an object that conforms to the *GanttChartCollectionProvider* protocol:

```
class ViewController: NSViewController, GanttChartCollectionProvider {
    override func viewDidLoad() {
        let itemManager = GanttChartItemManager(collectionProvider: self)
        let contentController = GanttChartContentController(
            itemManager: itemManager)
        …
    }
    var totalRowCount: Int { return … }
    var preferredTimeline: TimeRange { return TimeRange(from: …, to: …) }
    func filteredItems(range: RowRange,
                       timeline: TimeRange) -> [GanttChartItem] {
        return […]
    }
    func filteredDependencies(range: RowRange,
                              timeline: TimeRange) -> [GanttChartDependency] {
        return […]
    }
}
```

Item fields

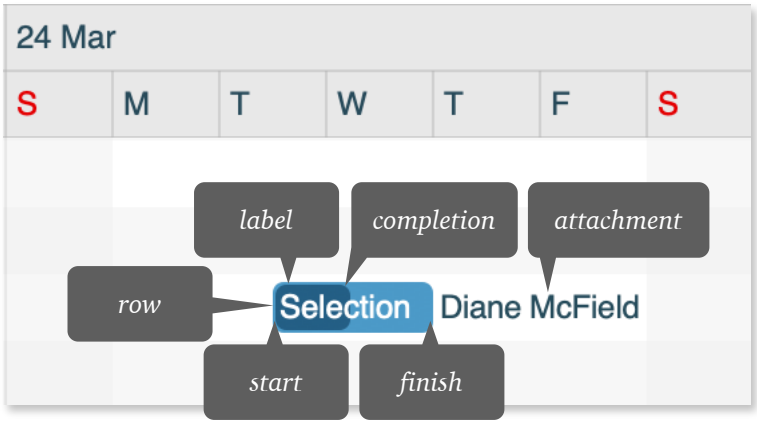
Regardless of the manager type you use, individual *GanttChartItem* objects may be defined using these main fields:

label	Text to be displayed on the bar in chart.
row	Determines the vertical position of the bar; multiple items may share the same row value, for example when defining a schedule chart that would display all tasks assigned to an individual resource in a single row.
start, *finish* (*time*)	Values defining the interval that the item spans, available as well as an aggregating range field (*time*).
completion	The rate of completion for the item, between 0 and 1.
attachment	Text to be displayed to the right of the bar in chart.
details	Text to be displayed as tooltip when the end user would hover the bar (macOS only).
type	Defines how to represent the item in chart: as *standard* bar (default), *milestone* diamond, or *summary* polygon.

All fields are optional, except for *row* and *time*, so items can easily be created using shorter initializer forms:

```
var item = GanttChartItem(label: "A", row: 0,
                          start: date1, finish: date2)
```

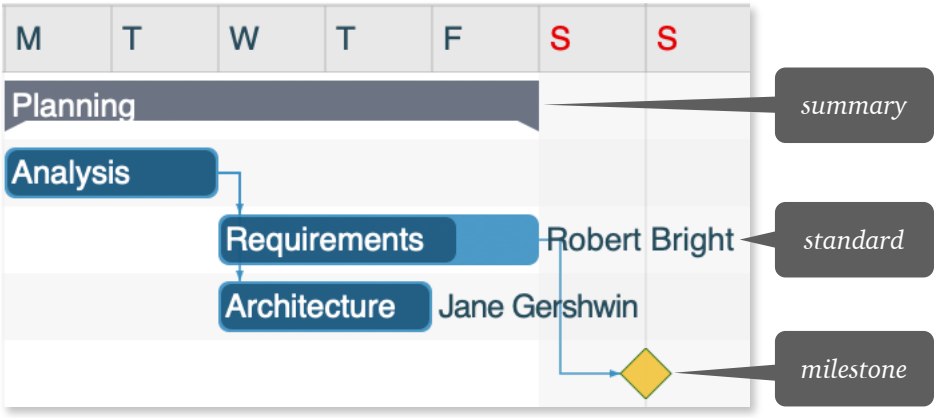
The item fields and their correspondence to the elements of associated bar presented in the user interface are shown below:



Note that the *details* of the item will be displayed as a tooltip upon hovering the associated bar.

Also note that the completion bar is bound to a timeline position that depends on the actual working time that the item spans, and therefore its screen width is not always proportional to the item's *completion* rate.

The available item types and their shape correspondence in the user interface are also indicated in the screenshot below:



Dependency fields

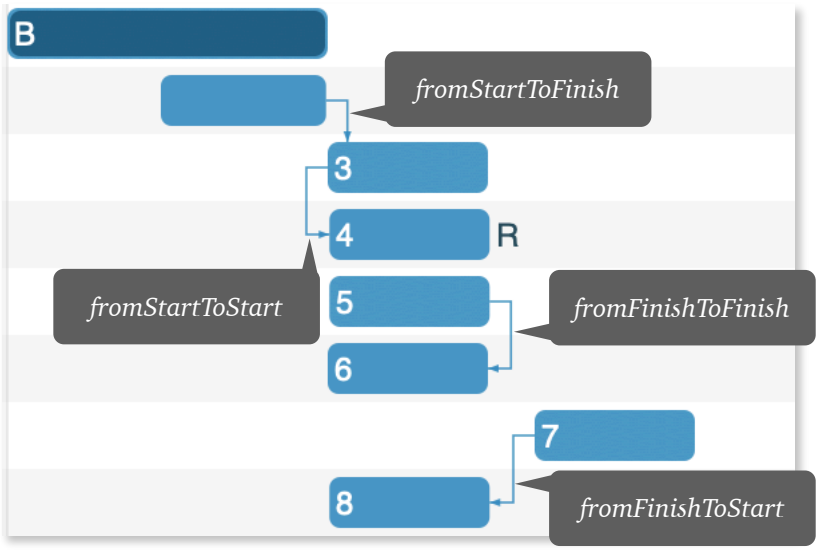
You can define item dependencies as *GanttChartDependency* objects using these fields:

from, *to*	The items that the dependency is defined between.
details	Text to be displayed as tooltip when the end user would hover the arrow line (macOS only).
type	Defines the dependency type: *fromFinishToStart* (default), *fromStartToStart*, *fromFinishToFinish*, or *fromStartToFinish*.

All fields are optional, except for *from* and *to*:

```
var dependency = GanttChartDependency(from: item1, to: item2,
                                      type: .fromStartToStart)
```

The available dependency types are presented in the screenshot below:



Filtered objects

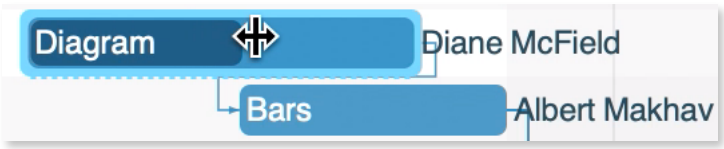
For optimization purposes *GanttChartItemManager* object allows its owner (e.g. the controller of the user interface component) to set *range* and *timeline* properties for specifying the rows and interval that are to be considered active (i.e. scrolled to) at a specific moment, and would allow retrieving the items and dependencies that are matching those filters through *filteredItems* and *filteredDependencies* properties.

You can, of course, also use these arrays to browse the (fully or partially) visible items and dependencies on screen.

Handling changes

If the developer doesn't disable the feature, end users may change items and dependencies directly in the user interface of the Gantt chart.

For example, users can drag a bar horizontally to update its item's *start* and *finish* fields at the same time, resize the bar to update *start* or *finish* and increase or decrease the duration of the item, resize a completion bar to update the *completion* rate for the item, drag a bar vertically from its bottom to change its associated *row* value, draw new dependency lines between items by performing drag and drop operations (starting from a thumb that would appear when hovering one of the *from* item's bar ends, and finishing on the start or finish area of the *to* item's bar), delete items and dependencies using context menus, and so on:



For your code to react to data field changes triggered in the application by the end user you may implement *GanttChartItemObserver* protocol and bind the implementation to the item manager instance of the controller:

```
class ViewController: NSViewController, GanttChartItemObserver {
    override func viewDidLoad() {
        let contentController = GanttChartContentController(…)
        contentController.itemManager.itemObserver = self
        …
    }
    func itemWasAdded(_ item: GanttChartItem) { … }
    func timeDidChange(for item: GanttChartItem,
                       from originalValue: TimeRange) { … }
    func completionDidChange(for item: GanttChartItem,
                             from originalValue: Double) { … }
    func rowDidChange(for item: GanttChartItem,
                      from originalValue: Row) { … }
    func itemWasRemoved(_ item: GanttChartItem) { … }
    func dependencyWasAdded(_ dependency: GanttChartDependency) { … }
    func dependencyWasRemoved(_ dependency: GanttChartDependency) { … }
}
```

To propagate item and dependency changes to appropriate external source objects (such as to persist updates in specific tables of the source database), you can use this approach:

- set the *context* field (of type *Any*) of each item and dependency to the key field value of the source data item (such as the data table's identifier column) or to the reference of an external object that encapsulates it when using data entity abstraction layers;
- use this *context* value from the item or dependency received in the notification (i.e. available as method arguments in the observer protocol's implementation) to find the appropriate data source rows or objects and update them according to the identified changes.

Newly created objects

In case you (also) need to customize the way items and dependencies are set up when they are created by the end user, you can implement *GanttChartItemFactory* protocol (as well):

```
class ViewController: NSViewController, GanttChartItemFactory {
    override func viewDidLoad() {
        let contentController = GanttChartContentController(…)
        contentController.itemManager.itemFactory = self
        …
    }
    func createItem(row: Row, time: Time, isMilestone: Bool) -> GanttChartItem {
        return GanttChartItem(…)
    }
    func createDependency(from: GanttChartItem, to: GanttChartItem,
                          type: GanttChartDependencyType)
                                                    -> GanttChartDependency {
        return GanttChartDependency(…)
    }
}
```

Collections

Item and dependency collections may be updated and queried for computed values after initialization too, through *GanttChartItemManager* object's interface, as follows:

addNewItem	Creates and adds a new item on the specified *row* and at the specified *time*, optionally being of type *milestone*.
removeItem	Removes an existing item.
addNewDependency	Creates and adds a new dependency between the specified *from* and *to* items, using the optionally specified *type*.
removeDependency	Removes and existing dependency.
scheduledDuration	Computes the total scheduled duration for an item (from *start* to *finish*), optionally in a specific time *unit*.
completedDuration	Computes the completed duration for an item, optionally in a specific time *unit*, considering the total scheduled duration and the *completion* rate of the item.

timeCompletedUntil	Computes the date and time up to which the completed duration goes since the item's *start*.
updateTime	Updates an item to *start* on a specific value.
updateDuration	Updates and item's *start* or *finish* to ensure a specific scheduled duration.
updateCompletion	Updates an item's *completion* rate to ensure a specific completed duration.
updateRow	Updates an item's *row*, moving an item vertically.
schedule	Determines the schedule definition to apply to an item, being either the schedule at the item level if it was defined, or the schedule of the manager object otherwise.
applySchedule	Updates the schedule for all items in the managed collection, or for a single specified item applying the schedule definition changes. This is useful if you modify schedule definitions directly, or if they use logic that would change if re-run (such as to determine excluded intervals), and you want the collection refreshed.
collectionDidChange	Call this method for observers (such as the user interface) to be notified one or more properties may have been changed for the managed items without using item manager's interface. This is useful if you modify item properties directly, and you want the user interface to be refreshed accordingly.

Data source

If you would like to develop your own item collection with a custom data source you may inherit from *GanttChartItemManager* and (optionally) override these main properties and methods:

• *sourceTotalRowCount*, *sourcePreferredTimeline;*

• *sourceFilteredItems*, *sourceFilteredDependencies;*

• *addNewSourceItem*, *removeSourceItem*, *createNewSourceItem;*

• *addNewSourceDependency*, *removeSourceDependency*, *createNewSourceDependency*.

The *source-* properties should return the values to be used as total row count, preferred timeline (scrollable interval) and the filtered items and dependencies given the current *range* and *timeline* (properties of *GanttChartItemManager* base class.)

createSource- methods (if overridden) should return new objects of appropriate type (initializing directly, or inheriting from *GanttChartItem* and *GanttChartDependency* classes), allocating their associated data objects or identifiers, as needed; if they are not overridden the base class' methods would simply call the factory instead.

addNewSource- methods should internally call the built-in or custom *createSource-* methods and add the appropriate associated objects to the external data source. They are called before the objects are added to the managed context.

removeNewSource- methods are called just before objects are removed from the managed context, to allow you to remove the associated data objects from the external source.

As already indicated, *GanttChartItemSource* class inherits from *GanttChartItemManager* too, defining its own data source as a tuple of item and dependency arrays (*items* and *dependencies* properties, passed at initialization time.)

By design, the total row count and preferred timeline of a *GanttChartItemSource* are computed based on *row* and *time* values of the items in the managed context. Filtered items are matching range and timeline, as expected, and filtered dependencies are those that span from or to any of these filtered items.

Time values

Gantt chart items and multiple other framework objects use the *Time* structure to record dates and times, rather than foundation's *Date*, while full casting capabilities to and from *Date* values are, of course, provided. *Time* values are internally defined using these members in order to have the data optimized for the user interface at all times:

week	The number of weeks passed since reference week of 1/1/2001 (technically the first day of week 0 is Sunday, 12/31/2000).
dayOfWeek	The day of week (*Sunday*=0 to *Saturday*=6).
timeOfDay	Time of day in seconds passed since midnight, up to the next midnight (inclusive).

For convenience, however, *Time* values can also be initialized by passing:

dayNumber or *year*, *month*, *day*	The number of days passed since reference date of 1/1/2001 or its Gregorian calendar components.
hours, *minutes*, *seconds*	Time of day components, from 00:00:00 to 24:00:00 (inclusive).

Note that to properly support interval finish times, 24:00:00 value is considered as a valid time of day. By using end-of-day time values you don't need to check for special midnight cases anymore — the date would still represent the actual ending day, and not the following one. (While, of course, comparing *Time* values continues to work as expected: the end of day midnight equals the start of the following day.)

You can compute a Time value shifting the current value with a specified duration by calling its *adding* method.

Intervals

The *TimeRange* structure allows defining a time interval as well, between *start* and *finish* *Time* values (e.g. the *time* property of *GanttChartItem* instances.)

A time range value offers support to easily determine interval *duration* in seconds or another (specified) time unit, to shift the interval at both ends by a specific duration, to check whether the interval contains a specific *Time* value or not, and to determine its intersection with another *TimeRange*.

A time range can be *entropic*, meaning that *finish* time is after *start*. When *finish* equals *start*, it also becomes *momentary*. (Note: momentary ranges are still considered entropic.)

Diagram sizing settings

To configure the size of the item bars and attachment labels in the chart you may set *rowHeight*, *hourWidth*, and *attachmentLabelWidth* properties of *GanttChartContentController*.

When both content and headers are used (such as with an aggregated *GanttChart* component instance), *hourWidth* is controlled using the (otherwise synchronized) *GanttChartHeaderController*'s property, though:

```
contentController.rowHeight = 28
headerController.hourWidth = 2.8
contentController.attachmentLabelWidth = 100
```

To configure the scrollable timeline of the diagram if you don't want it to be automatically updated based on items' *start* and *finish* values, set content controller's *scrollableTimeline* property to the required time range:

```
let now = Time(year: 2019, month: 03, day: 17, hour: 0)
let later = now.adding(weeks: 5)
contentController.scrollableTimeline = TimeRange(from: now, to: later)
```

And finally, if you want to ensure that the diagram defines vertical space for a minimal row count instead of it being based on items' *row* values alone, set the *desiredScrollableRowCount* property as well:

```
contentController.desiredScrollableRowCount = 20
```

Note that on iOS there are inherent limits for the total size of the chart area, due to the performance-oriented design of the *UIView* instances that Gantt chart components use. Therefore you should carefully set up (and throughly test) the selected *hourWidth* and *rowHeight* values in conjunction with the *scrollableTimeline* and maximum row count, controlled by *desiredScrollableRowCount* and *itemManager.totalRowCount* + *extraRowCount*.

By default *extraRowCount* value of the content controller is set to 1, to show one supplemental empty row to the bottom of the diagram, allowing end users to increase the total count by dragging items vertically to that row.

Zoom level

Setting content controller's *hourWidth* property allows the developer to control the default widths of the bars in the diagram. However, the *actualHourWidth* computed at runtime is also considering an arbitrary zoom level property, *zoom*, which is by default 1 (100%).

The developer is allowed to set the initial zoom level of the diagram by just updating this value at component instance setup time, and then the end user is also able to increase or decrease the zoom level by performing a drag operation on the chart headers area (macOS and iOS) and/or by a pinch gesture on the main chart area itself (iOS):

```
contentController.zoom = 1.5
```

Scroll and zoom changes

Scroll and zoom changes are notified through *GanttChartContentViewportObserver* protocol.

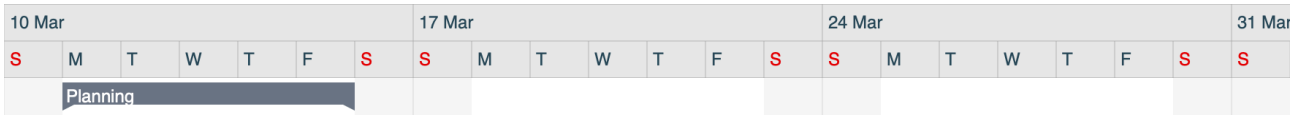
Schedule and headers

Once the items and dependencies are set up in the Gantt chart, it's time to also configure their working and nonworking time, optionally synchronized with the headers that are displayed at the top of the diagram, and with in-chart vertical separators and highlights for different time intervals as needed in your application.

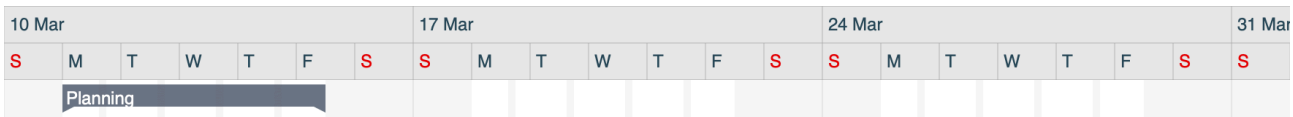
For example, items may be configured to be managed within the range of a standard Monday to Friday week, and between 8 AM and 4 PM each day, excepting holidays. Or you may want them to use the interval of 8 AM to 5 PM each day with 1 hour break between 12 PM and 1 PM. Or set up that Fridays are shorter days, ending at 3 PM.

Other times, however, the continuous schedule (Sunday-Saturday, 24 hours/day) might fit better. Or a full Sunday-Saturday week, but with only the standard 8 hours each day. Or a standard Monday-Friday week, but with 3 shifts per day. (Or any combination of these.)

Moreover, in the user interface you may want to display more (or less) time than the working intervals. For example, even when you have set up a standard week as working time, you might still want to display weekends and holidays — yet probably slightly highlighted — to allow the end user easily understand the real duration of the items in the chart:



Standard week schedule

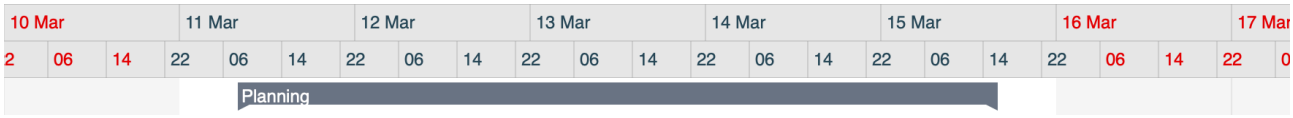


Highlighting nonworking time

And of course, besides the schedule based highlighting the developer may want to specifically highlight one or more sets of time intervals with different colors, to identify certain periods, as necessary in your specific application.

Of course, the diagram headers are used to allow the end user to identify the actual dates and times that the visible items share. You might want to display only one header (such as for individual days), two (e.g. weeks and days, months and weeks, or days and hours, optionally with periods, such as to show shifts), a custom number of headers with custom intervals, or even a custom range updating based on the zoom level of the diagram:

Month-week-day headers



Day-shift headers

Schedule definitions

To specify the working and nonworking time for items to use (or to indicate which time intervals should be visible and/or which to be highlighted and which not), Ganttis module offers two types: *ScheduleDefinition* and *Schedule*.

ScheduleDefinition can be used as base class for custom schedules, overriding its members:

weekInterval	Defines the working days of week (e.g. Monday to Friday.)
dayInterval	Defines the working time of day (e.g. 8 AM to 4 PM.)
excludedIntervals	Function that defines nonworking time intervals (e.g. holidays or breaks) given an input time range.

A *hasExcludedIntervals* flag also provides a general way to optimize Ganttis algorithms when no nonworking time intervals would be ever returned by *excludedIntervals* function.

Alternatively, you can initialize a *ScheduleDefinition* instance by passing a *ScheduleDefinitionProvider* object (that would also be an *ExcludedTimeIntervalProvider*.) *ScheduleDefinitionProvider* protocol defines the same members except for *hasExcludedIntervals* flag.

Schedule objects

Schedule class inherits from *ScheduleDefinition* and hosts excluded intervals as an array rather than defining them through a function. It can optionally be enriched using a custom *ExcludedTimeIntervalProvider* object, though (and that can be a functionally customizable

ExcludedTimeIntervalSource). It's therefore the preferred way to define working and non-working time when you use Ganttis components:

```
let schedule = Schedule(
    weekInterval: WeekRange(from: .monday, to: .thursday),
    dayInterval: DayRange(
        from: TimeInterval(from: 8, in: .hours),
        to: TimeInterval(from: 16, in: .hours)),
    excludedIntervals: [
        TimeRange(from: time11, to: time12),
        TimeRange(from: time21, to: time22)],
    excludedIntervalProvider: ExcludedTimeIntervalSource { time, limit in
        return [TimeRange(…), …] })
```

The excluded interval source function would receive a *Time* value to determine the excluded intervals around (as an array of *TimeRange* values), and the *limit* up (or down) to which those intervals are needed.

The limit is also a *Time* value and it indicates the direction in which the intervals are needed (when it's not the same as the time value, in which case you would just need to return the intervals that contain that moment) and it's very useful to avoid infinite (or too long) search loops that may otherwise try to identify the next (but far away) excluded intervals starting from that specific time value.

Note that usually the excluded interval source's function would just need to return the largest interval near the input time towards the limit, but the API supports returning an entire array as well to allow your logic to be simplified, whenever possible (and avoid the actual sorting and single interval selection, which can also be made by Ganttis module itself.)

Some static (built-in) schedule instances are provided on *Schedule* class:

continuous	Sunday to Saturday, midnight to next midnight (24/7).
standard	Monday to Friday, 8 AM to 4 PM.
fullWeek	Sunday to Saturday, 8 AM to 4 PM.
fullDay	Monday to Friday, midnight to next midnight.

As previously mentioned, schedule definition instances are used in multiple settings areas of Ganttis components. The most important ones are listed below:

contentController.visibilitySchedule	Sets the visible timeline of the chart (excluded intervals are hidden).
itemManager.schedule	Sets the default working (and nonworking) time of the items displayed in the diagram.
item.schedule	Sets specific rules for the working (and nonworking) time for a specific item displayed in the diagram (overriding *itemManager.schedule*).

Header rows

To set up the Gantt Chart header rows you will need to initialize the *rows* collection of *GanttChartHeaderController* using *GanttChartHeaderRow* instances:

```
headerController.rows = [
    GanttChartHeaderRow(.weeks, format: "dd.MM"),
    GanttChartHeaderRow(.days, format: .dayOfWeekAbbreviation)]
```

Technically, the *GanttChartHeaderRow* initializer supports either one instance or an array of *TimeSelector* objects as input argument:

```
let headerRow = GanttChartHeaderRow(selector)
let multiSelectorHeaderRow = GanttChartHeaderRow([selector1, selector2])
```

However, in practice, it is usually enough to associate one time selector to each header row.

(You would associate multiple selectors to a single row only if you'd want multiple types of intervals shown in the same header, but even then, only one of them should have labeled values displayed.)

The *TimeSelector* object initializer receives two arguments: a *TimeIntervalSelector* used to define which would be the header intervals within the visible time range, and a *TimeLabelGenerator* that prepares the text to be displayed as labels for the visible header intervals:

```
let selector = TimeSelector(intervalSelector: intervalSelector,
                            labelGenerator: labelGenerator)
```

The interval selector and label generator objects may be custom instances, but a *TimeSelector* extension offers multiple convenience initializers as well, allowing you to set up the actual arguments providing a *TimeIntervalType* enumeration value and optionally associated values such as label *format* and *locale*, and an *intervals* Boolean flag that would indicate whether the values for both interval ends (start and finish) should be used (optionally separated by a custom *separator* string overriding the dash based default.) The *weeks* and *days* header rows exemplified in the beginning of this section use *TimeSelector* shortcut initializers.

Note that *format* (and all associated arguments) may also be omitted and in this case the actual text format of the label values is determined by the selected interval type.

Interval selectors

You can set up an interval selector for a header row using an object that implements *TimeIntervalSelector* protocol, such as *PeriodSelector*, *CalendarPeriodSelector*, or a functionally customizable *TimeIntervalSource*.

PeriodSelector objects allow you to define interval selectors based on weeks, days, half-days, hours, minutes, seconds, or submultiples of second. They can be periodically repetitive (e.g. repeat every 3 days, or every 90 minutes) and they may start with a time offset (phase) if needed (e.g. every day starts at 6 AM, or every hour starts at minute 30):

```
let intervalSelector = PeriodSelector(
    period: 1.5, in: .hours,
    schedule: .standard, origin: projectStart,
    offset: 30, offsetIn: .minutes)
```

The initialization above would create an interval selector that returns periods of 1.5 hours, considering only the standard schedule (Monday to Friday, 8 AM to 4 PM), starting at *projectStart* time and with an offset of 30 minutes.

CalendarPeriodSelector objects allow you to define interval selectors based on months, quarters, years, or multiples of year. They can be periodically repetitive (e.g. repeat every 3 months, or every 2 years) and they may start with a time and/or calendar offset (phase) if/as needed (e.g. every month starts on day 10, or every year starts at hours 12:00 on the first day):

```
let intervalSelector = CalendarPeriodSelector(
    period: 2, in: .quarters,
    origin: projectStart,
    calendarOffset: 1, calendarOffsetUnit: .months,
    offset: 10, offsetIn: .days)
```

The initialization above would create an interval selector that returns periods of 2 quarters, starting at *projectStart* time and with an (aggregated) offset of 1 month and 10 days.

Special time interval selector objects can be created using *MomentPeriodSelector* class which allows defining time intervals for or between specific time values:

```
let intervalSelector = MomentPeriodSelector(
    for: [time1, time2], separating: true)
```

The initialization above would create an interval selector that generates intervals from the past to *time1*, between *time1* and *time2*, and from *time2* to the future.

Also, *EnclosedTimeIntervalSelector* may be used to return the same intervals as the interval selector object passed to it upon initialization, except that the intervals at the start and finish of the range for which the intervals are needed (e.g. the visible timeline) are trimmed to the limits of the range itself. This is very useful to generate intervals that update dynamically on screen when the end user scrolls the diagram horizontally, showing the visible start and/or finish time displayed in the current viewport:

```
let intervalSelector = EnclosedTimeIntervalSelector(
    PeriodSelector(
        period: 1.5, in: .hours,
        schedule: .standard, origin: projectStart,
        offset: 30, offsetIn: .minutes))
```

The initialization above would create an interval selector that is similar to the *PeriodSelector* defined as the first example, but the output intervals would be limited, at runtime, to the visible timeline of the *GanttChart* view.

For convenience, appropriate interval selectors are available when calling a shortcut *TimeSelector* initializer with a *TimeIntervalType* value:

days(*period*, *startingAtHours*)	Selects days with optionally defined period and offset in hours.
weeks(*period*, *startingOn*)	Selects weeks with optionally defined period and offset in days (given by start day of week.)
monthly(*period*, *startingOn*)	Selects months with optionally defined period and offset in days (given by start day of month.)
quarters(*period*, *startingOnMonthOfQuarter*)	Selects quarters with optionally defined period and offset in months (given by start month of quarter.)
years(*period*, *startingOnMonth*)	Selects years with optionally defined period and offset in months.
decades(*period*, *startingOnYearOfDecade*), *centuries*, *millennia*	Selects decades (or centuries, millennia) with optionally defined period and offset in years.
halfdays(*period*, *startingAtHour*)	Selects half-days with optionally defined period and offset in hours (given by start hour of half-day.)
hours(*period*, *startingAtMinute*)	Selects hours with optionally defined period and offset in minutes.
minutes(*period*, *startingAtSecond*)	Selects minutes with optionally defined period and offset in seconds.
seconds(*period*, *startingAtMillisecond*), *deciseconds*, *centiseconds*	Selects seconds (or deciseconds, centiseconds) with optionally defined period and offset in milliseconds.
milliseconds(*period*, *startingAtNanosecond*)	Selects milliseconds with optionally defined period and offset in nanoseconds.
time(*value*[*s*], *separating*)	Selects the specified moment values (if *separating* flag is *false*, as by default) or the intervals between those moments (if *separating* flag is *true*).
visibleTimeline(*type*)	Selects the same intervals as an interval selector created for the specified *type*, except that they would be trimmed to the visible timeline in the chart area.

Label generators

You can set up a label generator for a header row using an object that implements *LabelGenerator* protocol, such as *FormattedTimeLabelGenerator*, *FormattedTimeIntervalLabelGenerator*, *DurationTimeLabelGenerator*, *DurationTimeIntervalLabelGenerator*, or a functionally customizable *TimeLabelSource*.

FormattedTimeLabelGenerator and *FormattedTimeIntervalLabelGenerator* return label values using a *DateFormatter*. The latter returns texts identifying both the start and the finish times of the interval that the label refers to (separated by a dash or by the specified optional separator):

```
let dateFormatter = DateFormatter()
dateFormatter.dateFormat = "yyyy-MM-dd"
let labelGenerator = FormattedTimeLabelGenerator(dateFormatter)

let intervalLabelGenerator = FormattedTimeIntervalLabelGenerator(dateFormatter,
    separator: "/")
```

DurationTimeLabelGenerator and *DurationTimeIntervalLabelGenerator* return label values by counting intervals, considering a reference time, an optional schedule, and returning either one-based (default) or zero-based values, and excluding (default) or including negative numbers. The latter returns texts identifying both the start and the finish times of the interval the label refers to (separated by a dash or by the specified optional separator):

```
let labelGenerator = DurationTimeLabelGenerator(
    reference: projectStart,
    in: .hours,
    schedule: .standard,
    zeroBased: true,
    includingNegativeValues: true)

let intervalLabelGenerator = FormattedTimeIntervalLabelGenerator(
    reference: projectStart,
    in: .hours,
    schedule: .standard,
    zeroBased: true,
    includingNegativeValues: true,
    separator: "/")
```

For convenience, appropriate label generators are however available when calling a shortcut *TimeSelector* initializer with *format*, *locale*, *intervals*, and optional *separator* values. The table below indicates the returned generator for specific *format* arguments:

dateTime	Short date and time styles.
longDate	Full date style and no time value.
date	Medium date style and no time value.
shortDate	Short date style and no time value.
time	Medium time style and no date value.
shortTime	Short time style and no date value.

day	"d" format string (1-31 days).
dayWithLeadingZero	"dd" format string (01-31 days).
dayMonth	"d MMMM" format string (e.g. “19 March”).
dayMonthYear	"d MMMM yyyy" format string (e.g. “19 March 2019”).
dayOfYear	"D" format string (1-366 days).
dayOfYearWidthLeadingZeroes	"DDD" format string (001-366 days).
dayOfWeek	"EEEE" format string (e.g. “Tuesday”).
dayOfWeekShortAbbreviation	"EEEEE" format string (e.g. “T”).
dayOfWeekAbbreviation	"EEEEEE" format string (e.g. “Tu”).
dayOfWeekLongAbbreviation	"E" format string (e.g. “Tue”).
weekOfMonth	"W" format string (1-5 weeks).
weekOfYear	"w" format string (1-53 weeks).
weekOfYearWithLeadingZero	"ww" format string (01-53 weeks).
month	"MMMM" format string (e.g. “March”).
monthShortAbbreviation	"MMMMM" format string (e.g. “M”).
monthAbbreviation	"MMM" format string (e.g. “Mar”).
monthNumber	"M" format string (1-12 months).
monthNumberWithLeadingZero	"MM" format string (01-12 months).
monthYear	"MMMM yyyy" format string (e.g. “March 2019”).
quarter	"QQQ" format string (e.g. “Q1”).
quarterNumber	"Q" format string (1-4 quarters).
quarterYear	"QQQ yyyy" format string (e.g. “Q1 2019”).
year	"yyyy" format string (e.g. “2019”).
yearOfCentury	"yy" format string (00-99 years).
periodOfDay	"a" format string (AM, PM periods).
hour	"H" format string (0-23 hours).
hourWithLeadingZero	"HH" format string (00-23 hours).
hourOfPeriod	"h" format string (1-12 hours).
hourOfPeriodWithLeadingZero	"hh" format string (01-12 hours).
minute	"m" format string (0-59 minutes).
minuteWithLeadingZero	"mm" format string (00-59 minutes).

second	"s" format string (0-59 seconds).
secondWithLeadingZero	"ss" format string (00-59 seconds).
decisecond	"S" format string (1 fractional digit of second value).
centisecond	"SS" format string (2 fractional digits of second value).
millisecond	"SSS" format string (3 fractional digits of second value).
secondWithDecisecond	"ss.S" format string (00.0-59.9 seconds).
secondWithCentisecond	"ss.SS" format string (00.00-59.99 seconds).
secondWithMillisecond	"ss.SSS" format string (00.000-59.999 seconds).
numeric(*reference*, *in unit*, *schedule*, *zeroBased*, *includingNegativeValues*)	Numbers representing the durations passed since a reference time, using a specific time unit, and optionally considering a schedule for measuring the passing time. The values can be set as well to be zero-based (by default they are one-based) and optionally the negative values may be returned as labels as well (by default negative values are omitted).
none	No text label.
string (e.g. "dd-MM-yyyy")	Using the specified formatter string (e.g. "19-03-2019").
not specified	Default format based on interval type — see the following section for details.

Default formats

If the format is not specified when you use shortcut *TimeSelector* initializers there are defaults that would apply automatically, depending on passed interval *type*, as indicated in the table below:

days	*dayWithLeadingZero*
weeks	*shortDate*
monthly	*monthYear*
quarters	*quarterYear*
years	*year*
decades, *centuries*, *millennia*	*year*
halfdays	*periodOfDay*
hours	*hourWithLeadingZero*
minutes	*minuteWithLeadingZero*
seconds	*secondWithLeadingZero*

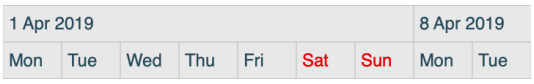
deciseconds	*decisecond*
centiseconds	*centisecond*
milliseconds	*millisecond*
time(*separating*: *true*)	*dateTime*
time(*separating*: *false*)	*none*
visibleTimeline(*type*)	Default format for the specified interval *type*.

Shortcut initializer call examples

As mentioned, header row sets may be easily created using *TimeSelector* shortcut initializers. Here are several call examples with screenshots (although not a complete list — a full comprehensive collection could only be obtained by combining all possible interval types and formats presented in the previous sections.)

Each set is actually going to be passed as an array argument to the *TimeSelector* initializer, but the actual call is removed for brevity from the table below:

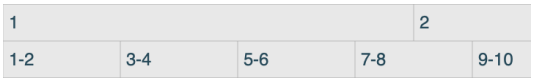
```
headerController.rows = […]
```

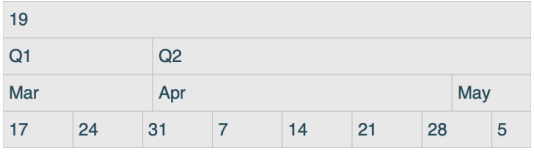
`.weeks(startingOn: .monday), format: .date` `.days, format: .dayOfWeekLongAbbreviation`	

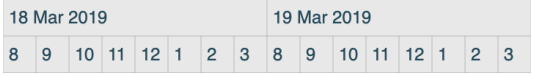
`.visibleTimeline(.months, format: .month)`
`.visibleTimeline(.weeks,`
`  format: .dayWithLeadingZero)`
`.days, format: .dayOfWeekShortAbbreviation`
	

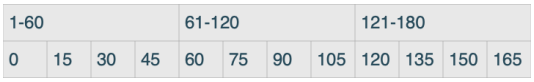
`.days(period: 5), format: .dayMonth` `.days, format: "EEEEE-d"`	

`.weeks(startingOn: .monday)`
`.days, format: .numeric(`
`  in: .days, schedule: .standard)`
	

`.weeks, format: .numeric(in: .weeks)`
`.days(period: 2), format: .numeric(`
`  in: .days, intervals: true)`
	

`.years, format: .yearOfCentury`
`.quarters, format: .quarter`
`.months, format: .monthAbbreviation`
`.weeks, format: .day`
	

`.days(startingAtHours: 8), format: .date` `.hours, format: .hourOfPeriod`	

`.hours, format: .numeric(`
`  in: .minutes, intervals: true)`
`.minutes(period: 15), format: .numeric(`
`  in: .minutes, zeroBased: true)`
	

Dynamic rows

If you want the diagram to update header rows depending on the actual zoom level at any certain time, you can configure a *rowSelector* for your header controller this way:

```
class ViewController: NSViewController, GanttChartHeaderRowSelector {
    override func viewDidLoad() {
        let headerController = GanttChartHeaderController(…)
        headerController.rowSelector = self
        …
    }
    func rows(for hourWidth: Double) -> GanttChartItem {
        if hourWidth < 7.5 {
            return [
                GanttChartHeaderRow(.months),
                GanttChartHeaderRow(.weeks, format: "dd"),
                GanttChartHeaderRow(.days, format: .dayOfWeekShortAbbreviation)]
        } else {
            return [
                GanttChartHeaderRow(.weeks, format: "dd MMM"),
                GanttChartHeaderRow(.days, format: .dayOfWeekShortAbbreviation)]
        }
    }
}
```

rows function (defined by *GanttChartHeaderRowSelector* protocol) should return an array of *GanttChartHeaderRow* objects depending on the actual *hourWidth* value (which changes upon zoom level changes) received as input argument.

Of course, you can reuse the same header row type (such as the *days* header above) or you can return different header rows and even a different number of them (such as 2 or 3 above) when *hourWidth* falls in a certain range, as needed. Note that header row height will update automatically in case the header count changes to allow all rows to still fit into and span all the original header space.

Chart intervals

When you set up header rows their settings only apply for the header area itself. If you want to have vertical separators or highlighting settings applying to specific intervals in the main chart area too, you will need to set *intervalHighlighters* property of *GanttChartContentController*, and you would need to rely again on one or more *TimeSelector* instances.

TimeSelector type, already described in the previous section, is used here to select the intervals to be highlighted in the chart area. You would usually copy the values that you have set on the main header rows (although it's not technically required), but you wouldn't usually display text labels in the main chart area, though (while texts are, nevertheless, supported):

```
contentController.intervalHighlighters = [
    TimeSelector(.months), TimeSelector(.weeks),
    TimeSelector(.time)]
```

Note that by adding a *time* based selector (like in the example above), you would configure the chart to render an automatically updating vertical line for the current time, too.

Schedule based highlighting

If you want to highlight entire working or nonworking time of a schedule object, you can set *scheduleHighlighters* property of *GanttChartContentController*, using an array of *ScheduleTimeSelector* objects. Technically you can highlight working or nonworking intervals generated by multiple schedules, but you would usually select only one for this purpose.

ScheduleTimeSelector type requires you to identify a schedule object upon initialization, and you can indicate whether you want to select working or nonworking time for that definition by passing it with *timesOf* or *timeoutsOf* label, respectively:

```
let nonworkingTimeHighlighter = ScheduleTimeSelector(timeoutsOf: customSchedule)
contentController.scheduleHighlighters = [nonworkingTimeHighlighter]
```

You may also use a shortcut *ScheduleTimeSelector* initializer, allowing you to easily select one of these enumeration values as argument, instead:

```
contentController.scheduleHighlighters = [ScheduleTimeSelector(.weekends)]
```

workdays	Standard working time (8 AM to 4 PM, Monday to Friday).
workbreaks	Standard nonworking time (midnight to 8 AM and 4 PM to next midnight, Monday to Friday).
days	Standard working time for all week days (8 AM to 4 PM, each day).
nights	Standard nonworking for all week days (midnight to 8 AM and 4 PM to next midnight, each day).
weekdays	All time of standard working days (24h, Monday to Friday).
weekends	All time of standard nonworking days (24h, Sunday and Saturday).

Settings and styling

Multiple settings are available to allow you to configure the Ganttis components, being accessible either directly through controller properties or from *settings* properties of the objects:

```
contentController.value = …
headerController.value = …

contentController.settings.value = …
headerController.settings.value = …
```

Some settings, however, are (also) available for individual items or dependencies:

```
item.settings.value = …
dependency.settings.value = …
```

Options

Multiple options are available to select at development time. You can define custom activation and editing functions for items and dependencies, show or hide specific elements in the user interface, enable or disable features for the components, and/or assign complex built-in (or even custom) behaviors to run when items or dependencies are updated by the end user.

Activating items

You may define activation actions for item bars, dependency lines, and for the empty area of the chart, to be run upon clicking, double clicking, or tapping, as configured by the developer and as available on the target platform, by setting content controller's *activator* property to an object that conforms to *GanttChartContentActivator* protocol:

```
class ViewController: NSViewController, GanttChartContentActivator {
    override func viewDidLoad() {
        let contentController = GanttChartContentController(…)
        contentController.activator = self
        …
    }
    func activate(bar: GanttChartBar) {
        let item = bar.item
        …
    }
    func activate(dependencyLine: GanttChartDependencyLine) {
        let dependency = dependencyLine.dependency
        …
    }
    func activate(position: GanttChartPosition) {
        let row = position.row, time = position.time
        …
    }
}
```

Editing items

You may define edit actions for items and dependencies presented by the chart component, to be run upon selecting Edit actions in the contextual menus and upon creating new items, by setting content controller's *editor* property to an object that conforms to *GanttChartContentEditor* protocol. The Edit actions will appear, however, only if the controller's *allowsEditingElements* and *allowsEditingItems/allowsEditingDependencies* properties are set to true:

```
class ViewController: NSViewController, GanttChartContentEditor {
    override func viewDidLoad() {
        let contentController = GanttChartContentController(…)
        contentController.editor = self
        …
    }
    func edit(item: GanttChartItem) {
       …
    }
    func edit(dependency: GanttChartDependency) {
       …
    }
}
```

edit functions should allow the end user to update the appropriate objects. Note that when managed properties change (either updated by these functions or by Ganttis internal handlers for drag and drop editing operations), *hasChanged* flags of the items and dependencies are set to *true*. If you use *GanttChartItemSource* for item management, you can reset them back to *false* in bulk (e.g. at save time) by simply calling its *acceptChanges* method.

Showing/hiding elements

To configure the elements to be shown or hidden on Gantt chart content component, or to override the settings for individual item and dependency instances use these main settings:

GanttChartContentController properties:

preferredTimelineMargin	Indicates the horizontal space size that is to be appended to the left and right margins of the preferred timeline (when no scrollable timeline value is specified).
showsAttachments	Indicates whether to show attachment labels to the right side of the bars in the diagram. Default: *true*.
viewportExtensionWidth, *viewportExtensionHeight*	Indicate the horizontal and vertical extensions to be considered for the visible timeline upon filtering items and dependencies that are to be drawn within the viewport (such as to be able to draw marginal bars and dependency lines).
timeScale	Indicates the time granularity to be used upon updating the time values for items in the diagram. By default it is set to the continuous scale, indicating that any time is acceptable. Other values may be defined using *intervalsWith*(*period*, *in unit*, *origin*, *rule*), where period is the time scale *period*, given in the specified *unit*, optionally considering the *origin* date and time, and the specified floating point rounding *rule*.
timeScaleSchedule	Optional schedule to use when applying time granularity upon initializing or updating the time values for items in the diagram.

GanttChartContentController settings:

showsLabels	Indicates whether to show labels on bars in the diagram. Default: *true*.
showsToolTips	Indicates whether to show tooltips when hovering bars in the diagram with the mouse cursor (macOS only). Default: *true*.
showsCompletionBars	Indicates whether to show completion bars for the bars in the diagram. Default: *true*.
showsDependencyLines	Indicates whether to show dependency lines (between bars of dependent items) in the diagram. Default: *true*.

minBarWidth	The minimum bar width to use regardless of zoom level, for the very short and momentary non-milestone items.
temporaryBarWidth	The bar width to use for an item that is about to be created (while displaying the context menu that allows the item's creation).

GanttChartItem settings:

isHighlighted	Indicates that the bar should be highlighted in the diagram (regardless of focus). Useful (optionally in conjunction with *dependency.settings.isHighlighted*) to highlight chains of Gantt Chart items (and dependencies). Default: *false*.

GanttChartDependency settings:

isHighlighted	Indicates that the dependency line should be highlighted in the diagram (regardless of focus). Useful (optionally in conjunction with item.settings.isHighlighted) to highlight chains of Gantt Chart (items and) dependencies. Default: *false*.

Enabling/disabling features

To configure the enabled and disabled features of Gantt chart header and content components, or to override the settings for individual item and dependency instances use these settings:

GanttChartHeaderController settings:

allowsZooming	Indicates whether to allow zooming operations on the diagram header using dragging. Default: *true*.
usesCache	Indicates whether to use an internal cache for certain user interface related computed values in order to improve the runtime performance of the component. (When false, the cache is cleared upon each drawing operation.) Default: *true*.

GanttChartContentController settings:

allowsZooming	Indicates whether to allow zooming operations on the diagram using pinch gestures (iOS only — on macOS zooming can be enabled and disabled from *GanttChartHeaderController*'s settings). Default: *true*.
allowsActivatingBars	Indicates whether to run activation actions for bars in the diagram. Default: *true*.
isReadOnly	Indicates whether the diagram elements are all read only. Default: *false*.

isTypeReadOnly[*type*]	Indicates whether the diagram elements for a specific item type are all read only.
allowsActivatingDependencyLines	Indicates whether to run activation actions for dependency lines in the diagram. Default: *true*.
allowsMovingBars	Indicates whether to allow moving bars horizontally in the diagram using dragging operations. Default: *true*.
allowsMovingBarsForType[*type*]	Indicates whether to allow moving specific types of bars horizontally in the diagram using dragging operations when allowsMovingBars is set to true.
allowsResizingBars	Indicates whether to allow resizing bars horizontally in the diagram using dragging operations. When setting this property, allowsResizingBarsAtStart is also set to the same value. Default: *true*.
allowsResizingBarsForType[*type*]	Indicates whether to allow resizing specific types of bars horizontally in the diagram using dragging operations when allowsResizingBars is set to true. When setting this property, allowsResizingBarsForType is also set to the same value.
allowsResizingBarsAtStart	Indicates whether to allow resizing bars horizontally at their start in the diagram using dragging operations when allowsResizingBars is true. Default: *true*.
allowsResizingBarsAtStartForType[*type*]	Indicates whether to allow resizing specific types of bars horizontally at their start in the diagram using dragging operations when allowsResizingBars and allowsResizingBarsAtStart are set to true.
allowsMovingBarsVertically	Indicates whether to allow moving bars vertically in the diagram using dragging operations. Default: *true*.
allowsMovingBarsVerticallyForType[*type*]	Indicates whether to allow moving specific types of bars vertically in the diagram using dragging operations when allowsMovingBarsVertically is set to true.
allowsResizingCompletionBars	Indicates whether to allow resizing completion on bars in the diagram using dragging operations. Default: *true*.
allowsResizingCompletionBarsForType[*type*]	Indicates whether to allow resizing completion on specific types of bars in the diagram using dragging operations when allowsResizingCompletionBars is set to true.
preservesCompletedDurationUponResizingBars	Indicates whether completion duration should be preserved, when possible, upon resizing bars. By default it is set to true; set this to false in order to preserve the completion percent upon resizing bars, instead.

allowsCreatingBars	Indicates whether new bars (and bound items) can be created in the diagram (and the associated collection) by right clicking empty area on macOS or long pressing it on iOS, and selecting Create item or Create milestone from the contextual menu. Default: *true*.
allowsCreatingMilestones	Indicates whether milestone can be created (or only standard items are permitted) when allowsCreatingBars is set to true. Default: *true*.
allowsDeletingBars	Indicates whether existing bars (and bound items) can be deleted from the diagram (and the associated collection) by right clicking the bar on macOS or tapping it on iOS, and selecting Delete item from the contextual menu. Default: *true*.
allowsDeletingBarsForType[*type*]	Indicates whether existing bars of specific types (and bound items) can be deleted from the diagram (and the associated collection) by right clicking the bar on macOS or tapping it on iOS, and selecting Delete item from the contextual menu.
allowsCreatingDependencyLines	Indicates whether new dependency lines (and bound dependencies) can be created in the diagram (and the associated collection) by dragging from a temporary thumb that appears to the right or left side of a bar to another bar. Default: *true*.
allowsCreatingDependencyLinesForType[*type*]	Indicates whether new dependency lines (and bound dependencies) can be created in the diagram (and the associated collection) as a link from or to a specific item type.
allowsCreatingDependencyLinesFromStart	Indicates whether dependency lines can be created from bar start areas (fromStartTo* dependency types) when allowsCreatingDependencyLines is set to true. Default: *true*.
allowsCreatingDependencyLinesToFinish	Indicates whether dependency lines can be created to bar finish areas (from*ToFinish dependency types) when allowsCreatingDependencyLines is set to true. Default: *true*.
allowsDeletingDependencyLines	Indicates whether existing dependency lines (and bound dependencies) can be deleted from the diagram (and the associated collection) by right clicking the line on macOS or tapping it on iOS, and selecting Delete dependency from the contextual menu. Default: *true*.
allowsDeletingDependencyLinesForType[*type*]	Indicates whether existing dependency lines (and bound dependencies) from or to a specific item type can be deleted from the diagram (and the associated collection).
editsNewlyCreatedItems	Indicates whether to call editing for the newly created item upon adding them to the collection, with support from the editor delegate. Default: *true*.

editsNewlyCreatedDependencies	Indicates whether to call editing for the newly created dependencies upon adding them to the collection, with support from the editor delegate. Default: *true*.
allowsSelectingElements allowsEditingElements, allowsEditingItems, allowsEditingDependencies	Indicate whether to allow selecting and editing elements when an editor delegate is defined (bars and dependency lines) in the diagram; set them to true if you want end users to be able to select and perform edit operations for the elements in the diagram, respectively. Defaults: *false* for generic elements based settings, *true* for specific items/dependencies settings.
selectsNewlyCreatedElements	Indicates whether a newly created element becomes automatically selected if allowsSelectingElements is set to true. Default: *false*.
selectsEditedElements	Indicates whether an edited element becomes automatically selected if allowsSelectingElements is set to true. Default: *false*.
numberOfClicksRequiredToActivateElements	Indicates the number of clicks required to activate elements on macOS. By default 1; set it to 2 to get elements activated on double clicks only.
autoScrollMargin, autoScrollInterval	Defines the size of the left, right, top, and bottom areas where hovering during drag and drop operations would perform auto-scrolling, and its periodicity.
autoShiftsScrollableTimelineBy	Indicates whether the scrollable timeline is automatically updated when the end user scrolls to its ends and what time interval it should be shifted by (in seconds.) Default: *nil*.
alternativeRowsOnCount	Indicates whether alternative row style is applied to either even or odd rows depending on the row count. Default: *true*.
usesCache	Indicates whether to use an internal cache for certain user interface related computed values in order to improve the runtime performance of the component. (When false, the cache is cleared upon each drawing operation.) Default: *true*.

GanttChartItem settings:

isReadOnly	If set to true, overrides controller's settings.
allowsMovingBar, *allowsResizingBar*, *allowsResizingBarAtStart*, *allowsMovingBarVertically*, *allowsResizingCompletionBar*, *allowsDeletingBar*	Set in bulk by inverting isReadOnly value. If set to false, override controller's settings.

allowsCreatingDependencyLinesTo, *allowsCreatingDependencyLinesFrom*, *allowsDeletingDependencyLinesTo*, *allowsDeletingDependencyLinesFrom*	Set in bulk by inverting isReadOnly value. If set to false, override controller's settings.

GanttChartDependency settings:

isReadOnly	If set to true, overrides controller's settings.
allowsDeletingDependencyLine	Set by inverting isReadOnly value. If set to false, overrides controller's settings.

Zoom limits

The range of zoom levels from and up to which the end user is allowed to slide when zooming is enabled (such as by performing dragging operations in the header area or by pinching the chart on iOS) can be configured by the developer in the header controller's settings:

```
headerController.settings.minZoom = 0.4
headerController.settings.maxZoom = 8
```

You should carefully select the minimum zoom level because if the end user zooms out to the limit and the resulting chart width becomes less than the actually available width of the view, the chart will simply be left aligned and unused space would remain in the right side — and the end users may consider this as an unexpected behavior of your application.

As previously discussed, note as well that as on iOS there are limits for the total size of the chart area's *UIView*, and therefore you should also carefully select (and throughly test) the maximum zoom limit in conjunction with the *hourWidth* and *scrollableTimeline* of the controller.

Appearance

Main appearance settings can be initialized at header and content controller objects' level. Some values, however, are overridable on item and dependency instances too. To configure them, use the appropriate properties under *settings.style* and *item.style* dictionaries, also available using the following shortcut templates:

```
headerController.style.value = …
contentController.style.value = …
item.style.value = …
dependency.style.value = …
```

GanttChartHeaderController style properties:

backgroundColor, *borders*, *borderColor*, *borderLineWidth*	Properties that control the look of the header area. Borders define where they should appear: left, right, top, bottom, and each may have a different color and line width.
labelBorders, *labelBorderColor*, *labelBorderLineWidth*, *labelForegroundColor*, *labelAlignment*, *verticalLabelAlignment*, *horizontalLabelInset*	Properties that control the look of the header cell labels. Borders define where they should appear: left, right, top, bottom, and each may have a different color and line width. Alignment can be left, center, or right. Vertical alignment can be top or center.
highlightingTimeFillColor	Defines the default highlighting color for scheduled working times in the diagram header (times of schedule) selected by an item in scheduleHighlighters collection of the component.
highlightingTimeoutFillColor	Defines the default highlighting color for scheduled nonworking times in the diagram (timeouts of schedule) selected by an item in scheduleHighlighters collection.
cellStyleSelector	Optionally selects a cell style (overriding the default) given an input time interval argument, the selector that generated it, and the row of the selector.
cellStyle	Defines an optional default cell style used for specific time interval areas in the diagram (instead of the default time area and label style settings) selected by an item in row selector collections of the component.

GanttChartContentController style properties:

backgroundColor, *borders*, *borderColor*, *borderLineWidth*	Properties that control the look of the content area of the chart.
rowBorderColor, *rowBorderLineWidth*	Define the way row separators look like.
barFillColor, *barFillColorForType*[*type*]	Main color of item bars for all or for a specific item type (standard, summary, or milestone.)
secondaryBarFillColor, *secondaryBarFillColorForType*[*type*]	Secondary color for item bars; if specified, it is used as a vertical gradient stop at the top of item bars.

barStrokeColor, *barStrokeColorForType*[*type*], *barStrokeWidth*, *barStrokeWidthForType*[*type*]	Border color and width of item bars for all or for a specific item type (standard, summary, or milestone.)
completionBarFillColor, *completionBarFillColorForType*[*type*], *secondaryCompletionBarFillColor*, *secondaryCompletionBarFillColorForType*[*type*], *completionBarStrokeColor*, *completionBarStrokeColorForType*[*type*], *completionBarStrokeWidth*, *completionBarStrokeWidthForType*[*type*]	Main and secondary (vertical gradient stop) colors, and border color and width of item completion bars for all or for a specific item type (standard, summary, or milestone.)
cornerRadius, *completionCornerRadius*	Indicates the roundness of bar and completion bar corners.
verticalBarInset	Determines the vertical positioning of a bar inside the row that contains it. Use a higher value to obtain bars with smaller height. Use zero to have the bar sharing the height of the row.
horizontalCompletionBarInset	Determines the horizontal positioning of a completion bar inside the main item bar that contains it.
verticalCompletionBarInset	Determines the vertical positioning of a completion bar inside the main item bar that contains it. Use a higher value to obtain completion bars with smaller height. Use zero to have the completion bar sharing the height of the main bar.
focusColor, *focusColorForType*[*type*], *focusWidth*, *focusWidthForType*[*type*]	Supplemental border color and width to show on bars for all or for a specific item type (standard, summary, or milestone) when the focus is on, such as upon hovering and while the contextual menu is shown on an item upon right clicking (macOS) or long pressing (iOS).

highlightColor, *highlightColorFor-Type*[*type*], *highlightWidth*, *highlightWidthFor-Type*[*type*]	Supplemental border color and width to show on bars for all or for a specific item type (standard, summary, or milestone) when the highlight is on (item.isHighlighted is set to true.)
selectionColor, *selectionColorFor-Type*[*type*], *selectionWidth*, *selectionWidthFor-Type*[*type*]	Supplemental border color and width to show on bars for all or for a specific item type (standard, summary, or milestone) when the selection is on. Applicable only when allowsSelectingElements setting is true.
labelForegroundColor, *labelForegroundColorFor-Type*[*type*], *labelAlignment*, *labelFont*, *horizontalLabelInset*, *milestoneHorizontalLabelInset*, *verticalLabelInset*	Properties that control the look of the text labels shown on item bars for all or for a specific item type (standard, summary or, milestone). Alignment can be left, center, or right.
attachmentForeground-Color, *attachmentFont*, *horizontalAttachment-Inset*, *verticalAttachmentInset*	Properties that control the look of the text labels shown to the right side of item bars.
temporaryBarColor	The color of the temporary bar shown when an item is about to be created.
dependencyLineColor, *dependencyLineWidth*	Properties that control the look of dependency lines.
dependencyLineFocus-Width	Supplemental width to add to dependency lines when the focus is on, such as upon hovering and while the contextual menu is shown on an item upon right clicking (macOS) or long pressing (iOS).
dependencyLineHighlightWidth	Supplemental width to add to dependency lines when the highlight is on (dependency.isHighlighted is set to true.)
dependencyLineSelectionWidth	Supplemental width to add to dependency lines when the selection is on. Applicable only when allowsSelectingElements setting is true.

dependencyLineThumbColor, *dependencyLineThumbRadius*	Properties that control the look of temporary thumbs (circles) appearing at the ends of item bars when hovering (on macOS) or tapping (on iOS) those areas, allowing the end user to create new dependencies by dragging arrow lines towards other items.
dependencyLineArrowWidth, *dependencyLineArrowLength*, *dependencyLineEndLength*	Properties that control the drawing details of the dependency arrow lines.
temporaryInvalidDependencyLineColor, *temporaryDependencyLineColor*	Properties that control the temporary arrow line drawn while the dragging operation for creating a new dependency is in progress. An invalid dependency line is drawn as dotted line when the target of the dependency is not yet known or it's not accepted.
isTimeAreaBackgroundExtending	When set to true (default) it would extend the background of the time areas to the space visible below the diagram rows that it spans.
areTimeAreaBordersExtending	When set to true (default) it would extend the vertical borders of the time areas to the space visible below the diagram rows that it spans.
timeAreaBorders, *timeAreaBorderColor*, *timeAreaBorderLineWidth*, *timeAreaForegroundColor*, *timeLabelAlignment*, *timeLabelFont*, *verticalTimeLabelAlignment*, *horizontalTimeLabelInset*, *verticalTimeLabelInset*	Properties that control the look of time areas (the vertical time separators) generated by interval highlighters defined on the content controller, and of their labels if they are set up to be generated.
rowStyleSelector	Optionally selects a row style (overriding the default rowStyle or alternativeRowStyle) given an input row argument.
rowStyle	Defines the default row style in the diagram.
alternativeRowStyle	Specifies the row style for alternative rows (overriding the default rowStyle on each second row, starting with first if the total row count is even, or with a virtual row before the first if the count is odd, if alternativeRows and settings.alternativeRowsOnCount are set to true); by default it is set to only define a highly transparent gray background.

alternativeRows	Indicates whether to use different style for alternative rows or not; by default it is set to true.
itemStyleSelector	Optionally selects an item style to use for a bar in the diagrams (instead of the default bar style settings), given an item input argument.
dependencyStyleSelector	Optionally selects a style to use for a dependency line in the diagrams (instead of the default dependency line style settings), given an input dependency argument.
highlightingTimeFillColor	Defines the default highlighting color for scheduled working times in the diagram (times of schedule) selected by an item in scheduleHighlighters collection of the component.
highlightingTimeoutFillColor	Defines the default highlighting color for scheduled nonworking times in the diagram (timeouts of schedule) selected by an item in scheduleHighlighters collection.
timeAreaStyleSelector	Optionally selects a time area style (overriding the default) given an input time interval argument and the selector that generated it.
timeAreaStyle	Defines an optional default time area style used for highlighting specific time intervals in the diagram (instead of the default time area and label style settings) selected by an item in intervalHighlighters collection of the component.

GanttChartItem style properties:

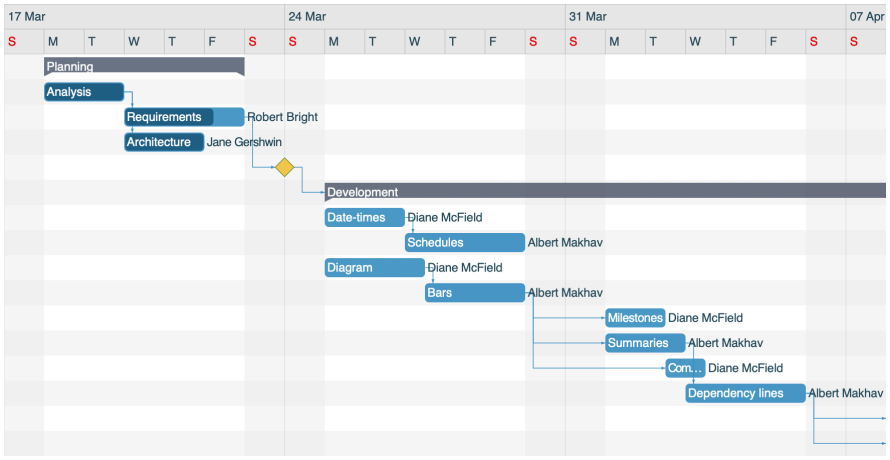
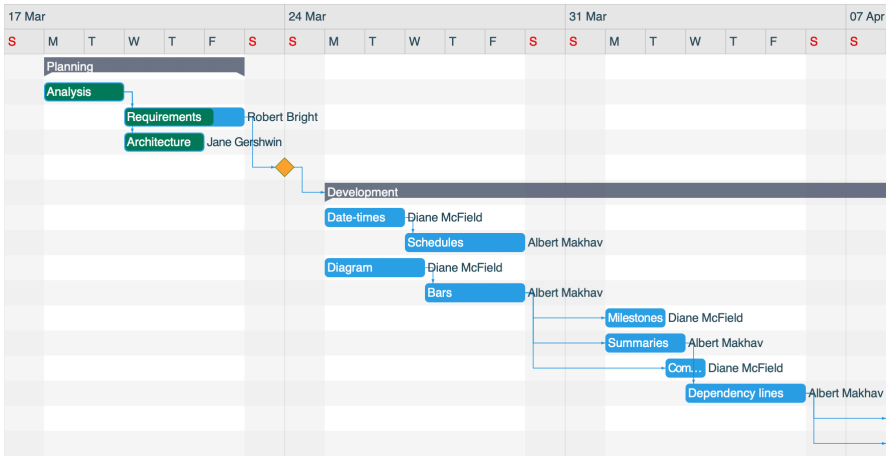
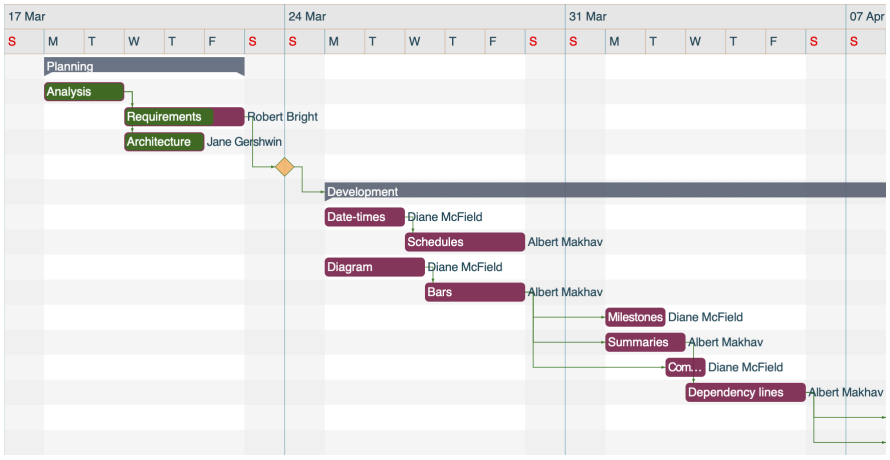
barFillColor, *secondaryBarFillColor*, *barStrokeColor*, *barStrokeWidth*	Override controller's item bar appearance settings.
completionBarFillColor, *secondaryCompletionBarFillColor*, *completionBarStrokeColor*, *completionBarStrokeWidth*	Override controller's item completion bar appearance settings.
labelForegroundColor, *labelFont*, *attachmentForegroundColor*, *attachmentFont*	Override controller's item label appearance settings.

GanttChartDependency style properties:

lineColor, *lineWidth*	Override controller's dependency appearance settings.

Themes

A set of header and content style attributes can be logically reunited in a theme. The framework provides a few built-in themes and supports customizing new ones, both on macOS and iOS. The built-in themes are listed below, with macOS screenshots:

standard	
aqua	
jewel	

To apply a theme you need to set the *theme* property of the target controller object (*GanttChartController* object supports delegating themes to its inner controllers automatically):

```
controller.theme = .jewel
```

Note that a minimal *generic* theme is also available — you can use that one as starting point if you want to restyle all elements from scratch.

After you set a theme, default style attributes of objects accessible from the affected controllers (including the inner controllers of *GanttChartController*) are automatically updated to reflect the new theme setting but *style* dictionary overrides still apply.

If you want to define a custom theme that you could switch to and from with ease, you'd need to define it as base style attribute values under each of the targeted controller objects and bind them to the same custom name, and then simply apply the attribute set as a *custom* theme type later:

```
var style = GanttChartContentBaseStyle(.standard)
style.backgroundColor = Color(red: 0.5, green: 0.75, blue: 1, alpha: 0.125)
style.barFillColor = .orange
var headerStyle = GanttChartHeaderBaseStyle(.standard)
headerStyle.labelForegroundColor = Color(red: 0.25, green: 0.5, blue: 0.75)
contentController.setStyleForTheme("My", to: style)
headerController.setStyleForTheme("My", to: headerStyle)

controller.theme = .custom(name: "My")
```

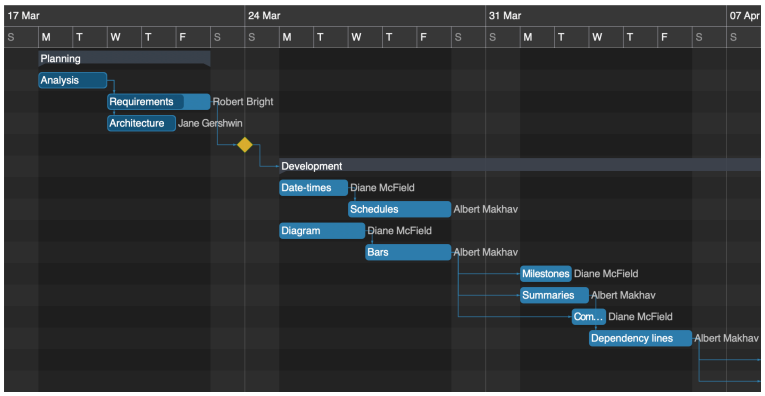
Modes

Since macOS 10.14 and iOS 13 Apple supports dark mode in the operating systems. By default Ganttis automatically matches the style attributes to the current system appearance.

The developer may override the automatically determined mode (obtained by checking *effectiveAppearance* property of the root *NSView* on macOS or by looking at *UIView*'s *traitCollection.userInterfaceStyle* on iOS) using *mode* property available at controller level.

The supported modes are *light* and *dark*, the former being the default when the system doesn't provide mode selection itself:

```
controller.mode = .dark
```



You can also define dark mode styles for your custom themes, if needed, and they will be automatically be picked up when the current mode value of the component becomes dark (this examples continues the initialization lines of code from the previous one):

```
var darkStyle = GanttChartContentBaseStyle(.standard, mode: .dark)
darkStyle.backgroundColor = Color(red: 0.25, green: 0.4, blue: 0.5, alpha: 0.25)
darkStyle.barFillColor = .orange
var darkHeaderStyle = GanttChartHeaderBaseStyle(.standard, mode: .dark)
darkHeaderStyle.labelForegroundColor = Color(red: 0.5, green: 0.75, blue: 0.9)
contentController.setStyleForTheme("My", mode: .dark, to: darkStyle)
headerController.setStyleForTheme("My", mode: .dark, to: darkHeaderStyle)
```

Note that if you update *defaultStyle* attributes, the previously loaded theme won't react anymore to appearance changes at system level. Therefore, the preferred and recommended way to define custom styles and still support mode changes is either using *style* dictionary overrides or custom themes with dark mode support, as shown in the examples above.

Finally, it should be stated that dark mode is supported for Ganttis components also when they are run on previous macOS versions or on iOS, but there you will need to write code to imperatively set the controller's mode property to get it applied at runtime.

Localization

Ganttis components may be localized by customizing the internal string values dynamically at runtime, whenever needed, using the appropriate properties under *settings.strings* dictionary, also available using the following shortcut template:

```
contentController.strings.value = "…"
```

GanttChartContentController string values:

createItem	Label for create item command in chart area's contextual menu.
createMilestoneItem	Label for create milestone command in chart area's contextual menu.
editItem	Label for edit item command in chart item's contextual menu.
deleteItem	Label for delete item command in chart item's contextual menu.
deleteMilestoneItem	Label for delete milestone command in chart item's contextual menu.
editDependency	Label for edit dependency command in chart dependency's contextual menu.
deleteDependency	Label for delete dependency command in chart dependency's contextual menu.

Note that the date and time formats are fully configurable and a secondary *locale* argument allows you to indicate the language that the names of weeks and months should be using whenever a *format* argument is applied (e.g. upon *TimeSelector* shortcut initializers.)

Behaviors

In the framework's context, a behavior is an object that may be applied (and automatically reenforced upon updates) at item collection level — using *GanttChartItemManager.behavior* property — performing specific actions on the managed items and dependencies.

For example, with built-in behaviors you can ensure that:

- a Gantt chart only has one item per row (regardless of vertical drag operations that the end user performs) — *GanttChartItemColumnBehavior*;
- summary items may be expanded and collapsed upon activation (if *contentController.settings.activationTogglesExpansionForSummaryItems* is true), and are automatically scheduled based on and, when updated, scheduling back their affected child items' times — *GanttChartItemHierarchicalBehavior*;
- successor items (considering the available dependencies) are rescheduled when predecessors change, optionally respecting custom dependency lag definitions — *GanttChartItemAutoSchedulingBehavior*;
- specific time constraints on certain items are always respected — *GanttChartItemConstraintBehavior*;
- or a combination of the above — *GanttChartItemBehaviorSet*.

Classic behavior set

You can define a custom behavior by conforming to *GanttChartItemBehavior* protocol, but the built-in support and shortcut setters are often enough. To apply a combination of default behaviors you can use the static *GanttChartItemBehaviorSet.classic* method that allows you to set up arguments to identify which behaviors are included in your customized set:

hierarchyProvider	*GanttChartItemHierarchySource* or *GanttChartItemHierarchy* that define parent-child relationships among items (in a functional or direct manner, respectively.)
autoSchedulingApplyingToUpdatingItems	Indicates whether to auto-schedule items when predecessors change.
autoSchedulingAggregatingSources	Indicates whether to check all source dependencies rather than target dependencies when an item is updated.
autoSchedulingLagProvider	*GanttChartDependencyLagSource* or *GanttChartDependencyLagSet* that define dependency lags (in a functional or direct manner, respectively.)
constraintProvider	*GanttChartItemConstraintSource* or *GanttChartItemConstraintSet* that may define the time constraints for managed items (in a functional or direct manner, respectively.)

preservingDurations	Indicates whether the durations are to be preserved, as much as possible, when applying auto-scheduling behavior(s).

The classic set obtained this way can be used upon *GanttChartItemManager* initialization (*behavior* argument) or set as its *behavior* property later (having *applyBehavior* also called afterwards, if it's necessary to be enforced immediately at that time):

```
itemManager.behavior = GanttChartItemBehaviorSet.classic(…)
```

Item source behavioral settings

However, if you use *GanttChartItemSource* for item management, you can also use these shortcut properties and methods instead of manually initializing a (classic or custom) behavior set:

isColumn	Sets up column constraints (single item per row) applied automatically as an internally managed behavior for the component.
addHierarchicalRelations(*parent*, *items*)	Adds hierarchical relation definitions between the specified parent and child items.
removeHierarchicalRelations(*parent*, *items*)	Removes the hierarchical relation definitions between the specified parent and child items.
removeFromHierarchy(*item*)	Removes hierarchical relation definitions that use the specified item.
isAutoScheduling	Sets up the auto scheduling constraints (based on dependencies) applied automatically as an internally managed behavior for the component.
setLag(*for dependency*, *to value*)	Sets the lag definition for a dependency to a specific value.
removeLag(*from dependency*)	Removes the lag definition from a dependency.
setConstraints(*for item*, *to value*)	Sets the constraint definitions of an item to a specified set of values.
removeConstraints(*from item*)	Removes the constraint definitions of an item.

Diagram algorithms

The internal algorithms used by the component to prepare the diagram are defined by an object that conforms to *GanttChartDiagramGenerator* protocol.

By design, however, the content controller component itself provides the default algorithms, including the dependency line generation one.

Dependency line settings

If you need to control specific aspects of the default dependency line generation algorithm you can use these content controller settings:

drawsVerticallyEndingDependencyLinesWhenApplicable	Indicates whether finish-to-start dependency lines between items that are time-close to each other should be defined by only two segments, ending by a vertical one. By default *true*.
drawsDependencyLinesSpanningHorizontalDistancePrimarilyOnSourceRow	Boolean that indicates whether dependency lines are generated so that they use mostly the source item's row, rather than the target item's row. Useful if multiple dependencies end on the same, and less start from the same item. By default *false*.

Diagram generator

To fully customize the way the diagram is prepared you can set the *diagramGenerator* property of *GanttChartContentController*.

Method *dependencyPolyline* of the *GanttChartDiagramGenerator* object would receive the *start* and *finish* rectangles and the *type* of the dependency to be drawn, along with supplemental flags indicating whether the source and target items are of type milestone, and should return the list of points that the dependency line segments should be drawn through. Note that this way you can only customize the points themselves, not the type of curve that would link them.

For example, assuming that the owner class conforms to *GanttChartDiagramGenerator* protocol and that only finish-to-start dependency lines are allowed in the diagram, you can draw single oblique lines rather than connected horizontal and vertical segments like this:

```
contentController.diagramGenerator = self

func dependencyPolyline(from start: Rectangle, to finish: Rectangle,
                        type: GanttChartDependencyType,
                        fromMilestone: Bool, toMilestone: Bool) -> Polyline {
    return [start.centerRight, finish.centerLeft]
}
```

Custom drawing

While multiple settings of Ganttis components can be easily changed through built-in properties and/or by implementing objects that conform to specific protocols, to update the way user interface components draws content (and, on macOS, also provide tooltips) you will need to inherit from *GanttChart* (or *GanttChartHeader* and/or *GanttChartContent*) views.

To define a custom view inheriting from *GanttChart* component, override the appropriate *draw** and *tooltip** methods — see the following sections. The open method names are self descriptive, and their arguments would be set up by the component for you.

Note, however, that for developing a Cocoa Touch view you would will need to adapt the arguments accordingly (e.g. *CGRect* and *UIColor* instead of *NSRect* and *NSColor*, etc.)

```
class CustomGanttChart: GanttChart {
    override func drawBorder(for row: Row, in rectangle: NSRect,
                             p1: NSPoint, p2: NSPoint,
                             lineWidth: CGFloat, color: NSColor) {
        let border = NSBezierPath()
        border.move(to: p1)
        border.line(to: p2)
        border.lineWidth = lineWidth * 2
        color.setStroke()
        border.stroke()
    }
    override func tooltip(for item: GanttChartItem) -> String? {
       return "Custom tooltip: \(item.details)"
    }
}
```

Layout

Open methods for drawing the main component areas (header, content, and rows):

```
drawHeaderBackground(color: UIColor, size: CGSize)
drawHeaderBorder(in rectangle: CGRect, p1: CGPoint, p2: CGPoint,
                 lineWidth: CGFloat, color: UIColor)
drawContentBackground(color: UIColor, size: CGSize)
drawContentBorder(in rectangle: CGRect, p1: CGPoint, p2: CGPoint,
                  lineWidth: CGFloat, color: UIColor)
drawBackground(for row: Row, in rectangle: CGRect, color: UIColor)
drawBorder(for row: Row, in rectangle: CGRect, p1: CGPoint, p2: CGPoint,
           lineWidth: CGFloat, color: UIColor)
```

Bars

Open methods for drawing the item bars in the chart area — note that the default implementation of the wrapper *draw bar* method will call the following ones:

```
draw(bar: GanttChartBar)
drawBar(for item: GanttChartItem, in rectangle: CGRect,
        fillColor: UIColor, secondaryFillColor: UIColor, strokeColor: UIColor?,
        strokeWidth: CGFloat, cornerRadius: CGFloat,
        isHighlighted: Bool, isFocused: Bool, isSelected: Bool,
        highlightColor: UIColor, focusColor: UIColor, selectionColor: UIColor,
        highlightWidth: CGFloat, focusWidth: CGFloat, selectionWidth: CGFloat,
        allowsMoving: Bool,
        allowsResizing: Bool, allowsResizingAtStart: Bool,
        allowsMovingVertically: Bool, thumbDistance: CGFloat)
drawSummaryBar(for item: GanttChartItem, in rectangle: CGRect,
               fillColor: UIColor, secondaryFillColor: UIColor,
               strokeColor: UIColor?, strokeWidth: CGFloat,
               triangleInset: CGFloat, triangleScale: CGFloat,
               isExpanded: Bool,
               isHighlighted: Bool, isFocused: Bool, isSelected: Bool,
               highlightColor: UIColor,
               focusColor: UIColor, selectionColor: UIColor,
               highlightWidth: CGFloat,
               focusWidth: CGFloat, selectionWidth: CGFloat,
               allowsMoving: Bool,
               allowsResizing: Bool, allowsResizingAtStart: Bool,
```

```
                allowsMovingVertically: Bool, thumbDistance: CGFloat)
drawMilestone(for item: GanttChartItem, in rectangle: CGRect,
              fillColor: UIColor, secondaryFillColor: UIColor,
              strokeColor: UIColor?, strokeWidth: CGFloat,
              isHighlighted: Bool, isFocused: Bool, isSelected: Bool,
              highlightColor: UIColor,
              focusColor: UIColor, selectionColor: UIColor,
              highlightWidth: CGFloat,
              focusWidth: CGFloat, selectionWidth: CGFloat,
              allowsMoving: Bool, allowsMovingVertically: Bool,
              thumbDistance: CGFloat)
drawCompletionBar(for item: GanttChartItem, in rectangle: CGRect,
                  fillColor: UIColor, secondaryFillColor: UIColor,
                  strokeColor: UIColor?, strokeWidth: CGFloat,
                  cornerRadius: CGFloat,
                  allowsResizing: Bool, thumbDistance: CGFloat)
drawBarLabel(for item: GanttChartItem, in rectangle: CGRect, text: String,
             foregroundColor: UIColor, alignment: NSTextAlignment, font: NSFont)
drawAttachmentLabel(for item: GanttChartItem, in rectangle: CGRect,
                    text: String, foregroundColor: UIColor, font: NSFont)
```

Dependency lines

Open methods for drawing the dependency lines in the chart area — note that the default implementation of the wrapper *draw dependencyLine* method will call the following ones:

```
draw(dependencyLine: GanttChartDependencyLine)
drawDependencyLine(for dependency: GanttChartDependency,
                   through points: [CGPoint], color: UIColor, width: CGFloat,
                   arrowWidth: CGFloat, arrowLength: CGFloat,
                   isHighlighted: Bool, isFocused: Bool, isSelected: Bool,
                   highlightWidth: CGFloat,
                   focusWidth: CGFloat, selectionWidth: CGFloat)
drawDependencyLineThumb(for item: GanttChartItem,
                        type: GanttChartDependencyEndType,
                        center: CGPoint, radius: CGFloat, color: UIColor)
drawTemporaryDependencyLine(from: GanttChartItem, to: GanttChartItem?,
                            type: GanttChartDependencyType,
                            through points: [CGPoint], color: UIColor,
                            width: CGFloat,
                            arrowWidth: CGFloat, arrowLength: CGFloat,
                            dashWidth: CGFloat)
drawTemporaryBar(in rectangle: CGRect, color: UIColor,
                 cornerRadius: CGFloat, dashWidth: CGFloat)
```

Time areas

Open methods for drawing the time area in the header and chart areas (including their labels):

```
drawHeaderTimeArea(for highlighter: ScheduleTimeSelector,
                   in rectangle: CGRect, fillColor: UIColor)
drawContentTimeArea(for highlighter: ScheduleTimeSelector,
                    in rectangle: CGRect, fillColor: UIColor)
drawHeaderCell(for selector: TimeSelector, of row: GanttChartHeaderRow,
               in rectangle: CGRect, backgroundColor: UIColor)
drawHeaderCellBorder(for selector: TimeSelector, of row: GanttChartHeaderRow,
                     in rectangle: CGRect, p1: CGPoint, p2: CGPoint,
                     lineWidth: CGFloat, color: UIColor)
```

```
drawHeaderCellLabel(for selector: TimeSelector, of row: GanttChartHeaderRow,
                    in rectangle: CGRect, text: String,
                    foregroundColor: UIColor, alignment: NSTextAlignment,
                    font: NSFont, verticalAlignment: VerticalTextAlignment)
drawContentTimeArea(for highlighter: TimeSelector, in rectangle: CGRect,
                    backgroundColor: UIColor)
drawContentTimeAreaBorder(for highlighter: TimeSelector, in rectangle: CGRect,
                          p1: CGPoint, p2: CGPoint, lineWidth: CGFloat,
                          color: UIColor)
drawContentTimeAreaLabel(for highlighter: TimeSelector, in rectangle: CGRect,
                         text: String, foregroundColor: UIColor,
                         alignment: NSTextAlignment, font: NSFont,
                         verticalAlignment: VerticalTextAlignment)
```

Tooltips

Open methods for providing the content to be displayed as item and dependency tooltips — applicable on macOS only:

```
toolTip(for item: GanttChartItem) -> String?
toolTip(for dependency: GanttChartDependency) -> String?
```

Reloading data

You may reload underlying data and refresh the Gantt chart diagram at any time (such as when you change Gantt chart item values in response to other events than end user's manipulation within the Gantt chart components) by calling this method supported by *GanttChart* and *GanttChartContent* components:

```
reloadData()
```

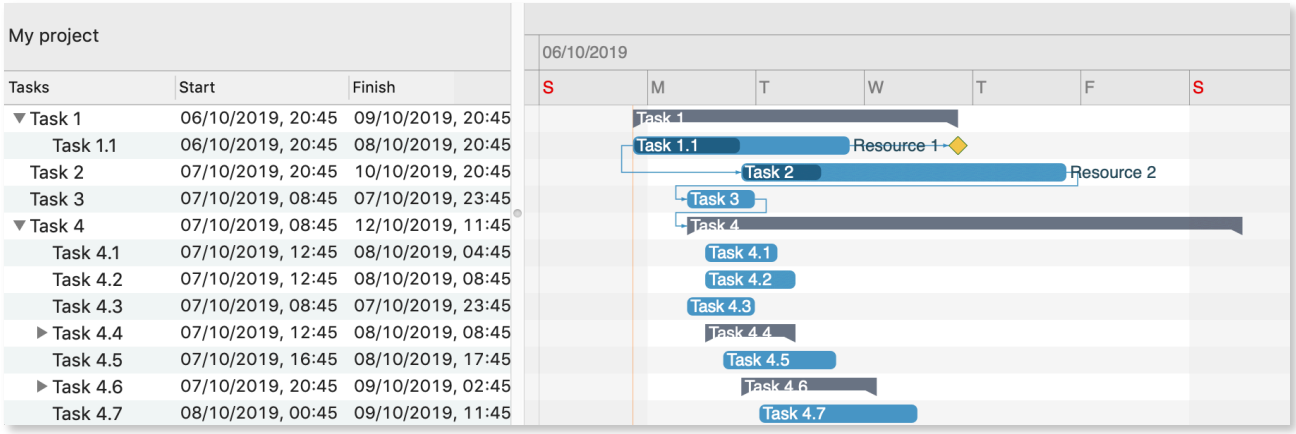
Exporting images

On macOS you can export images with the content displayed by *GanttChart*, *GanttChartHeader*, and *GanttChartContent* components using these read-only properties:

image	The image of the view, as *NSImage* type.
imageRepresentation	The representation of the view, as *NSBitmapImageRep* type.
imageData	Image content using PNG format, returned as *Data* type (easily exportable to a file, for example.)

Outline views

On macOS you can also use an *OutlineGanttChart* component instance to display and manage a Gantt chart diagram in association and synchronized with an *NSOutlineView*, presenting and allowing the end user to expand, collapse, and edit values for hierarchical items associated to the chart rows:



Data source

To set up an *OutlineGanttChart* instance you need to provide a *dataSource* of type *OutlineGanttChartDataSource*, that is partially similar to an *NSOutlineViewDataSource* that would would need to set on a classic *NSOutlineView*.

An *OutlineGanttChart* data set would be defined instantiating these types:

- *OutlineGanttChartRow* — outline view item synchronized to a row in the chart area;
  - *OutlineGanttChartItem* — presented as a bar in the chart area (very similar to *GanttChartItem*), displayed on the row associated to its *OutlineGanttChartRow* container;
  - *OutlineGanttChartDependency* — dependency between two chart items (possibly from different rows), presented as an arrow line in the chart area that connects the linked items' bars (very similar to *GanttChartDependency*).

The data source provider should define these functions (they will be called by the component when specific information is needed):

- *outlineGanttChart(_:child:ofItem)* — should return an *OutlineGanttChartRow* representing the *n*th child item of the specified parent row, or the *n*th root item in the collection (if the received parent item is *nil*);
- *outlineGanttChart(_:isItemExpandable)* — should indicate whether or not a specific outline item (row) is expandable in the hierarchy, i.e. whether it has child items (rows);
- *outlineGanttChart(_:numberOfChildrenOfItem)* — should return the number of child items of the specified parent row, or the number of root items available in the collection (if the received parent item is *nil*);
- *outlineGanttChart(_:objectValueFor:byItem)* — should return the value to be presented in the outline view's cell corresponding to the specified table column (received as *objectValueFor* argument) for the specified row item;

- *outlineGanttChart(_:setObjectValue:for:byItem)* — should save the updated value (received as *setObject* argument) of the outline view's cell corresponding to the specified table column (received as *for* argument) for the specified row item;
- *outlineGanttChart(_:dependenciesFor)* — should return an array of *OutlineGanttChartDependency* objects that refer the specified *OutlineGanttChartItem* objects (received as *dependenciesFor* array argument); this function will be called when a specific set of chart items are known to be displayed in the viewport of the chart area, allowing you to optimize the dependencies retrieved for drawing by limiting them to those that need to appear between those items' bars;
- *outlineGanttChart(_:timeDidChangeFor:from)* — should save the updated time range of a specified *OutlineGanttChartItem* (received as *timeDidChangeFor* argument);
- *outlineGanttChart(_:completionDidChangeFor:from)* — should save the updated completion rate of a specified *OutlineGanttChartItem* (received as *completionDidChangeFor* argument);
- *outlineGanttChart(_:rowDidChangeFor:from:to)* — should save the updated row item of a specified *OutlineGanttChartItem* (received as *rowDidChangeFor* argument) when it moves from an original *OutlineGanttChartRow* container (*from* argument) to another one (*to* argument);
- *outlineGanttChart(_:didAddItem:to)* — should save a newly created *OutlineGanttChartItem* (received as *didAddItem* argument) when it is added to a specified *OutlineGanttChartRow* container (*to* argument);
- *outlineGanttChart(_:didRemoveItem:from)* — should delete an *OutlineGanttChartItem* (received as *didRemoveItem* argument) when it is removed from a specified *OutlineGanttChartRow* container (*from* argument);
- *outlineGanttChart(_:didAddDependency)*, *outlineGanttChart(_:didRemoveDependency)* — should save a newly created *OutlineGanttChartDependency* (received as *didAddDependency* argument) when it is added and delete an *OutlineGanttChartDependency* (received as *didRemoveDependency* argument) when it is removed.

When the data source changes, you should mark the outline Gantt Chart as needing redisplay, so it will reload the data for visible cells and draw the new values and the update diagram by calling *OutlineGanttChart*'s *reloadData()* method.

Settings

An *OutlineGanttChart* instance can configured by using the following settings:

schedule	Defines the schedule to applied to the outline Gantt chart's item manager.

autoApplySchedule	Allows applying the schedule set up on the internal outline Gantt chart's item manager both at initialization time and upon vertical scrolling actions. Default: *false*.
isAutoScheduling	Sets up the auto scheduling constraints (based on dependencies) applied automatically as an internally managed behavior for the component. Default: *false*.
behavior	Defines an optional behavior to be applied to the outline Gantt chart's item manager besides the default hierarchical one, and auto-scheduling if set. If *autoApplyBehavior* is false, the *behavior* is applied to the visible items upon item changes; you may, however, call *applyBehavior* function manually if you want to apply it to the currently visible items immediately.
autoApplyBehavior	Allows applying the behavior set up on the internal outline Gantt chart's item manager both at initialization time and upon vertical scrolling actions. Default: *false*.
isPagingEnabled	Allows loading items with paging by loading subsequent chunks of items only when the end user scrolls down. By default pagination is not enabled.
minPageSize	Defines the minimum number of items of a page when *isPagingEnabled* is set to true. If nil or too low, the number of actually visible items in the viewport is assumed to be a page. Default: *nil*.
minPageCount	Defines the minimum number of pages to be initialized when *isPagingEnabled* is set to true. Default: *2*.

Also, you can refer the following internal components of an *OutlineGanttChart* instance to set up further user interface settings:

splitView	The *NSSplitView* component that separates the Gantt chart (right side) and its associated outline view (left side).
outlineHeaderSpacingBox	An *NSBox* that occupies the space available between the top bound of *OutlineGanttChart* and the top of the internal *outlineView*'s frame which is positioned to ensure outline items and chart rows are vertically synchronized; specifically, when multiple header rows are displayed in the Gantt chart area, the spacing view height will cover the header rows that do not match the single header row of the *outlineView*.
outlineHeaderSpacingLabel	The *NSTextView* component that appears by default within *outlineHeaderSpacingView*, which can be used, for example, to present a title for the diagram.

outlineView, *outlineScrollView*, *outlineClipView*	The *NSOutlineView* component that presents the hierarchy of *OutlineGanttChartRow* objects, and its associated *NSScrollView* and *NSClipView* instances.
ganttChart	The inner *GanttChart* component that presents the diagram defined by *OutlineGanttChartItem* and *OutlineGanttChartDependency* instances.

Technical reference

API documentation for *Ganttis* and *GanttisTouch* modules is available in Xcode *Quick Help*. You can also simply right click on a type or member name in the source code editor and select *Jump to Definition* to view the item among other public *Ganttis* headers.

Support

Remember that you may also contact DlhSoft whenever you have technical (or any other type of) questions related to the product — the team is heading to always answer with the highest possible responsiveness level, and always in full detail so that you would rarely need to ask further questions afterwards.

Licensing

You can order a Ganttis license from DlhSoft Web site. To setup the string code obtained upon purchasing the license to your application, import *Ganttis* or *GanttisTouch* and set the *license* property of the module in your *AppDelegate*'s initializer:

macOS	iOS

`import Ganttis`
`class AppDelegate: NSObject, … {`
`  override init() {`
`  super.init()`
`  Ganttis.license = "…"`
`  }`
`}`
	`import GanttisTouch`
	`class AppDelegate: UIResponder, … {`
	`  override init() {`
	`  super.init()`
	`  GanttisTouch.license = "…"`
	`  }`
	`}`

Objective C considerations

Because Objective C doesn't support setting module properties, to initialize the license value you will need to first define a helper *NSObject* based class using Swift, and then you can simply instantiate it to run the license setup code from within your *AppDelegate* implemented in Objective C:

Swift	Objective C

`public class GanttisLicense: NSObject {`
`  public override init() {`
`  Ganttis.license = "…"`
`  }`
`}`
	`@implementation AppDelegate`
	`- (instancetype)init {`
	`  self = [super init];`
	`  [GanttisLicense new];`
	`  return self;`
	`}`